A Design Environment for Addressing Architecture and Compiler Interactions

JACK W. DAVIDSON
Department of Computer Science, University of Virginia, Charlottesville, VA 22903, U.S.A.

DAVID B. WHALLEY
Department of Computer Science, Florida State University, Tallahassee, FL 32306, U.S.A.

SUMMARY

Often a computer architecture is designed and implemented without determining whether its associated compilers will actually use all of the architecture's features. A more effective machine can result when the interactions between an architecture and a compiler are addressed. This paper presents an environment that integrates the tasks of translating a source program to machine instructions for a proposed architecture, imitating the execution of these instructions, and collecting measurements. The environment, which is easily retargeted and quickly collects detailed measurements, facilitates experimentation with a proposed architecture and a compiler.

KEY WORDS: Computer Architecture Architectural Design Dynamic Analysis

INTRODUCTION

Several factors have caused architectures to be designed and implemented without determining whether a compiler can make effective use of the architecture's features. Many compilers require a significant amount of time to retarget. In the rapidly changing computer industry, manufacturers may not wish to delay the implementation of the initial design for such a time period. Even after a compiler has been retargeted, it is not a simple matter to provide a mechanism to execute instructions and gather measurements on a machine that does not yet exist. Because of the large execution time penalties incurred with some of these mechanisms, experimentation is discouraged. However, despite these difficulties, we believe that better computer systems (both hardware and software) are possible if the architecture is designed to operate synergistically with the compiler. Examples of such systems include the IBM 801 [Rad82] and the MIPS [HJB82] processors. Their designs were influenced largely by the decision to make pervasive use of high-level languages and powerful compilers.

To effectively evaluate a proposed architecture, one should analyze measurements from typical programs that are to be executed by the machine. Three tasks must be accomplished to be able to obtain these program measurements. The first task is generating the machine instructions for the proposed architecture that correspond to each of the test programs. The second task is providing the ability to imitate the execution of these machine instructions since the proposed architecture does not exist. The third task is establishing a method for the extraction of measurements from the execution of the programs.

This paper presents an environment called *ease* (Environment for Architecture Study and Experimentation) that integrates the tasks of producing instructions for a proposed machine, imitating the execution of the instructions, and collecting measurements. Integration of these tasks results in a substantial reduction in effort as compared to traditional methods. Also, by using program-flow information calculated by the compiler, detailed measurements can be obtained with very little overhead.

PRODUCING CODE FOR THE PROPOSED MACHINE

To be able to analyze measurements from representative test programs for a proposed architecture, each of the test programs must first be translated to instructions for the proposed machine. If a set of small test programs is used, the instructions for each of the test programs can be generated by hand. The measurements extracted from these programs would probably not produce representative results since the size of the programs would not be realistic and the quality of the code would depend on the skill of the author of the programs. A more realistic test set would typically require the construction of a compiler. The problem of generating code is further complicated since most machines require ‘‘optimizing’’ compilers in order for their capabilities to be exploited [HeP90].

The compiler technology used in this environment to develop back ends of compilers is known as *vpo* (Very Portable Optimizer) [BeD88, DaF84, Dav86]. The optimizer, *vpo*, replaces the traditional code generator used in many compilers and has been used to build C, Pascal, and Ada compilers.[1] *vpo* is retargeted by supplying a description of the target machine. Using the diagrammatic notation of Wulf [WJW75], Figure 1 shows the overall structure of a set of compilers constructed using *vpo*. Vertical columns within a box represent logical phases which operate serially. Columns divided horizontally into rows indicate that the subphases of the column may be executed in an arbitrary order. IL is the Intermediate Language generated by a front end. Register transfers or register transfer lists (RTLs) describe

[1] While the current version of *ease* only works with C front ends, the environment can be extended for use with other languages when the corresponding front ends are updated to be compatible with the latest version of *vpo*.

the effects of machine instructions and have the form of conventional expressions and assignments over the hardware's storage cells. For example, the RTL

