Automatic Derivation of Compiler Machine Descriptions

CHRISTIAN S. COLLBERG
University of Arizona

We describe a method designed to significantly reduce the effort required to retarget a compiler to a new architecture, while at the same time producing fast and effective compilers. The basic idea is to use the native C compiler at compiler construction time to discover architectural features of the new architecture. From this information a formal machine description is produced. Given this machine description, a native code-generator can be generated by a back-end generator such as BEG or `burg`. A prototype automatic Architecture Discovery Tool (called ADT) has been implemented. This tool is completely automatic and requires minimal input from the user. Given the Internet address of the target machine and the command-lines by which the native C compiler, assembler, and linker are invoked, ADT will generate a BEG machine specification containing the register set, addressing modes, instruction set, and instruction timings for the architecture. The current version of ADT is general enough to produce machine descriptions for the integer instruction sets of common RISC and CISC architectures such as the Sun SPARC, Digital Alpha, MIPS, DEC VAX, and Intel x86.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Portability*; D.3.2 [**Programming Languages**]: Language Classifications—*Macro and assembly languages*; D.3.4 [**Programming Languages**]: Processors—*Translator writing systems and compiler generators*

General Terms: Languages

Additional Key Words and Phrases: Back-end generators, compiler configuration scripts, retargeting

1. INTRODUCTION

An important aspect of a compiler implementation is its *retargetability*. For example, a new programming language whose compiler can be quickly retargeted to a new hardware platform or a new operating system is more likely to gain widespread acceptance than a language whose compiler requires extensive retargeting effort.

In this paper we will briefly review the problems associated with two popular approaches to building retargetable compilers, *C Code Code Generation*, and *Specification-Driven Code Generation*. We will then propose a

This paper is a revised and extended version of two previous papers, Collberg [1997a, 1997b].
Author's address: Department of Computer Science, University of Arizona, Tucson, AZ 85721; email: collberg@cs.arizona.edu.

new method, *Self-Retargeting Code Generation*, which attempts to overcome these problems.

The basic idea behind Self-Retargeting Code Generation is very simple. Consider a scenario where we have just been delivered a new machine $\mathcal{M}$, with a CPU whose ISA is unknown to us. In order to retarget the compiler for our favorite language to $\mathcal{M}$, we need to find out about $\mathcal{M}$'s instruction set, register set, addressing modes, instruction timings, procedure calling conventions, etc. We could attempt to read the manuals that were shipped with $\mathcal{M}$, but it might be easier if, somehow, we could coax this information directly out of $\mathcal{M}$ and the systems software (native code compilers and assemblers) that were shipped with it. In fact, we can treat $\mathcal{M}$ (and its system software) as an *oracle* that has complete self-knowledge, and we can think of clever ways of querying $\mathcal{M}$ about itself, convincing it to give up this information. For example:

(1) We could ask any native code compiler about the instruction set, register set, and addressing modes of the ISA, as well as the standard calling conventions of the operating system.
(2) We could ask any assembler about the instruction encoding of the machine, and, if it is an optimizing assembler, the instruction scheduling rules of the architecture.
(3) We could ask $\mathcal{M}$ itself, finally, about the cost of each instruction, since $\mathcal{M}$ is an expert on executing instructions for its own ISA.

The Architecture Discovery Tool, or ADT, the tool we will be describing in this paper, implements the Self-Retargeting Code Generation idea. Given the Internet address of a machine, ADT will query it and its native C compiler and assembler, extracting information about the ISA, instruction costs, procedure calling conventions, assembler syntax, etc.. From this information the ADT will synthesize a formal machine description from which the BEG back-end generator can produce a new code generator.

Before we continue to discuss the design of ADT, we will briefly discuss the merits of C Code Code Generation and Specification-Driven Code Generation.

1.1 C Code Code Generation

The back-end of a C Code Code Generation-based compiler generates C code which is compiled by the native C compiler. If care has been taken to produce portable C code, then targeting a new architecture requires no further action from the compiler writer. Furthermore, any improvement to the native C compiler's code generation and optimization phases will automatically benefit the compiler. A number of compilers have achieved portability through C Code Code Generation. Examples include early versions of the SRC Modula-3 compiler [Digital Systems Research Center 1996] and the ISE Eiffel compiler [Interactive Software Engineering 1996].

Unfortunately, experience has shown that generating truly portable C code is much more difficult than it might seem. Not only is it necessary to handle architecture and operating system specific differences such as word size and

alignment, but also the idiosyncrasies of the C compilers themselves. Machine-generated C code will often exercise the C compiler more than code written by humans, and is therefore more likely to expose hidden problems in the code generator and optimizer. Other potential problems are the speed of compilation[1] and the fact that the C compiler's optimizer (having been targeted at code produced by humans) may be ill equipped to optimize the code emitted by our compiler.

Further complications arise if there is a large semantic gap between the source language and C. For example, if there is no clean mapping from the source language's types to C's types, the generated C program program will be very difficult to debug.

C Code Code Generation-based compilers for languages supporting garbage collection face even more difficult problems. Many collection algorithms assume that there will always be a pointer to the beginning of every dynamically allocated object, a requirement which is violated by some optimizing C compilers. Under certain circumstances, this will result in live objects being collected.

Other compelling arguments against the use of C as an intermediate language can be found in Chase and Ridoux [1990]. C−− [Ramsey and Jones 2000] is a recent attempt to design a C-like intermediate language that overcomes these difficulties.

1.2 Specification-Driven Code Generation

The back-end of a Specification-Driven Code Generation compiler generates intermediate code which is transformed to machine code by a specification-driven code generator. The main disadvantage is that retargeting becomes a much more arduous process, since a new specification has to be written for each new architecture. A `gcc` [Stallman 1995] machine specification, for example, can be several thousand lines long. Popular back-end generators such as BEG [Emmelmann et al. 1989] and `burg` [Fraser et al. 1992] require detailed descriptions of the architecture's register set and register classes, as well as a set of pattern-matching rules that provide a mapping between the intermediate code and the instruction set.

Writing *correct* machine specifications can be a difficult task in itself. This can be seen by browsing through `gcc`'s machine descriptions. The programmers writing these specifications experienced several different kinds of problems:

Documentation/Software Errors/Omissions. The most serious and common problems seem to stem from documentation being out of sync with the actual hardware/software implementation:

(1) "`... the manual says that the opcodes are named movsx ..., but the assembler ... does not accept that.''` [i386]

(2) "`WARNING! There is a small i860 hardware limitation (bug?) which we may run up against ... we must avoid`

[1]In some C Code Code Generation-based compilers the most expensive part of compilation is compiling the generated C code.