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

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

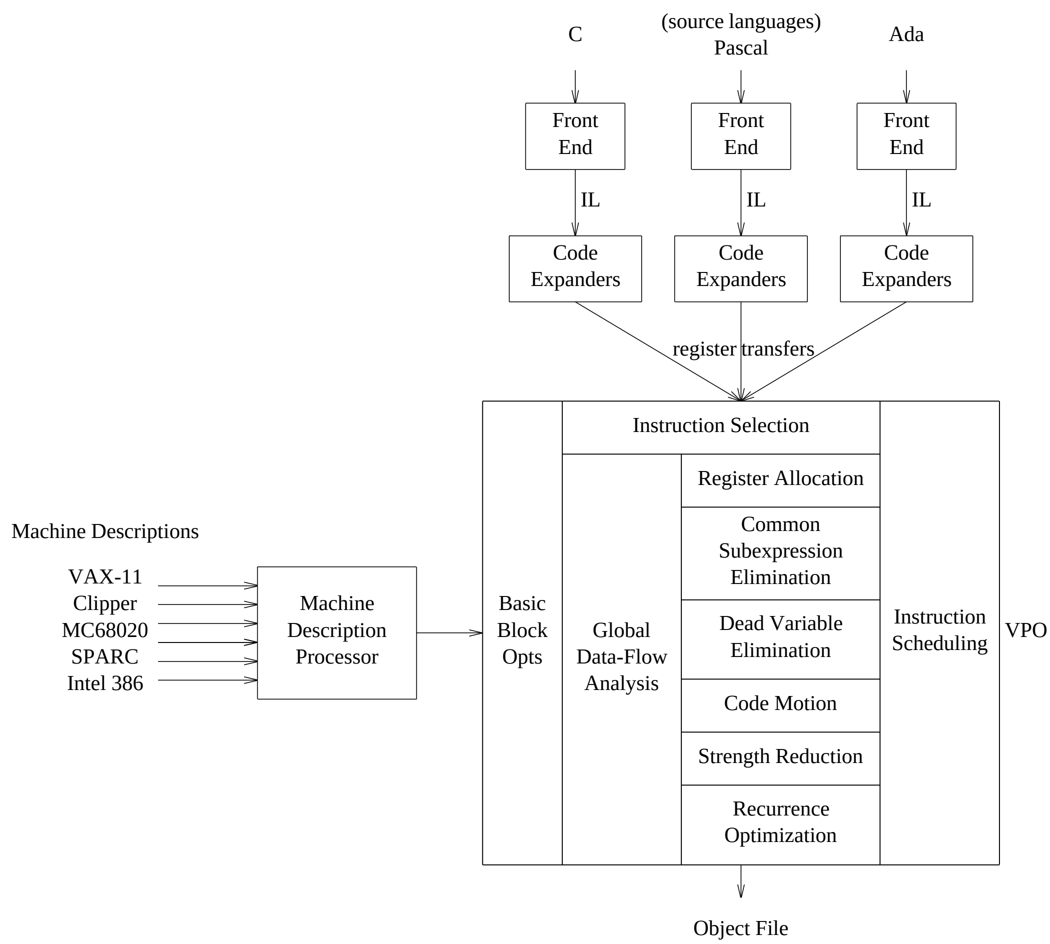

Figure 1: Compiler Structure

There are a number of advantages to using RTLs as the basis of the intermediate representation. Because the form is machine-independent, one can construct programs that manipulate RTLs in machine-independent ways. For example, the phase that performs data-flow analysis on RTLs is largely machine-independent. Because RTLs represent actual machine instructions, details of the target machine are exposed to the various optimization phases, resulting in more complete and thorough optimization. For instance, by performing the optimizations after code generation, all of the instructions that are invariant become candidates for being moved out of loops when code motion is applied. Finally, because RTLs are well-defined, one can easily construct recognizers to determine whether an RTL represents a legal target machine instruction, which is essential to our optimization strategy. For example, an RTL created during instruction selection by combining two or three RTLs together is verified to be legal before the original RTLs are replaced.

To retarget *vpo* to a new architecture, one must write a description of the architecture's instruction set. This machine description consists of a grammar and semantic actions. The grammar is used to produce a parser that checks the syntax of an RTL. The semantic actions check context-sensitive constraints imposed by a particular architecture. Currently, the RTL parsers are constructed using the Unix parser generator *yacc* [Joh78].[2]

Machine description grammars are relatively easy to write [Dav85]. The goal is to compose a grammar with semantic actions to produce a parser that accepts all legal RTLs (instructions) and rejects all illegal RTLs. In our experience it is easier to write a machine description for an instruction set than it is to write a grammar for a programming language. The task is further simplified by the similarity of RTLs across machines, which permits a grammar for one machine to be used as the model for a description of another machine. We have used this technique to describe the instruction sets of the following machines: VAX-11, Motorola 68020/68881, National Semiconductor 32016, Concurrent Computer Corporation 3230, Western Electric 32100, Intel 80386/80387, Harris HCX-9, IBM PC/RT, Intergraph Clipper, SUN SPARC, AT&T DSP32, Hewlett-Packard 800, Motorola 88100, and MIPS R2000. The C

[2] There is a certain appeal to the symmetry of employing the same tool for constructing both the front and back ends of the compiler.

compiler has been ported to new architectures in as little time as two weeks by one experienced with the technology.

IMITATING THE EXECUTION OF CODE FOR THE PROPOSED MACHINE

To be able to evaluate an architecture, one should determine the effect of executing instructions from representative test programs for the architecture. If the architecture has not yet been implemented, then one must imitate this execution by other means. One solution is to use a simulator, which imitates the machine by interpreting the machine instructions [AlW75, BSG77]. Since a simulator can be written in a high-level language, it can be executed on any machine for which there is a compiler for that language. Another solution is to produce a program that translates the assembly instructions for the proposed architecture into assembly instructions for an existing machine. Once translated, the programs can then be executed on the host machine.

There are problems when using either of these schemes. The effort required to construct a simulator is comparable to the effort to construct a compiler. Although not as difficult as constructing a simulator, the effort required to produce a translator to map assembly instructions from one machine to another is nontrivial. The translator must parse and identify the mnemonic of each instruction and the addressing mode of each field in the instruction. The time required to implement these schemes may discourage one from collecting measurements. Furthermore, the total execution time of a simulated program is typically hundreds of times slower than if the program were executed directly [HLT87]. A large execution time penalty can lead to extraction of measurements from a small number of simulated instructions and less meaningful results.

The last step in the *vpo* compilation process is the conversion of an RTL to assembly language for the target machine and emitting it to a file that will be processed by the system's assembler. In the *ease* environment, an instruction for the proposed machine can either be generated as an assembly instruction for the proposed architecture or as one or more equivalent assembly instructions for an existing architecture. As an RTL representing an instruction is parsed, characteristics of the instruction are collected and used for semantic checks. This semantic record can also be used to produce assembly code that corresponds to the RTL. Which assembly code is produced depends upon a switch set when invoking the compiler. The assembly code for the existing architecture can then be assembled, linked, and executed. Figure 2 contains the code that allows the VAX-11 increment instruction to be produced in equivalent SPARC assembly instructions.

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

Figure 2: Code to Generate an Increment Instruction

ease can also be used to emulate architectural features that are not directly equivalent to features on an existing architecture. For instance, the number of available registers on a proposed architecture may exceed the number of registers available on the host machine. In this situation, translation of an RTL to assembly code on a host machine is accomplished in the following manner. First, a set of currently available registers, equal in number to the maximum number of unique registers that could be used in one instruction, are reserved. Any references to the reserved or nonexistent registers are replaced by corresponding memory references. If one of these registers cannot be directly replaced by a memory reference in the instruction (e.g. in an addressing mode), then the value for the register is loaded from memory into a reserved register prior to the execution of the instruction. The reserved register, instead of the memory reference, is then used in the instruction. If one of these registers is updated as a side effect of using a particular addressing mode, then the new value for the register is stored after the instruction. Updating a *vpo* compiler for the VAX-11 to emulate its execution with twice the number of available registers was accomplished in half an hour by adding less than 50 lines of code. An example of translating VAX-11 RTLs referencing additional registers is shown in Figure 3.

```
r[30] = r[30] + 1;     -- add 1 to register 30

=>

incl    R30(fp)        -- increment r[30] memory reference



L[a[r[30]++]] = 0;     -- store 0 in memory at r[30]
                       -- and add 4 to r[30]

=>

movl    R30(fp),r6     -- load r[30] into available register
clrl    (r6)+          -- clear memory and add 4 to register
movl    r6,R30(fp)     -- store updated value of r[30]
```

Figure 3: Translating Additional Registers

EXTRACTING MEASUREMENTS FOR THE PROPOSED MACHINE

To be able to evaluate an architecture effectively, one must examine its behavior when executing real programs. To be able to extract this behavior, one must collect measurements from the program's execution. Collecting these measurements with a simulator is straightforward since the simulator must already recognize the characteristics of each instruction in order to correctly imitate the instruction's effects. As each instruction is interpreted, detailed measurements can be updated. Extracting measurements for a proposed architecture can also be accomplished when the technique of translating target assembly instructions to assembly instructions on a host machine is used. As each assembly instruction is translated, characteristics of the instruction can be recorded. Program instrumentation can be used to modify the translated program by inserting instructions to increment frequency counters or to record other events. The frequency counts, obtained by executing the instrumented program, can be correlated with instruction information that was recorded during the translation to produce accurate measurements. No mapping of assembly instructions is required when the control flow of the basic blocks is identical on both the proposed and existing machines [HLT87]. Frequency counts can be gathered from the program executed on the existing machine and correlated with instruction information gathered from the assembly program on the proposed machine.