`using an 'addu' instruction to perform such comparisons because ... This fact is documented in a footnote on page 7-10 of the ... Manual.`" [i860]

Lack of Understanding of the Architecture. Even with access to manuals, some specification writers seemed uncertain of exactly which constructs were legal:

(3) "`Is this number right?`" [mips]

(4) "`Can this ever happen on i386?`" [i386]

(5) "`Will divxu always work here?`" [i386]

Hardware/Software Updates. Often, updates to the hardware or systems software are not immediately reflected by updates in the machine specification:

(6) "`This has not been updated since version 1. It is certainly wrong.`" [ns32k]

Lack of Time. Sometimes the programmer knew what needed to be done, but simply did not have the time to implement the changes:

(7) "`This INSV pattern is wrong. It should ... Fixing this is more work than we care to do for the moment, because it means most of the above patterns would need to be rewritten ....`" [Hitachi H8/300]

Note that none of these comments are `gcc` specific. Rather, they express universal problems of writing and maintaining a formal machine specification, regardless of which machine-description language and back-end generator is being targeted.

1.3 Self-Retargeting Code Generation

In this paper we will propose an approach to the design of retargetable compilers which combines the advantages of the two methods outlined above, while avoiding most of their drawbacks. The basic idea is to use the native C compiler to discover architectural features of the new target machine, and then to use that information to automatically produce a specification suitable for input to a back-end generator. We will refer to this method as *Self-Retargeting Code Generation*.

More specifically, our system generates a number of small C programs[2] which are compiled to assembly-code by the native C compiler. We will refer to these codes collectively as *samples*, and individually as *C code samples* and *assembly-code samples*.

The assembly-code samples are analyzed to extract information regarding the instruction set, the register set and register classes, the procedure calling convention, available addressing modes, and the sizes and alignment constraints of available data types.

[2]Obviously, other widely available languages such as FORTRAN will do equally well.

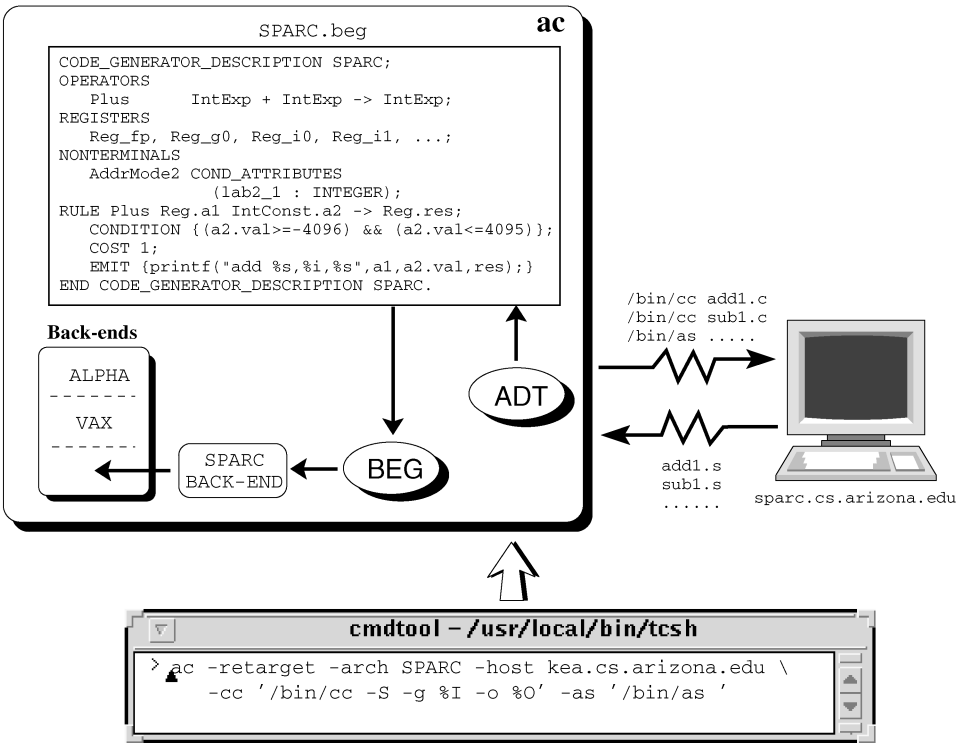

Fig. 1. The structure of a self-retargeting compiler ac for some language *L*. The back-end generator BEG and ADT are integrated into ac. In this example, the user asks ac to retarget itself to the SPARC.

The primary application of the ADT is to aid and speed up manual retargeting. Although a complete analysis of a new architecture can take a long time (several hours, depending on the speed of the host and target systems and the link between them), it is still 1-2 orders of magnitude faster than manual retargeting.

However, with the advent of Self-Retargeting Code Generation it will also become possible to build *self-retargeting compilers*, that is, compilers that can automatically adapt themselves to produce native code for any architecture. Figure 1 shows the structure of such a compiler ac for some language *L*. Originally designed to produce code for the Alpha and VAX, ac is able to retarget itself to the SPARC architecture. The user only needs to supply the Internet address of a SPARC machine and the command lines by which the C compiler, assembler, and linker are invoked on this machine. ADT, which is integrated into the compiler, then proceeds to compile a large number of small C samples on the SPARC, analyzes the resulting assembly code, and based on the information it has collected, constructs a machine specification from which BEG (also included in ac) generates a new back-end.

The architecture discovery package will have other potential uses as well. For example, machine-independent tools for editing of executables (EEL)

[Larus and Schnarr 1995], ATOM [Srivastava and Eustace 1994], dynamic compilation (DCG) [Engler and Proebsting 1994], and decompilation [Cifuentes and Gough 1995] all need access to architectural descriptions, and their retargeting would be simplified by automatic architecture discovery.

2. SYSTEM OVERVIEW AND REQUIREMENTS

For a system like this to be truly useful it must make few requirements—of its users as well as of the target machines. The prototype implementation has been designed to be as automatic as possible, to require as little user input as possible, and to require the target system to provide as few and simple tools as possible:

(1) We require a user to provide the Internet address of the target machine and the command-lines by which the C compiler, assembler, and linker are invoked. For a wide range of machines, all other information is deduced by the system itself, without further user interaction.
(2) We require the target machine to provide a C compiler that produces assembly code, an assembler which flags illegal assembly instructions,[3] a linker, and a remote execution facility such as `rsh`. The C compiler is used to provide assembly-code samples for us to analyze; the assembler is used to deduce the syntax of the assembly language; and the remote execution facility is used for communication between the development and target machines.

If these requirements have been fulfilled, the ADT will produce a BEG machine description completely autonomously.

The ADT consists of five major components (see Figure 2). The *Generator* generates `C` code programs and compiles them to assembly code on the target machine. The *Lexer* extracts and tokenizes relevant instructions (i.e., corresponding to the `C` statements in the sample) from the assembly code. The *preprocessor* builds a data-flow graph from each sample. The *Extractor* uses this graph to extract the semantics of individual instructions and addressing modes. The *synthesizer*, finally, gathers the collected information together and produces a machine description, in our case for the BEG back-end generator.

3. THE GENERATOR AND LEXER

The Generator produces a large number of simple C code samples. Samples may contain arithmetic and logical operations like

```
⌜main(){int b=5,c=6,a=b+c;}⌝,
```

conditionals like

```
⌜main(){int b=5,c=6,a=7; if(b<c)a=8;}⌝,
```

[3]The manner in which errors are reported is unimportant; assemblers which simply crash on the first error are quite acceptable for our purposes.

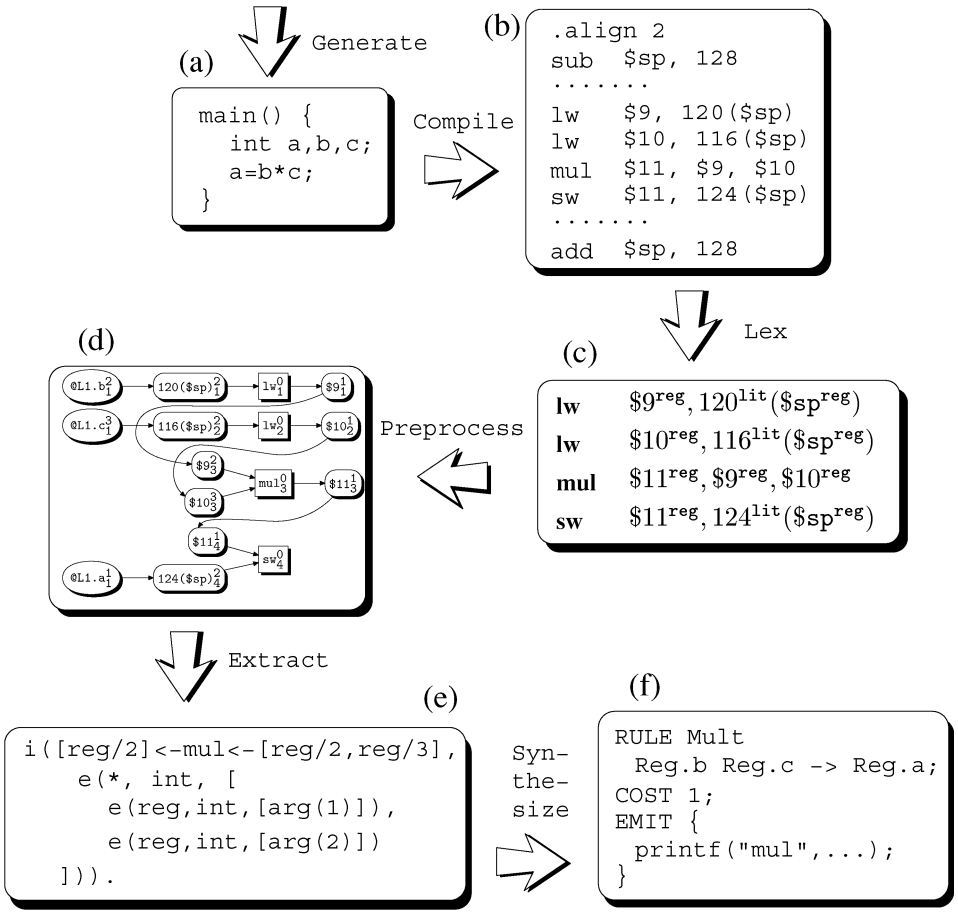

Fig. 2. An overview of the major components of the architecture discovery system. The Generator produces a large number of small C programs (a) and compiles them to assembly on the target machine. The Lexer analyzes the raw assembly code (b) and extracts and tokenizes the instructions that are relevant to our further analyses (c). The Preprocessor deduces the signature of all instructions, and builds a data-flow graph (d) from each sample. The semantics of individual instructions (e) are deduced from this graph, and from this information, finally, a complete BEG specification (f) is built.

and procedure calls like

$$\ulcorner \texttt{main()\{int b=5,a; a=P(b);\}} \urcorner .$$

We would prefer to generate a "minimal" set of samples, the smallest set such that the resulting assembly-code samples would be easy to analyze and would contain all the instructions produced by the compiler. Unfortunately, we cannot know whether a particular sample will produce interesting code combinations for a particular machine until we have tried to analyze it. We must therefore

```
main(){
  int a,b,c;
  a = b+c;
}
```

```
        .align      4
        .file       2 "x.c"
        .globl      main
        .loc        2 1
#    1  main()int a,b,c;a=b+c;
        .ent        main 2
main:
        .option     O1
        ldgp        $gp, 0($27)
        lda         $sp, -32($sp)
        .frame      $sp, 32, $26, 0
        .prologue   1
        ldl         $1, 16($sp)
        ldl         $2, 8($sp)
        addl        $1, $2, $3
        stl         $3, 24($sp)
        .loc        2 1
        bis         $31, $31, $0
        .livereg 0xFC7F0002,0x3FC00000
        lda         $sp, 32($sp)
        ret         $31, ($26), 1
        .end        main
```

Fig. 3. A C code sample and the resulting assembly code generated by the native Alpha compiler. The four relevant instructions of the sample have been shaded.

produce as many simple samples as possible. For example, for subtraction we generate

```
a=b-c,   a=a-b,   a=b-a,
a=a-a,   a=b-b,   a=7-b,
a=b-7,   a=7-a,   a=a-7.
```

This means that we will be left with a large number of samples, typically around 150 for each numeric type supported by the hardware. The samples are created by simply instantiating a small number of templates parameterized on type (int, float, etc.) and operation (+, −, etc.).

3.1 Extracting Relevant Instructions

The samples are compiled to assembly code by the native C compiler and the Lexer extracts the instructions relevant to our analysis. This is nontrivial, since the relevant instructions often only make up a small fraction of the ones produced by the C compiler.

The Alpha C compiler, for example, will compile ⌜`main(){int b,c,a=b+c;}`⌝ into the twenty-two nonblank lines shown in Figure 3. Eleven of these lines are directives, one line is a label, one is a comment, and nine are instructions. Only four instructions are relevant to us. Furthermore, compiling with a moderate level of optimization (`-O2`) makes the compiler remove the code generated from `a=b+c`, since it can determine that `a`'s value will never be used. Similarly, in the sample ⌜`main(){int b=1,c=2,a=b+c;}`⌝ (where initializations have been included in order to be able to execute the sample), the Alpha C compiler will replace `a=b+c` by `a=3` even with optimization turned off.