The previously mentioned problems with using simulation or assembly translation for imitating the execution of instructions on a proposed machine also apply when extracting measurements. Both a simulator and an assembly translator require a significant amount of effort to construct. Also, a simulated program executes much more slowly than the same program executed directly on the target machine.

Another problem is that these schemes record information on instructions after assembly code has been produced. Thus, many types of measurements related to the source code are not easily extracted. For instance, determining the average number of arguments allocated to registers requires information not readily available in the assembly code. Also, data-flow and control-flow information can be used to minimize the overhead of instructions that are inserted from program instrumentation. This information, accessible in an optimizing compiler, would have to be recalculated if a separate program were used to instrument the assembly program.

A direct mapping between the basic block structure of assembly programs on the proposed architecture and the existing architecture cannot be assured when an optimizing compiler is used. For instance, there is a greater chance that a RISC instruction within a loop will be invariant since more RISC instructions than CISC instructions are typically required to accomplish the same set of operations. Therefore, code motion can often be applied more extensively to loops containing RISC instructions. Applying code motion results in the introduction of a new basic block as a preheader of a loop if no suitable basic block that is a predecessor of the header of the loop can be found. Special instructions available on some architectures to store the results of comparisons for boolean operations and different code generation strategies, such as calling sequence conventions, can also affect the basic block structure. If an optimizing compiler is not used and there are few differences between the proposed architecture and an existing architecture, then it is possible that there would exist a direct mapping between the basic block structures of programs compiled on each machine. Even if a one-to-one mapping exists, then the assembly program for the proposed architecture must still be processed to extract the instruction information required to collect detailed measurements.

The method *ease* uses to collect measurements is to modify the back end of the compiler to store the characteristics of the instructions to be executed and to instrument the assembly code with instructions that will either count the number of times that each instruction is executed or invoke a routine to record events that are dependent on the order of the instructions executed. These modifications have been implemented in the *vpo* compiler system and are described in subsequent sections. This method is illustrated in Figure 4.

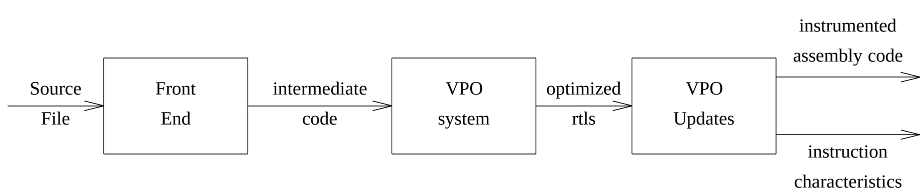

Figure 4: Method for Gathering Data

Instruction Characteristics

The first modification of *vpo* to produce code for collecting 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 several purposes. One use is to provide information to translate the RTL to the corresponding assembly instruction on the target machine or equivalent assembly instructions on a host machine. As each assembly instruction is produced, the characteristics of the instruction are also written to a file by invoking a machine-independent routine. The routine is only invoked if the option for collecting frequency measurements is set and the optimizer had been compiled to allow the collection of measurements. The routine receives the instruction type and the semantic record containing the fields of the instruction. It writes the instruction type and the addressing mode and data type for each field within the instruction to a file. Thus, very little extra code and no extra parsing of RTLs are required to collect this information. These instruction characteristics are emitted as if the target architecture exists. Therefore, the measurements are collected on the RTLs, which represent the target architecture instructions. The translation from RTLs to assembly instructions does not affect the measurements that are obtained. An example of a routine that stores information about a Motorola 68020 call instruction is shown in Figure 5.

```
/*
 * call - check semantics of call
 */
void call(i1)
struct sem_rec *i1;
{
   /* if generating assembly code */
   if (dassem || vaxassem) {
      if (dassem)
         printf("\tjbsr\t%s\n", i1->sem.call.addr->asmb);
      else
         printf("\tcalls\t$0,%s\n", i1->sem.call.addr->asmb);
#ifdef MEASURE
      if (swe)
         stinstinfo(JSBRI, i1);
#endif
      }

   /* else perform semantic checks */
   else {
      ...
      }
}
```

Figure 5: Storing Instruction Information

Program Instrumentation

The second modification to *vpo* is to have the compiler instrument the assembly code after all optimizations have occurred to either increment counters or invoke measurement routines. Counters are used to obtain information that is independent of the order in which the instructions are executed, such as the number of times each type of instruction is executed. Measurement routines are invoked to record order-dependent events, which includes trace generation and analysis of memory references.

a. Collecting Order-Independent Data

To collect order-independent information, *ease* must determine the number of times that each instruction was executed. Within each function there are groups of instructions, called 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. Using control-flow information, *ease* only generates an instruction to increment a counter for each execution class rather than for each basic block.

An example illustrating how control-flow analysis is used to reduce the overhead for collecting frequency measurements is given in Figures 6—9. Figure 6 contains a C function that returns the sum of the elements of an array. Figure 7 gives the Motorola 68020 assembly code produced by *vpo* for that C function. Figure 8 shows the same assembly code broken into basic blocks. Note that although there are five basic blocks there are four execution classes ({1}, {2, 4}, {3}, {5}). Figure 9 shows the modified 68020 assembly code with three instructions inserted to increment counters. The name of the file being compiled, *sum* in this case, is used to distinguish counters from other files in the same executable.

```
int sum(a, n)
int a[], n;
{
   int i, total;

   if (n > 0) {
      total = 0;
      i = 0;
      do
         total += a[i];
      while (++i < n);
      return(total);
      }
   return (-1);
}
```

Figure 6: C function

```
.globl _sum
_sum:
n.=12
a.=8
        link     a6,#-8        --  setup frame and stack ptrs
        moveml   d2/d3,a7@     --  save d2 and d3
        movl     a6@(n.),d3    --  load argument n into d3
        jle      L14           --  if d3 <= 0 goto L14
        clrl     d1            --  clear d1
        clrl     d2            --  clear d2
        movl     a6@(a.),a1    --  load argument a into a1
        lea      a1@,a0        --  move a1 to a0
L17:    addl     a0@+,d1       --  add mem value at a0 to d1;add 4 to a0
        addql    #1,d2         --  add 1 to d2
        cmpl     d3,d2         --  compare d3 and d2
        jlt      L17           --  if d2 < d3 goto L17
        movl     d1,d0         --  move d1 to d0 as return value
        moveml   a7@,d2/d3     --  restore d2 and d3
        unlk     a6            --  restore frame and stack ptrs
        rts                    --  return
L14:    movl     #-1,d0        --  move -1 to d0 as return value
        moveml   a7@,d2/d3     --  restore d2 and d3
        unlk     a6            --  restore frame and stack ptrs
        rts                    --  return
```

Figure 7: Motorola 68020 Assembly Code for Function in Figure 6

Apart from using execution classes, there are additional methods to reduce the number of times that counters need to be incremented. An instruction to increment a counter is not inserted if the execution class count can be inferred from the counts of other execution classes. As shown in Figure 8 basic block 1 has two successors, blocks 2 and 5. Both successors have only one predecessor, block 1. The number of times that block 1 is executed is equal to the sum of the times that blocks 2 and 5 are executed. Therefore, it is not necessary to increment a counter associated with the execution class representing blocks 2 and 4. The number of iterations of the loop containing block 3 can be determined at compile-time to be equal to the value in the register `d3` at the point where the loop is entered. Thus, the counter associated with that loop is incremented by the value in the register in the preheader of the loop.

Another type of frequency measurement collected by *ease* is the number of times that each type of conditional branch is taken. These values can be obtained by inserting an instruction to increment a

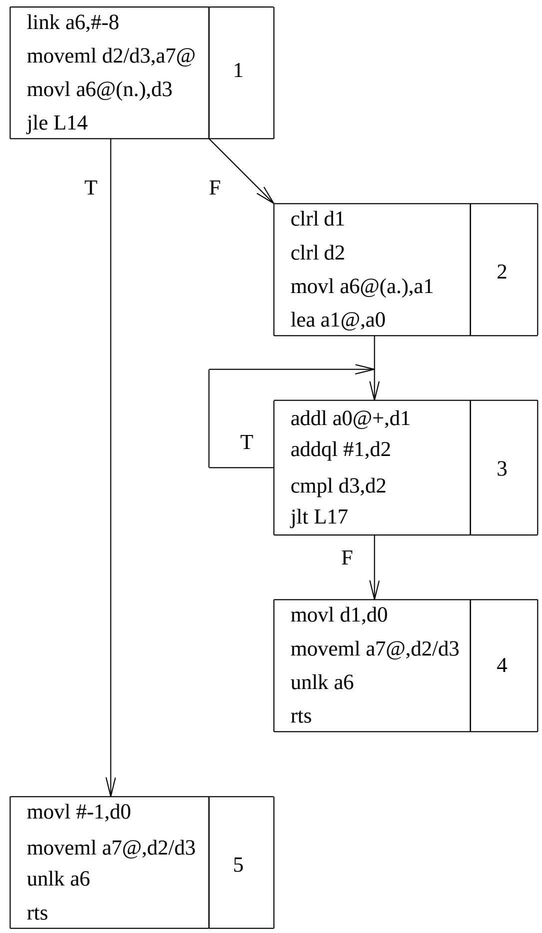

Figure 8: Assembly Code of Figure 6 in Basic Blocks

counter after each conditional branch. The counts can then be subtracted from the number of times that each type of conditional branch is executed to produce the desired measurements. The insertion of such instructions, however, is usually unnecessary. For instance, there is a conditional branch in blocks 1 and 3 in the example shown in Figure 8. In both situations the block that is executed if the conditional branch is not taken (2 and 4) has only one predecessor. Therefore, the number of times that each type of conditional branch is not taken is equivalent to the number of times that blocks 2 and 4 are executed. Occasionally, the block that is executed if the conditional branch is not taken has more than one predecessor. The insertion of an instruction to increment a counter is still unnecessary if the block at the target of the conditional branch has only one predecessor. In this case the number of times that the conditional branch

```
.globl _sum
_sum:
n.=12
a.=8
        addql   #1,_sum_counts
        link    a6,#-8
        moveml  d2/d3,a7@
        movl    a6@(n.),d3
        jle     L14
        clrl    d1
        clrl    d2
        movl    a6@(a.),a1
        lea     a1@,a0
        addql   d3,_sum_counts+8
L17:    addl    a0@+,d1
        addql   #1,d2
        cmpl    d3,d2
        jlt     L17
        movl    d1,d0
        moveml  a7@,d2/d3
        unlk    a6
        rts
L14:    addql   #1,_sum_counts+12
        movl    #-1,d0
        moveml  a7@,d2/d3
        unlk    a6
        rts
```

Figure 9: 68020 Assembly Code with Frequency Counters

is not taken is equal to the execution count for the block containing the conditional branch minus the execution count for the target block.

Instrumenting code naively would result in an instruction being inserted for each basic block and following each conditional jump. Assuming that `n > 0` in the example given in Figure 6, naive instrumentation would result in `n+5` increments of counters (`n+3` increments for the basic blocks and `2` increments following the conditional jumps) for each execution of the function. Only 2 increments for each execution of the function are required when control-flow analysis is used.

Determining whether a block belongs to an execution class is done in three steps. The first step calculates the set of blocks that dominate the current block. This information is already available in *vpo* since dominator information is needed to detect loops in a program [ASU86]. The second step determines if the current block is always a successor to the blocks which dominate it. This is accomplished by determining if all paths from one block eventually lead to the current block. The third step checks if the current block is in the same set of loops as the blocks in the execution class. The information for this step is also already available in *vpo*. Execution class information has also been used in *vpo* to help find instructions to place behind a delayed branch that are not in the current block or an immediately adjacent block. Figure 10 shows the dominators (DOM), always successors (AS), and execution classes (EC) for the set of blocks in Figure 8.

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

```
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) = {5}      EC(5) = {5}
```

Figure 10: Execution Classes for Blocks in Figure 8

b. Collecting Order-Dependent Data

Instructions can also be inserted to invoke routines to record the occurrence of order-dependent events. This strategy has been used to simulate instruction, data, or unified caches. The inserted calls pass information to a cache simulator. Not only does this avoid lengthy trace files, but it also speeds up the execution. Note that scratch registers can potentially be live at the point that instructions are inserted to invoke a trace routine. Using data-flow information calculated by the compiler, *ease* determines which scratch registers are live at the point where each call is inserted and thus avoids unnecessary saves and restores around the inserted calls.

ease inserts instructions to invoke a trace routine at the beginning of each basic block for instruction cache simulation. The instruction addresses are calculated prior to the execution by determining the size of each instruction. By obtaining size information, the addresses of instructions passed to the cache simulator are unaffected by the code that is inserted for capturing measurements. Instruction size information for existing machines is determined by placing a label before and after each instruction. The size of an instruction, calculated by the assembler, is the difference between the two labels.[3] To obtain instruction cache measurements for a proposed architecture, the size of each instruction must be known by the compiler since an assembler would not be available. A unique basic block number is passed to the trace routine, which uses this number to access information associated with that block. The trace routine in turn invokes the cache simulator, passing the beginning address of the basic block and the basic block size to the cache simulator. The number of cache hits is then adjusted to reflect the actual number of instructions in the basic block. If it is determined that a periodic context switch could occur during the execution of the block, then information about each individual instruction within the block is passed to the cache simulator.