(a)

```
/* init.h */
extern int z1,z2,z3,
           z4,z5,z6;
extern void Init();
```

(b)

```
/* init.c */
int z1,z2,z3,
    z4,z5,z6;
void Init(n,o,p)
int *n,*o,*p;
{
 z1=z2=z3=1;
 z4=z5=z6=1;
 *n=-1;
 *o=313; *p=109;
}
```

(c)

```
/* add.c */
#include "init.h"
main () {
   int a, b, c;
   Init(&a, &b, &c);
   if (z1) goto Begin;
   if (z2) goto End;
   if (z3) goto Begin;
   if (z4) goto End;
   if (z5) goto Begin;
   if (z6) goto End;
Begin:
   a = b + c;
End:
   printf("%i\n", a);
   exit(0);
}
```

(d)

```
      tstl   z1
      jeql   L1
      jbr    L2
L1:   tstl   z2
      jeql   L3
      jbr    L4
L3:   tstl   z3
      jeql   L5
      jbr    L2
L5:   tstl   z4
      jeql   L6
      jbr    L4
L6:   tstl   z5
      jeql   L7
      jbr    L2
L7:   tstl   z6
      jeql   L2
      jbr    L4
L2:   addl3  -12(fp),-8(fp),-4(fp)
L4:
```

Fig. 4. A C code sample and the resulting assembly-code sample for the VAX. The relevant instruction (`addl3`) can be easily found since it is delimited by labels `L2` and `L4`, corresponding to `Begin` and `End`, respectively, and the assembly code contains several jumps to these labels.

Fortunately, it is possible to design the C code samples to make it easy to extract the relevant instructions and to minimize the compiler's opportunities for optimizations that could complicate our analyses. In Figure 4 a separately compiled procedure `Init` initializes the variables `a`, `b`, and `c`, but hides the initialization values from the compiler to prevent it from performing constant propagation. The `main` routine contains three conditional jumps to two labels `Begin` and `End`, immediately preceding and following the statement `a=b+c`. The compiler will not be able to optimize these jumps away since they depend on variables hidden within `Init`. Two assembly-code labels corresponding to `Begin` and `End` will effectively delimit the instructions of interest. These labels will be easy to identify since they each must be referenced at least three times.

The `printf` statement ensures that a *dead code elimination* optimization will not remove the assignment to `a`.

3.2 Tokenizing the Input

Before we can start parsing the assembly code samples, we must try to discover as much as possible about the syntax accepted by the assembler. Fortunately, most modern assembly languages seem to be variants of a "standard" notation:

—there is at most one instruction per line;
—each instruction consists of an optional label, an operator, and a list of comma-separated arguments;
—integer literals are prefixed by their base; and
—comments extend from a special comment-character to the end of the line.

We use two fully automated techniques for discovering the details of a particular assembler:

(1) we can textually scan the assembly code produced by the C compiler, or
(2) we can draw conclusions based on whether a particular assembly program is accepted or rejected by the assembler.

We call these methods *Textual Analysis* and *Assembler Error Analysis*, respectively.

For example, one of the first things we need to discover is what integer literal syntax the assembler accepts:

(1) Which bases are accepted?
(2) Which prefixes do the different bases use?
(3) Are upper-, lower-, and/or mixed-case hexadecimal literals accepted?

To extract this information, we use a simple textual analysis. We start by compiling the program ⌜`main(){int a=1235;}`⌝ and then scan the resulting assembly code for the constant 1235, in all the common bases. If we find the string `"0x4d3,"` we can conclude that lower-case hexadecimal constants with the prefix `"0x"` are accepted by the assembler.

We use assembler error analysis to discover the comment-characters accepted by the assembler. We start out with the assembly code produced from ⌜`main(){ }`⌝ and add a line consisting of a suspected comment character (such as `#`) followed by other, obviously erroneous characters, yielding a line such as ⌜`#&$^*%*^)(&`⌝. We submit this to the assembler for acceptance or rejection. If the assembler does not produce any errors, we can conclude that `#` is indeed a comment character. We repeat this process with all combinations of one or two special characters. This way, we find out that the Alpha assembler accepts comments of the form

(1) ⌜␣#⌝ until end of line,
(2) ⌜/*⌝ until ⌜*/⌝, and
(3) ⌜//⌝ until end of line.

```
reg --> ['%', f, p].
reg --> ['%', s, p].
reg --> ['%', g], digit(0,0).
reg --> ['%', i], digit(0,5).
reg --> ['%', l], digit(0,7).
reg --> ['%', o], digit(0,7).
```

(a)

```
reg --> ['%'], letter, letter.
reg --> ['%'], letter, digit(0,9).
```

(b)

```
am(1,[reg(R)]) --> reg(R).
am(2,[int(I)]) --> int(I).
am(3,[lab(L)]) --> lab(L).
am(4,[reg(R1),reg(R2)]) --> ['('],
    reg(R1), [','], reg(R2), [')'].
am(5,[int(I),reg(R)]) -->
    int(I), ['('], reg(R), [')'].
am(6,[int(I1),reg(R),int(I2)]) -->
    int(I1), ['(',','], reg(R),
    [','], int(I2), [')'].
```

(c)

Fig. 5. Prolog definite clause grammars (DCGs) describing the integer register set of the SPARC ((a) and (b)) and the addressing modes of the x86 (c). The grammars can be used directly for either parsing or generation. `digit(L,H)` parses digits between `L` and `H`. `int(_)`, `reg(_)`, and `lab(_)` parse integers, registers, and labels, respectively, and return the parsed token.

3.3 Finding the Register Set and Addressing Modes

To find the register set supported by the architecture, we start with a simple textual analysis of the samples, using the heuristic that any operand or part of an operand that is not a label or a literal, and that does not contain a bracket, is a potential register. From these strings we build a grammar[4] that generates the strings (Figure 5(a)).

For architectures with large register sets, it is likely that not all registers will be present in the samples. We therefore construct a generalized grammar (Figure 5(b)) that is used to generate a stream of potential registers which are tested using assembler error analysis.

Classification of addressing modes is straightforward once we have gained a good understanding of the register set. A grammar (Figure 5(c)) is constructed that is able to parse an arbitrary operand, and return its kind (a number) and its constituent parts. The grammar in Figure 5(c), for example, classifies the string `"0(,%edx,8)"` as being an addressing mode of type 6 whose constituent parts are `[int(0),reg(%edx),int(8)]`. Obviously, at this point this is merely a syntactic classification. Later we will discover the exact mathematical function computed by the addressing mode.

We also use assembler error analysis to discover the accepted ranges of integer immediate operands. On the SPARC, for example, we would detect that the `add` instruction's immediate operand is restricted to [-4096,4095].

Finally, assembler error analysis is used to construct register classes. A register class is defined to be any subset of registers that can occur in at least one argument position of at least one instruction or addressing mode. For example, if an instruction set has two instructions ⌜`foo` r⌝ and ⌜`bar` s⌝, where r is one of the registers `R2` or `R5` and s is one of `R3` and `R5`, then we will construct two register classes {`R2`, `R5`} and {`R3`, `R5`}. Note that this is a different classification than what most ISA descriptions would give. Most manuals will classify two

[4]Algorithms developed in the field "Computational Learning Theory" can be used for this purpose. See Brāzma [1995].