For data cache simulation, *ease* invokes the cache simulator directly before each data memory reference. *ease* can easily extract memory references from an RTL and pass them to a cache simulator. For instance, the RTL below describes the effect of a memory to memory move on the 68020.

```
L[A[_b]] = L[A[_a]];
```

ease first examines the `src` from the `dst = src;` of the RTL. If it finds a memory reference, it strips off the memory type character and the outer brackets and uses the resulting string to push it onto the stack. For the previous RTL, a `pea` or push effective address instruction, represented by the following RTL, is inserted.

```
L[A[--a[7]]] = A[_a];
```

Other instructions are inserted to invoke the cache simulator. A similar process then occurs for the `dst` portion of the RTL.

All data declarations used for obtaining measurements are placed in a separate file which is specified to be linked after all the compiled files being measured. Thus, the actual address of each data reference can be passed to the cache simulator since the data declarations for collecting measurements do

[3] This task is unnecessary on most RISC machines since typically each instruction is the same size.

not perturb addresses of other data references in the program. Of course, there may be differences in the addresses of the data references produced when a program is executed on a host machine instead of the actual target machine since the organization of the executable file may differ.

To accurately intermix instruction and data references in a unified cache requires some additional work. Instructions are inserted to store the address for each data memory reference in a buffer immediately before the instruction containing the memory address is executed. The trace routine invoked at the beginning of each basic block will store the basic block number and process the last executed basic block. The number of data memory references associated with each RTL is determined at compile-time. Rather than processing each instruction individually to intermix instructions and data references, the cache simulator is invoked for groups of consecutive instructions within the block. Only the last instruction within a group can contain data memory references. After invoking the cache simulator for a group of instructions, the cache simulator will be invoked for each of the data references associated with the last instruction in the order they were stored. The actual point when the cache simulator should be invoked to process a data reference is machine-dependent since it depends on the pipeline structure of the machine.

Other Modifications

In the past, the code expander generated some instructions in assembly language if optimizations could not affect them. An instruction must be represented as an RTL, however, if its execution characteristics are to be collected. Therefore, every type of instruction that is executed must be described in the machine description.

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 the value in its count operand. This instruction, though its effects could not be accurately described in a single RTL, is still represented by an RTL in a functional notation as shown below.

```
addr = CM(expr, addr);
```

In order to accurately count the number of memory references made by the execution of the instruction, the value of the count operand must be stored with the other characteristics of the instruction.

Processing the Collected Data

At the end of the execution of the program, a routine is invoked that will write the collected measurements (either the frequency counts or the results of a cache simulation) to a file. This is accomplished by inserting the call before any return instructions in the *main* function and before all calls to the *exit* library routine.

The execution counts and the characteristics of the instructions are both used to produce dynamic frequency measurements, but the instruction characteristics can also be used to produce static measurements. Figure 11 shows how both static and dynamic frequency measurements can be obtained.

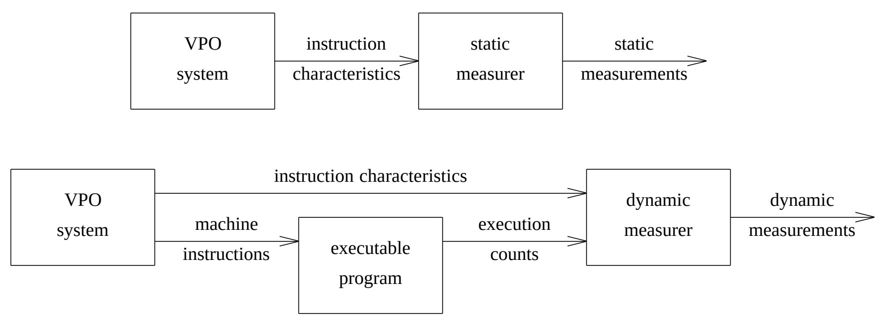

Figure 11: Producing Reports

Separating the collection and analysis of measurements has a number of advantages. Even if different evaluations of the data are required, the data is only collected once. If analysis of the execution of several different programs is needed, then the data can be collected from each program's execution separately. The analysis program can then accumulate the data collected from each of the executed programs and generate a single report. Finally, since the analysis of the data is separated from the generation of the data, the back end of the compiler requires less modification.

The generation of reports from the frequency measurements is also largely machine-independent. Most of the code from a 500 line C program that produces several different detailed reports has remained unchanged when ported to different machines. These reports include information about:

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

The cache simulator invoked during the program's execution contains no machine-dependent code. Information about a reference is passed through arguments each time the cache simulator is invoked. The characteristics of the simulated cache are determined by reading a configuration file at the beginning of the program's execution and a cache performance report is generated at the end of the execution of the program. To acquire a cache performance report with different cache characteristics simply requires modifying the configuration file and executing the program again.

Overhead of Collecting Measurements

Table I shows the execution overhead for collecting measurements for a set of programs on the Motorola 68020/68881.[4] The execution overheads of collecting frequency information and performing instruction, data, and unified cache evaluations are depicted. For the cache evaluations both the overhead for only generating the addresses for the trace and the total overhead including the cache simulation are given.

The instrumented code to collect frequency information runs on average 16% slower than uninstrumented code. This performance compares favorably with that of a different measurement system, called *bkgen*, which collects frequency measurements by inserting an instruction to increment a counter at the beginning of each basic block [HLT87]. Using *bkgen* on five different programs resulted in an average of 34% overhead. In addition to determining how often each instruction is executed, *ease* also collects information on the frequency that conditional branches are taken. The overhead using *ease* was less than

[4] All execution times reported in this paper were obtained by calculating the average of ten executions of each instance of a program. All cache simulations were run with a 1K byte direct-mapped cache with 16 byte lines. Periodic context switches were introduced at every 10,000 units of work where a hit required 1 unit and a miss required 10.

Program	Frequency Information	Cache Evaluation					
		Trace Generation Only			Cache Simulation Included		
		inst	data	unified	inst	data	unified
compact	1.28	11.22	13.40	50.45	55.82	63.41	195.57
cpp	1.28	8.73	13.01	49.50	53.33	64.11	196.27
diff	1.17	8.68	7.90	35.84	43.61	31.68	128.98
matmult	1.01	6.71	6.61	25.28	34.50	30.99	89.16
od	1.13	4.65	5.41	19.27	23.18	23.91	69.22
puzzle	1.30	19.62	8.00	60.55	89.46	36.92	192.24
queens	1.08	8.33	8.46	37.10	50.11	39.35	138.77
spline	1.03	1.62	1.56	3.71	4.05	3.31	10.42
average	1.16	8.70	8.04	35.21	44.26	36.71	127.58

Table I: Ratio to Normal Execution Time

half that required by *bkgen* since control-flow analysis was used to reduce the number of instructions that had to be inserted and executed to increment counters.