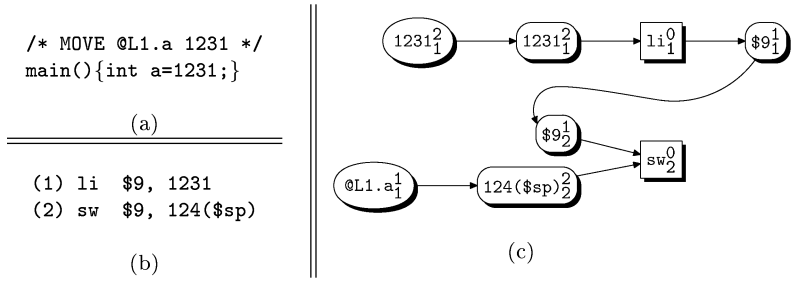

Fig. 6. A simple assignment sample, and the corresponding MIPS assembly code and data-flow graph.

registers as belonging to the same class if they "look the same," even if they are not completely interchangeable.

Some assembly languages can be quite exotic. The Tera [Tera Computer Company 1995], for example, uses a variant of Scheme as its assembly language. In such cases our automated techniques will not be sufficient, and we require the user to provide a translator into a more standard notation.

4. THE PREPROCESSOR

The samples produced by the lexical phase may contain irregularities that will make them difficult to analyze directly. Some problems may be due to the idiosyncrasies of the architecture, some due to the code generation and optimization algorithms used by the C compiler. It is the task of the preprocessor to identify any problems and convert each sample into a standard form (a *data-flow graph*) which can serve as the basis for further analysis. Referring back to Figure 2, the preprocessor takes a set of parsed assembly-code samples as input (Figure 2(c)) and produces a set of data-flow graphs (Figure 2(d)) as output.

A data-flow graph makes explicit the exact flow of information between individual instructions in a sample. For example, the preprocessor would convert the MIPS assembly code sample in Figure 6(b) into the data-flow graph in Figure 6(c). It is obviously much easier to extract information from the data-flow graph than directly from the assembly code.

In order to be able to build the data-flow graph, we must, for every instruction of every sample, know where the instruction takes its arguments and where it deposits its result(s). There are several major sources of confusion, some of which are illustrated in Figure 7.

For example, an operand that does not appear explicitly in the assembly code, but is hardwired into the instruction itself, is called an *implicit argument*. They occur frequently on older architectures (on the x86, `cltd` (Figure 12) takes its input argument and delivers its result in register `%eax`), as well as on more recent ones when procedure call arguments are passed in registers (Figure 7(a)). If we cannot identify implicit arguments, we obviously cannot accurately describe the flow of information in the samples.

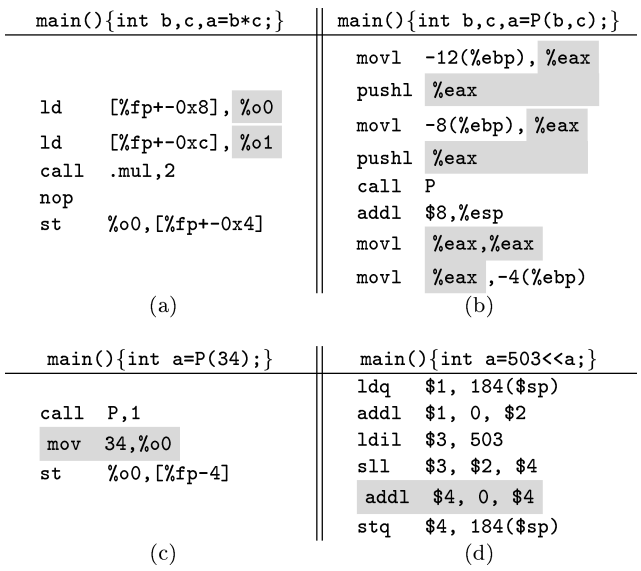

Fig. 7. Examples of compiler- and architecture-induced irregularities that the preprocessor must deal with. On the SPARC, procedure actuals are passed in registers `%o0`, `%o1`, etc. Hence these are implicit input arguments to the `call` instruction in (a). In (b), the x86 C compiler is using register `%eax` for three independent tasks: to push `b`, to push `c`, and to extract the result of the function call. The SPARC `mov` instruction in (c) is in the `call` instruction's delay slot, and is hence executed *before* the call. In (d), finally, the Alpha C compiler generated a redundant instruction ⌜`addl $4, 0, $4`⌝.

As shown in Figure 7(b), a sample may contain several distinct uses of the same register. Again, we need to be able to detect such *register reuse* or the flow of information within the sample cannot be identified.

4.1 Mutation Analysis

Static analysis of individual samples is not sufficient to accurately detect and repair irregularities such as the ones shown in Figure 7. Instead we use a novel dynamic technique called *Mutation Analysis* which compares the execution result of an original sample with one that has been slightly changed. Based on the result of this comparison, we can draw conclusions about the structure of the assembly code.

Figure 8 shows the overall structure of a mutation analysis. The particular mutation illustrated in Figure 8 tries to determine if there is a data dependence between the first two instructions of the sample. The mutation simply swaps the two load instructions, executes the mutated sample, and compares the result to the result of executing the original sample. In this case the results are identical, which allows us to conclude that the two load instructions are indeed data independent.

Figure 9 lists the mutations available to us. We can delete, move and copy instructions, rename registers, and insert "clobbering" instructions that overwrite a register with some random value.

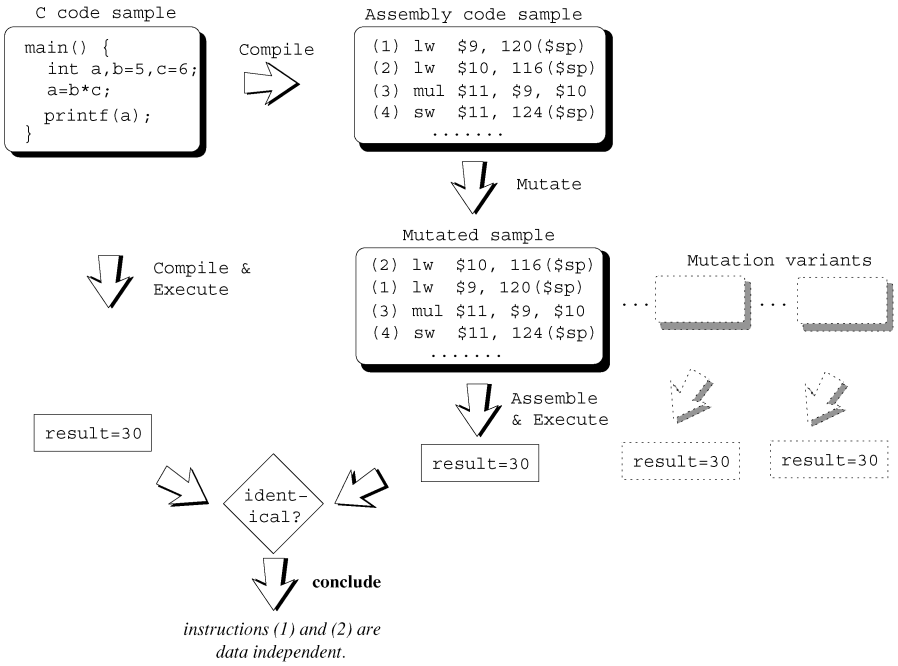

Fig. 8. Mutation Analysis algorithm. The original C code sample is compiled and executed, and its result is recorded. The same sample is also compiled to assembly code. This sample is mutated, assembled, linked, and executed, and its output is compared to that of the original sample. In most cases several variant mutations are generated, all of which have to produce the same value as the original sample in order for the mutation to succeed.