For calculating the overhead of only generating addresses for the trace, addresses were generated and passed to a dummy cache simulation routine. Instead of determining whether each reference was a hit or a miss, incrementing counters, and updating the state of the cache, the dummy routine simply returns immediately. It might seem that unified cache trace generation should require the overhead of the sum of separate trace generations for instruction and data caches. However, many more instruction references need to be generated to accurately intermix instruction and data references since entire basic blocks cannot be passed to the cache simulator.

As shown in Table I, most of the execution overhead for cache performance evaluation was due to time spent simulating the cache. Cache simulation using the traditional method of generating a trace to a file and then reading the trace file during cache simulation can require 1000 times the normal execution time [Smi82]. Performing instruction, data, and unified cache simulations during the execution of a program with *ease* results in the programs running only about an average of 44, 37, and 128 times slower, respectively. This overhead could be reduced if a simulator tuned for the particular cache configuration being simulated was used. Such a strategy requires relinking the program each time the cache configuration changes. As with trace generation only, the results with cache simulation included indicate that cache performance evaluation for a unified cache is more time-consuming than evaluating both instruction and data caches separately. The increased overhead is mainly due to the increased number of references processed by the cache simulator.

It is interesting to note the variation in overhead for the different programs. The execution overhead is affected by a number of factors including the average execution time for each instruction in the original program. For instance, the program incurring the lowest overhead was *spline*, which is floating-point intensive. Since most floating-point operations on the Motorola 68881 are much more time consuming than fixed-point operations on the Motorola 68020, the time spent simulating the cache represented a smaller percentage of the total execution time.

APPLICATIONS

The *ease* environment can be used to evaluate the impact of adding or deleting an architectural feature on a machine. First, a set of programs are compiled, executed, and measurements are collected. Next, the machine description within the compiler is modified to reflect the change in the architecture. Again, reports are obtained after compiling and executing the same set of programs. By comparing the two sets of measurements, the effect of the change can be evaluated. This approach was used in an experiment to determine the effect of varying both the number of user-allocable registers and the number of scratch versus nonscratch registers on a VAX-11 with a callee-save calling sequence [DaW91].[5] The method illustrated in Figure 3 was used when the number of available registers was increased. Table II shows the effect the different combinations had on the number of memory references. The results indicate that varying the number of user-allocable registers has little effect on the most effective percentage of scratch to nonscratch registers (about 40 per cent for the calling sequence used in Table II).

[5] With the default calling convention on the VAX-11, there are twelve allocable registers, six of which are scratch.

user-allocable registers	scratch registers	memory references
4	2 3	98,458,465 111,117,829
8	2 3 4	78,974,787 78,846,670 79,774,990
12	4 5 6	75,512,463 75,098,651 75,752,791
16	5 6 7	72,855,408 72,841,124 72,924,722

Table II: Results of Scratch/Nonscratch Combinations

ease can be used to evaluate the effect of adding new instructions to an architecture. For instance, the effect of using three different hardware support mechanisms to save and restore registers was investigated [DaW91]. Measurements were collected assuming that: 1) saves and restores are accomplished via the call instruction and a mask indicating the registers to save and restore, 2) special instructions are available to save and restore a specified set of registers on the run-time stack, and 3) saves and restores are done using primitive load and store instructions. The effect of these three mechanisms with a caller-save approach for a set of programs on the VAX-11 is shown in Table III. Note that the use of a mask with the `calls` instruction on the VAX-11 is available only with a callee-save convention. Likewise, the special instructions available to save and restore a set of registers, the `pushr` and `popr` instructions, also adjust the stack pointer. These instructions cannot be used in a caller-save approach since arguments may have already been pushed on the stack at the point of a call. Although measurements were obtained as if the desired instructions existed, the actual saves and restores were accomplished using individual `mov` instructions. The results indicate that while the use of a mask associated with a call instruction results in the fewest instructions being executed, it causes more memory references to be performed since for each call two additional memory references are required to save and restore the mask. The use of special instructions appears to be a good compromise.

measurement	mask	special inst	load/store
instructions	108,588,162	110,954,490	113,543,904
memory refs	84,976,490	79,071,798	79,071,798

Table III: Results of Different Save/Restore Mechanisms

The *ease* environment can also be used to assist in the design of a new architecture. A instruction set was designed to serve as a baseline for comparison in evaluating a new architectural feature [DaW90a]. The proposed baseline architecture was evaluated using the *ease* environment with the VAX-11 architecture serving as the host machine. Less than two weeks were required to generate a compiler, imitate the execution of a set of test programs, and collect measurements for the proposed architecture. To illustrate the detailed level of measurements that *ease* provides for a proposed architecture, various reports are shown in Tables IV—VII from measurements for the baseline architecture obtained by the program *compact* compacting its own source file. Not only can the total number of executed instructions be determined, but also the distribution of the different types of instructions that were executed. This information indicates whether the compiler makes use of the different instruction types. Likewise, a machine architect should be interested in whether the available addressing modes are used. The addressing mode and instruction type distributions for the program *compact* on the proposed baseline architecture are given in Tables IV and V, respectively.

addr mode	number executed	per cent
DISP	568,106	4.28
IMMED	2,715,704	20.48
INDEX	423,418	3.19
LABEL	818,555	6.17
REGDFR	776,319	5.85
REGISTER	7,959,571	60.02
total	13,261,673	100.00

Table IV: Addressing Mode Distribution for Compact

inst type	number executed	per cent
ADD	1,088,762	17.57
AND	314,040	5.07
CALL	28,009	0.45
CMP	735,542	11.87
DIV	5	0.00
JMP	93,071	1.50
JEQ	270,270	4.36
JGE	10,058	0.16
JGT	53,341	0.86
JLE	5,994	0.10
JLT	67,642	1.09
JNE	318,179	5.14
LOAD	1,226,515	19.80
MOV	174,836	2.82
MUL	3	0.00
NOP	199,300	3.22
OR	7,716	0.12
RESTORE	20,200	0.33
RET	28,004	0.45
SAVE	20,200	0.33
SETHI	246,193	3.97
SLA	504,748	8.15
SRA	51,808	0.84
STORE	513,319	8.29
SUB	208,788	3.37
XOR	8,760	0.14
total	6,195,303	100.00

Table V: Instruction Type Distribution for Compact

Some machines in the past have been designed with the goal of tightly encoding information in the instruction format. For instance, there are several instructions on the Motorola 68020 that do not allow specific operands to use certain addressing modes. If it was desired to design such a machine, then *ease* can provide this level of detail. Table VI shows, for the program *compact*, the different addressing modes and data type combinations used in the operand fields for load and store instructions.

It is useful to determine the cache performance for various cache configurations before the processor is actually implemented. Without requiring recompilation or relinking, cache performance results for different cache characteristics can be obtained in the *ease* environment by simply changing parameters in a configuration file and reexecuting the program. The hit ratios for the program *compact* with different

inst type	field	addressing mode	data type	number executed	per cent
LOAD	DST	REGISTER	LONG	1,226,515	100.00
	MEM	DISP	BYTE	6,481	0.53
			WORD	34,071	2.78
			LONG	364,105	29.69
		INDEX	LONG	391,826	31.95
		REGDFR	BYTE	10,054	0.82
			WORD	52,144	4.25
			LONG	367,834	29.99
STORE	MEM	DISP	BYTE	1	0.00
			WORD	10,061	1.96
			LONG	125,378	24.42
		INDEX	LONG	31,592	6.15
		REGDFR	BYTE	6,478	1.26
			WORD	75,992	14.80
			LONG	263,817	51.39
	SRC	REGISTER	LONG	513,319	100.00

Table VI: Load and Store Operand Information for Compact

instruction cache sizes and associativity are given in Table VII.

Besides evaluating proposed architectures or new architectural features, *ease* can be used for other purposes. The environment has been ported to ten different machines, including three RISCs to compare current architectures. Measurements were obtained and analyzed from executing nineteen programs on each of the architectures and an analysis was performed [Wha90]. Figure 12 shows the number of data

cache	associativity		
size	1	2	4
1K	92.10	92.13	92.24
2K	94.90	98.18	98.19
4K	96.11	98.19	98.81
8K	98.81	98.81	98.81

Table VII: Instruction Cache Hit Ratios for Compact memory references for each machine. The number of memory references due to referencing variables and spilling of temporaries is shown in solid lines. Dashed lines denote the additional number of memory references due to saving and restoring allocable registers. Dotted lines represent the additional number of memory references due to handling function linkage (stack pointer, frame pointer, program counter, etc.). Thus, *ease* calculates not only the total number of memory references, but also why each memory reference occurred. It is interesting to note that about 25% of the VAX-11 memory references are due to function linkage, implying that function calls on the VAX-11 are very expensive. On the other hand, the SPARC architecture, with its register windows, had very few memory references due to saving and restoring registers or function linkage (spilling and loading register windows).

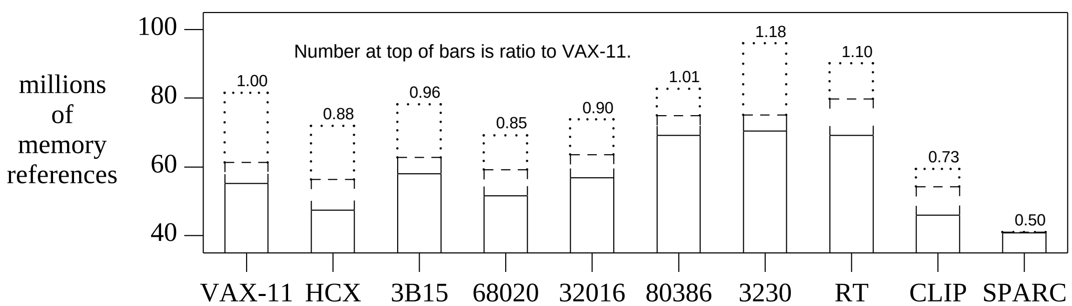

Figure 12: Number of Memory References

ease has also been used to analyze different code generation strategies. By recompiling the source files from the C run-time library, calling sequence conventions can be modified. By extracting measurements before and after each modification, the effect of a change can easily be analyzed. Six different methods for saving and restoring registers without interprocedural analysis or special hardware have been examined [DaW91]. Also, the benefits of passing arguments through registers as a calling sequence convention have been analyzed and the use of primitive call and return instructions has been compared with the use of their more complex counterpart instructions [DaW90b].

RELATED WORK

Traditional methods for obtaining architectural measurements for a proposed machine involve the use of a simulator [Bar81]. The *awb* (Architect's Workbench) environment uses an approach that allows the gathering of architectural measurements for different architectures from a single simulation of the execution of a program [FlW90]. Each high-level source program is translated to intermediate code and then separated into basic blocks. The intermediate-code operations are executed by an interpreter written in a high-level language. During the simulation, a trace routine is invoked at the beginning of each basic block that writes the basic block number being executed to a file. Characteristics of the instructions generated for each basic block with each architecture being studied are stored. Architectural analysis is based on the intermediate code basic block trace. Attributes of an architecture can easily be varied and the effect of each change can be determined. Unfortunately, using this approach results in no check for the validity of the translation since the target instructions for a proposed architecture are neither simulated nor translated to equivalent instructions and executed on a host machine. Even though several different architectures can be evaluated after obtaining the trace, the overhead of this approach is large. While actual times were not given, it was stated that the interpretive execution of each program, which excludes the overhead for producing the trace, required at least 100 times that of direct execution. A trace containing only basic block numbers can still be quite lengthy and require much disk space. While the interpretive execution of a program need only be performed once, the time required to read the basic block trace will also slow the analysis programs. Another problem with this approach is that it requires that a program always have the same basic block structure when translated for different architectures. When all optimizations are performed prior to code generation, such an assumption may be valid. Compiler optimizations may be more thoroughly applied to a program, however, when the characteristics of the target architecture are exposed to the optimizer [BeD88].

There has been much recent work on techniques that replace the use of simulators to collect architectural measurements. Several systems gather measurements by inserting instructions in the assembly code of a program produced by a compiler. Some packages instrument assembly programs by inserting instructions to increment counters to capture frequency information [Cme90, MMM90]. There have also been a variety of methods used to capture trace data efficiently for cache performance evaluations. The cache simulator can be linked directly with the executing program and trace information passed via function calls [StF89]. Alternatively, the cache simulator can be a separate task with the program being measured writing the trace data to trace buffers shared between the tasks [BKW90, EKK90].

The *ease* environment also captures frequency and trace information for similar measurements. One difficulty with retargeting some of the previously described systems is that they take assembly programs as input. While this allows measurements on assembly code to be captured even when the corresponding source code is not available, retargeting to another machine requires more effort since each assembly instruction needs to be parsed and analyzed. Program flow information may also need to be recalculated to avoid the situation in which the instructions inserted to obtain measurements change the value of live registers or condition codes. In most systems that capture frequency measurements, instructions are inserted at each basic block to increment counters. The *ease* environment attempts to minimize the incrementing of counters by using control and data-flow information available in *vpo*. Since the characteristics of the program are obtained during the compilation, some information not typically available from assembly code alone can also be measured.