	(2) OP$_2$ A_2^1, A_2^2, A_2^3 (1) OP$_1$ A_1^1, R1, A_1^3 (3) OP$_3$ R1, A_3^2, A_3^3 **move((1),after,(2))**	(1) OP$_1$ A_1^1, R1, A_1^3 (2) OP$_2$ A_2^1, A_2^2, A_2^3 MOV -13,R1 (3) OP$_3$ R1, A_3^2, A_3^3 **clobber(R1,after,(2))**
(1) OP$_1$ A_1^1, R1, A_1^3 (2) OP$_2$ A_2^1, A_2^2, A_2^3 (3) OP$_3$ R1, A_3^2, A_3^3 Original sample	(1) OP$_1$ A_1^1, R1, A_1^3 (2) OP$_2$ A_2^1, A_2^2, A_2^3 (3) OP$_3$ R1, A_3^2, A_3^3 (1') OP$_1$ A_1^1, R1, A_1^3 **copy((1),after,(3))**	(1) OP$_1$ A_1^1, R1, A_1^3 (3) OP$_3$ R1, A_3^2, A_3^3 **delete((2))**
	(1) OP$_1$ A_1^1, R1, A_1^3 (2) OP$_2$ A_2^1, A_2^2, A_2^3 (3) OP$_3$ R2, A_3^2, A_3^3 **rename(R1,R2,(3))**	(1) OP$_1$ A_1^1, R3, A_1^3 (2) OP$_2$ A_2^1, A_2^2, A_2^3 (3) OP$_3$ R3, A_3^2, A_3^3 **renameAll(R1, R3)**

Fig. 9. This table lists the available mutations: we can **move**, **copy**, and **delete** instructions, and we can **rename** and **clobber** (overwrite) registers. The A_i^js are operands not affected by the mutations.

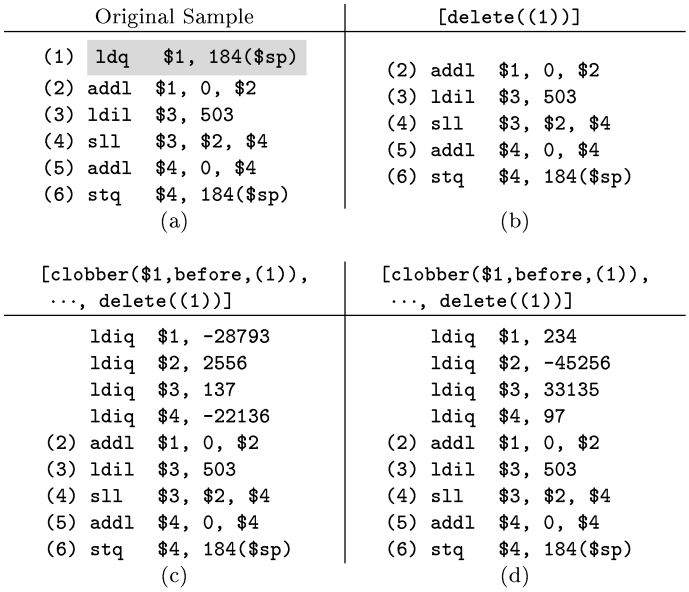

Fig. 10. Removing redundant instructions. In this example we are interested in whether instruction (1) is redundant.

To avoid a mutation succeeding (producing the same value as the original sample) by chance, we always try several variants of the same mutation. A mutation is successful only if all variants succeed. Two variants may, for example, differ in the values used to clobber a register, or in the new register name chosen for a `rename` mutation.

4.2 Eliminating Redundant Instructions

To illustrate this idea we will consider a trivial, but extremely useful, analysis, *redundant instruction elimination*. An instruction is removed from a sample and the modified sample is assembled, linked, and executed on the target machine. If the mutated sample produces the same result as the original one, the instruction is removed permanently. This process is repeated for every instruction of every sample. This mutation will yield samples which are smaller and simpler than the original, and hence easier to analyze.

Figure 10 illustrates this. In Figure 10(b) we delete instruction (1); assemble, link, and execute the sample; and compare the result with that produced from the original sample in (a). If they are the same, we conclude that instruction (1) is redundant and can be removed permanently.

Even for a trivial mutation like this, we must take special care for the mutation not to succeed by chance. For example, if register `$1` contains the same value as `184($sp)` then the sample will produce the correct value regardless of whether instruction (1) is present or not. To counter this possibility, we must clobber all registers with random values. To make sure that the clobbers themselves do not initialize a register to a correct value, two variant mutations

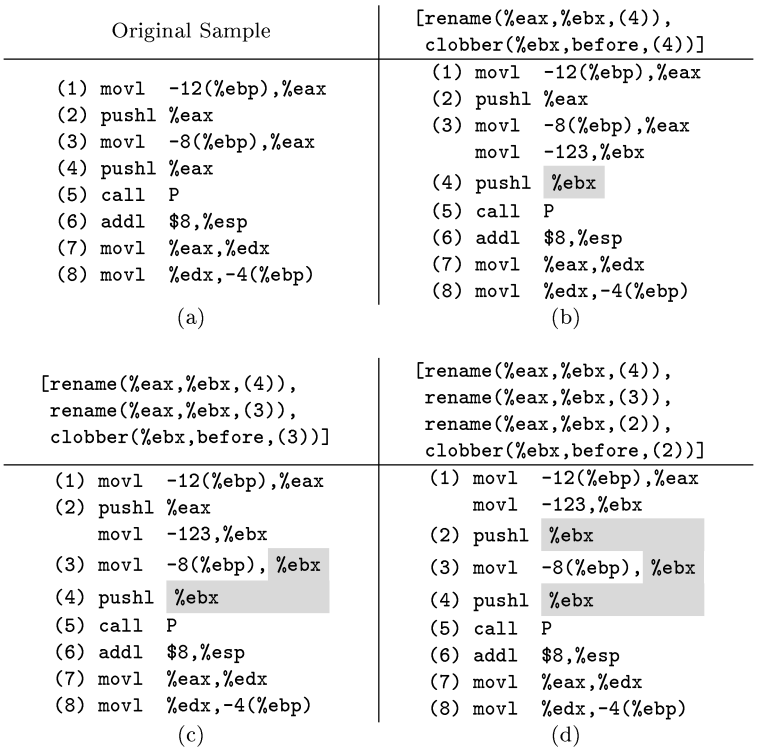

Fig. 11. Splitting the sample from Figure 7(b). (a) shows the sample after the references to %eax in (7) and (8) have been processed. Mutation (b) will fail (i.e., produce a value different from the original sample), since the region only contains the use of %eax, not its definition. The region is extended until the mutated sample produces the same result as the original (c), and then again until the results differ (d).

(Figure 10(c) and (d)) are constructed using different clobbering values. Both variants must succeed for the mutation to succeed.

The result of these mutations is a new set of simplified samples, where redundant instructions (such as ⌜move R1, R1⌝, ⌜nop⌝, ⌜add R1, 0, R1⌝) have been eliminated. These samples will be easier for the algorithms in Section 5 to analyze. They will also be less confusing to further mutation analyses.

To further illustrate the use of mutation analysis, we will next consider three more profound preprocessing tasks in detail, namely Live-Range Splitting, Implicit Argument Detection, and Register Definition/Use Computation.

4.3 Splitting Register Live-Ranges

Some samples (such as Figure 7(b)) will contain several unrelated references to the same register. To allow further analysis, we need to split such register references into distinct *regions*. Figure 11 shows how we can use the *rename* and *clobber* mutations to construct regions that contain the smallest set of registers that can be renamed without changing the semantics of the sample. Regions are grown backwards, starting with the last use of a register, and continuing

Original Sample

```
(1) movl  -8(%ebp),%ecx
(2) movl  %ecx,%eax
(3) cltd
(4) idivl -12(%ebp)
(5) movl  %eax,-4(%ebp)
```

(a)

```
[rename(%ecx,%ebx,(1)),
 rename(%ecx,%ebx,(2))]
```

```
(1) movl  -8(%ebp), %ebx
(2) movl  %ebx ,%eax
(3) cltd
(4) idivl -12(%ebp)
(5) movl  %eax,-4(%ebp)
```

(b)

```
[rename(%eax,%ebx,(2)),
 rename(%eax,%ebx,(5))]
```

```
(1) movl  -8(%ebp),%ecx
(2) movl  %ecx, %ebx
(3) cltd
(4) idivl -12(%ebp)
(5) movl  %ebx ,-4(%ebp)
```

(c)

```
[move((4),after,(5))]
```

```
(1) movl  -8(%ebp),%ecx
(2) movl  %ecx,%eax
(3) cltd
(5) movl  %eax,-4(%ebp)
(4) idivl -12(%ebp)
```

(d)

Fig. 12. Detecting implicit arguments. (a) is the original x86 sample. The (b) mutation will succeed, indicating that `cltd`, `idivl`, and `movl` are independent of `%ecx`. The (c) and (d) mutations will both fail, since `%eax` is an implicit argument to `idivl`.

until the region also contains the corresponding definition of that register. To make the test completely reliable, the new register is clobbered just prior to the proposed region, and each mutated sample is run several times with different clobbering values.

4.4 Detecting Implicit Arguments

Detecting implicit arguments is complicated by the fact that some instructions have a variable number of implicit input arguments (cf. `call` in Figure 7(a,c)), some have a variable number of implicit output arguments (the x86's `idivl` returns the quotient in `%eax` and the remainder in `%edx`), and some take implicit arguments that are both input and output.

The only information we get from running a mutated sample is whether it produces the same result as the original one. Therefore all our mutations must be "correctness preserving," in the sense that unless there is something special about the sample (such as the presence of an implicit argument), the mutation should not affect the result.

So, for example, it should be legal to move an instruction I_2 before an instruction I_1 as long as they (and any intermediate instructions) do not have any registers in common.[5] Therefore the mutation in Figure 12(c) should succeed, which it does not, since `%eax` is an implicit argument to `idivl`.

The algorithm runs in two steps. We first attempt to prove that, for each operator O and each register R, O is *independent* of R, that is, R is not an implicit argument of O. We do this by renaming R, which, if there are no implicit arguments, should not affect the result computed by the sample

[5]Note that while this statement does not hold for arbitrary codes (where, for example, aliasing may be present), it does hold for our simple samples.

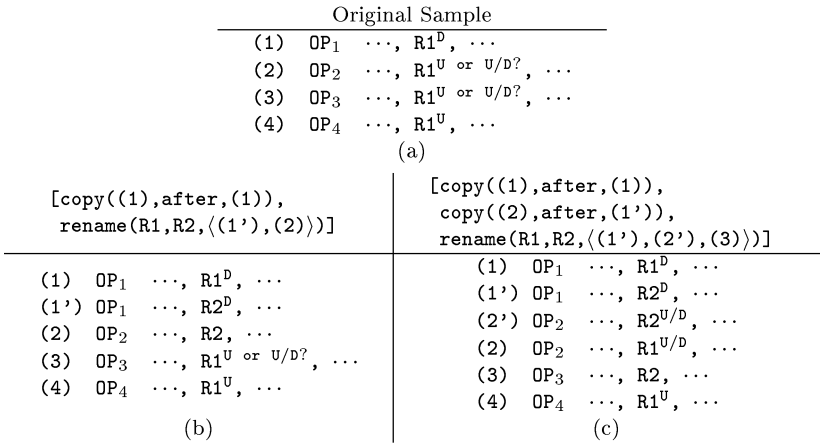

Fig. 13. Computing definition/use information. Necessary register clobbers have been omitted for clarity. The first occurrence of R1 in (a) is a definition (D), the last one a use (U). The references to R1 in (2) and (3) could be either uses, or use-definitions (U/D). The mutation in (b) will succeed iff (2) is a pure use. In (c) we assume that (b) failed, and hence (2) is a use-definition. (c) will succeed iff (3) is a pure use.

(see Figure 12 (b) and (c)). For those operator/register-pairs that fail the first step, we use **move** mutations to show that the registers are actually implicit arguments (Figure 12(d)).

4.5 Computing Definition/Use

When implicit register arguments have been detected and made explicit and distinct register uses have been split into individual live ranges, we are ready to attempt our final preprocessing task. Each register reference in a live range has to be analyzed and we have to determine which references are *pure uses*, *pure definitions*, and *use-definitions*. Instructions that both use and define a register are common on CISC machines,[6] but less common on modern RISC machines.

The first occurrence of a register in a live-range must be a definition; the last one must be a use. The intermediate occurrences can either be pure uses or use-definitions. For example, in the following x86 multiplication sample, the first reference to register %edx (%edx$_1$) is a pure definition, the second reference (%edx$_2$) a use-definition, and the last one (%edx$_3$) a pure use:

```
main () {
  int a,b,c;
  a = b * c;
}
```

```
movl -8(%ebp),%edx1
imull -12(%ebp),%edx2
movl %edx3,-4(%ebp)
```

Figure 13 shows how we use the `copy` and `rename` mutations to create a separate path from the first definition of a register R1 to a reference of R1 that is

[6]For example, on the VAX ⌜`addl2 5,r1`⌝ increments register one by 5.

either a use or a use-definition. If the reference is a use-definition, the mutation will fail, since the new value computed will not be passed on to the next instruction that uses `R1`.