The portion of *ease* that collects measurements for unified cache performance evaluation is similar to *trapeds* [StF89]. Both systems link the cache simulator with the program being traced and simulate the cache as the program executes. There are, however, some differences between *trapeds* and *ease*. *trapeds* is a separate program that takes assembly code as input. Therefore, *trapeds* would presumably be much more difficult to retarget since the syntax for assembly languages on various machines differ. The *trapeds* system also does not use data-flow analysis to minimize the number of registers to be saved and restored at each inserted call. Even so, faster trace generation and analysis times were reported for *trapeds*. The *trapeds* system required 30 times the execution overhead for trace generation only and 50 times the execution overhead when cache simulation was included. Since the techniques used were similar, the lower execution overheads reported with the *trapeds* system were probably due to the choice of three floating-point intensive programs to simulate, a much larger cache resulting in higher hit ratios,[6] and differences between the cache simulators.[7]

One measurement system that neither uses simulation nor parses assembly instructions is the *ae* environment [Lar90]. *ae* traces events using a technique called abstract execution. Similar to the *ease* environment being integrated into the *vpo* compiler system, the *ae* environment is integrated into the *gcc* compiler system.[8] Unlike the *vpo* compiler system, the structure of *gcc* complicates the task of storing information about a program. In both compiler systems programs are represented in an intermediate form called RTLs. Since an RTL in the *vpo* compiler system corresponds to a single assembly instruction, information about each instruction can easily be obtained by examining the semantic record representing the RTL. RTLs in a *gcc* compiler can often correspond to several machine instructions, making it more difficult to collect information about each instruction in the generated program and to insert instructions to collect measurements. The *ae* environment must monitor the code generation phase to count the number of instructions produced for each RTL. Details concerning the attributes of an instruction, such as the data type and addressing mode of each field, would also be more difficult to obtain. Both systems attempt to store information about each instruction when the assembly instructions are produced. *vpo* performs all optimizations, including instruction scheduling, before each RTL is translated to assembly code. Since there is not a one-to-one correspondence between an RTL and an assembly instruction in *gcc*, optimizations such as instruction scheduling must occur after assembly code generation, causing the RTLs and their corresponding assembly instructions to appear in different orders. The *ae* system must therefore recompute the relationship between the assembly code programs before and after instruction scheduling. Finally, since each RTL in *gcc* may be translated to more than one instruction, it would be difficult to extend the *ae* environment to imitate the execution and to collect measurements for a proposed architecture.

---

[6] There is less work required to simulate a cache hit than a cache miss.

[7] The cache simulator described in this paper allows the program to be executed again with a different cache configuration by simply changing a parameter in an input file. Some systems require relinking with a version of a cache simulator tuned for a particular cache configuration. Also, *ease* introduced periodic context switches in the simulations.

[8] *gcc* and *vpo* are descended from a common ancestor, *po* [DaF80].

The technique of program instrumentation has also been used recently to capture measurements for a proposed architecture. A system called *bkgen* [HLT87] produces a version of a program to be measured for a proposed machine that can be directly executed on an existing machine. *bkgen* requires either a direct mapping between the basic block structures of the assembly programs for the proposed and existing architectures or requires the construction of an assembly-to-assembly translator. A direct mapping between the basic block structures of the assembly programs for a proposed and existing architectures often does not occur when a compiler performs optimizations across basic blocks. The effort required to construct an assembly-to-assembly translator is nontrivial and can significantly delay the time required to develop a system for obtaining measurements. As shown in Figures 2 and 3, in the *ease* environment the translation of an RTL to assembly language of a different machine is quite simple.

CONCLUSIONS

The *ease* environment has been designed to minimize the effort involved in producing instructions for a proposed machine, imitating the execution of these instructions, and collecting measurements. The *vpo* system performs most optimizations in machine-independent code, which facilitates retargeting the compiler to a new machine. The same semantic record constructed while parsing an RTL to produce assembly code for an instruction on a proposed architecture can also be used to produce assembly code for an existing architecture and to store instruction information for the collection of measurements. Most of the code to perform the extraction of measurements is also machine-independent. The *vpo* compilers for ten different machines were modified to collect measurements [Wha90]. It typically took only three or four hours to make the machine-dependent modifications for the compiler on each machine. The *ease* environment collects measurements with little overhead as compared to more traditional techniques. *ease* also only required 15 to 20 percent overhead in compilation time.

The *ease* environment has been shown to be an efficient tool for architectural evaluation and design [DaW90a]. Since accurate and detailed reports can be produced for a variety of measurements, the impact of each modification can easily be determined. This allows one to use an iterative design method for evaluation of performance in a quantitative manner.

ACKNOWLEDGEMENTS

This work was supported in part by the U.S. National Science Foundation under grant CCR-8611653. Manuel Benitez implemented the machine-independent portion of the *vpo* compiler system and provided many useful suggestions for integrating *ease* with *vpo*. Anne Holler provided many helpful suggestions.

REFERENCES

[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.

[Bar81] M. R. Barbacci, ‘‘Instruction Set Processor Specifications (ISPS): The Notation and Its Application’’, *IEEE Transactions on Computers C-30*, 1 (January 1981), 24-40.

[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.

[BKW90] A. Borg, R. E. Kessler and D. W. Wall, ‘‘Generation and Analysis of Very Long Address Traces’’, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, Washington, May 1990, 270-279.

[Cme90] R. F. Cmelik, *Introduction to SpixTools*, Sun Microsystems Technical Memorandum, July 1990.

[DaF80] J. W. Davidson and C. W. Fraser, ‘‘The Design and Application of a Retargetable Peephole Optimizer’’, *ACM Transactions on Programming Languages and Systems 2*, 2 (April 1980), 191-202.

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

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

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

[DaW90a] 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.

[DaW90b] J. Davidson and D. Whalley, ‘‘Fast Function Calls and Returns’’, Tech. Rep. 90-20, University of Virginia, August 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.

[EKK90] S. J. Eggers, D. R. Keppel, E. J. Koldinger and H. M. Levy, ‘‘Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor’’, *Proceedings SIGMETRICS ’90 Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990, 37-47.

[FlW90] M. J. Flynn and R. I. Winner, ‘‘ASIC Microprocessors’’, *Proceedings 23’rd Annual Microprogramming Workshop*, 1990, 237-243.

[HJB82] J. Hennessy, N. Jouppi, F. Baskett, T. Gross and J. Gill, ‘‘Hardware/Software Tradeoffs for Increased Performance’’, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 2-11.

[HeP90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.

[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.

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

[Lar90] J. R. Larus, ‘‘Abstract Execution: A Technique for Efficiently Tracing Programs’’, *Software—Practice & Experience 20*, 12 (December 1990), 1241-1258.

[MMM90] *UMIPS-V Reference Manual (pixie and pixstats)*, MIPS Computer Systems, Sunnyvale, CA, 1990.

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

[Smi82] A. J. Smith, ‘‘Cache Memories’’, *Computing Surveys 14*, 3 (September 1982), 473-530.

[StF89] C. Stunkel and W. Fuchs, ‘‘TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation’’, *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, May 1989, 70-78.

[Wha90] D. B. Whalley, *Ease: An Environment for Architecture Study and Experimentation*, PhD Dissertation, University of Virginia, Charlottesville, VA, 1990.

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