4.6 Building the Data-Flow Graph

The last task of the preprocessor is to combine the gathered information into a data-flow graph for each sample. This graph will form the basis for all further analysis.

A data-flow graph describes the flow of information between elements of a sample. Information that was implicit or missing in the original sample but recovered by the preprocessor is made explicit in the graph. The nodes of the graph are

(1) operators in the assembly code sample,
(2) operands in the assembly code sample, and
(3) *semantic arguments*, that is, variables that occurred in the original C sample.

Figure 14 shows several examples of data-flow graphs. All the graph drawings in this paper were generated automatically (using `METAPOST` [Hobby 1992]) as part of the documentation produced by the ADT. Operator nodes are drawn as rectangular boxes, operand nodes as round-edged boxes, and semantic arguments as ovals. Nodes that represent implicit arguments are drawn unshaded.

For ease of identification, each node in a graph is labeled $\texttt{N}_{\texttt{i}}^{\texttt{a}}$, where `N` is the operator or operand, `i` is the instruction number, and `a` is `N`'s argument number in the instruction. In Figure 14(b), for example, $\texttt{\$11}_{3}^{1}$ refers to the use of register `$11` as the first argument to the third instruction (`mul`). Semantic arguments are represented by *data descriptors* [Holt 1987]. For example, in Figure 14(b), `@L1.a` refers to the variable `a` at static nesting level 1.

Informally, there is an edge $A \rightarrow B$ if B uses a value stored in or computed by A. More formally, and referring to Figure 14(b), there is an edge $A \rightarrow B$

(1) if A is an operator that stores a value into a register B (edge e_2),
(2) if B is an operator that reads a value or address stored in A (edges e_1, e_4, e_5),
(3) if $A = \texttt{R}_{\texttt{i}}^{\texttt{j}}$ and $B = \texttt{R}_{\texttt{k>i}}^{\texttt{l}}$ are the same register `R`, such that $\texttt{Op}_{\texttt{i}}^{\texttt{0}}$ stores a value into `R` that is subsequently used by $\texttt{Op}_{\texttt{k}}^{\texttt{0}}$ (edge e_3), or
(4) if A is a semantic argument and B is the node in which the computation of A's l- or r-value begins (edges e_6, e_7).

Given the information provided by the various mutation analyses, building the graph for a sample is straightforward. A node is created for every operator and operand that occurs explicitly in the sample. Instructions that were determined to be redundant (Section 4.2) are ignored. Extra nodes are created for all implicit input and output arguments (Section 4.4). Finally, based on the information gathered through live-range splitting (Section 4.3) and definition-use analysis (Section 4.5), output register nodes can be connected to the corresponding input nodes.

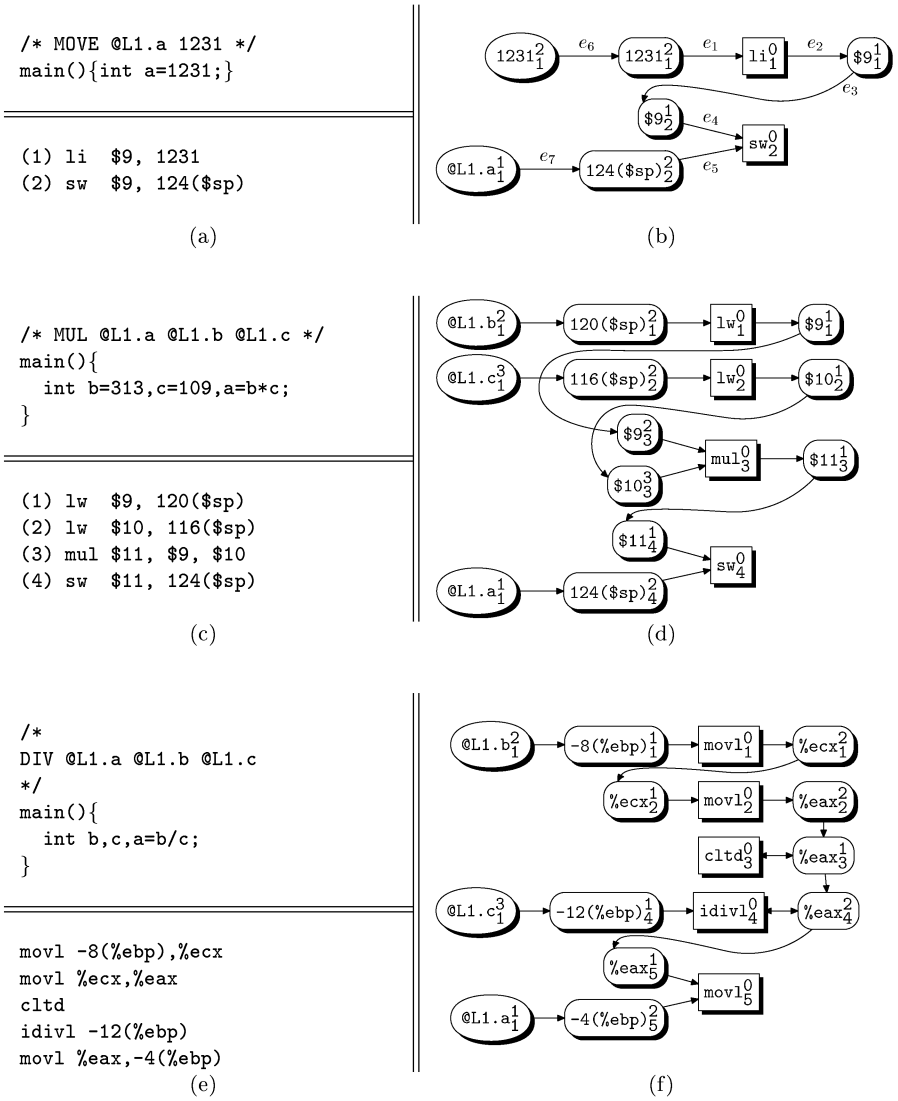

Fig. 14. (a) and (b) MIPS assignment, (c) and (d) MIPS multiplication, and (e) and (f) x86 division samples and their corresponding data-flow graphs.

Note that once all samples have been converted into data-flow graphs, we can easily determine the signatures of individual instructions. This is a first and crucial step toward a real understanding of the machine. From the graph in Figure 14(f), for example, we can conclude that `cltd` is a register-to-register instruction, *and* that the input and output registers both have to be %eax.

5. THE EXTRACTOR

The purpose of the extractor is to analyze the data-flow graphs generated by the preprocessor and to extract the function computed by each individual operator

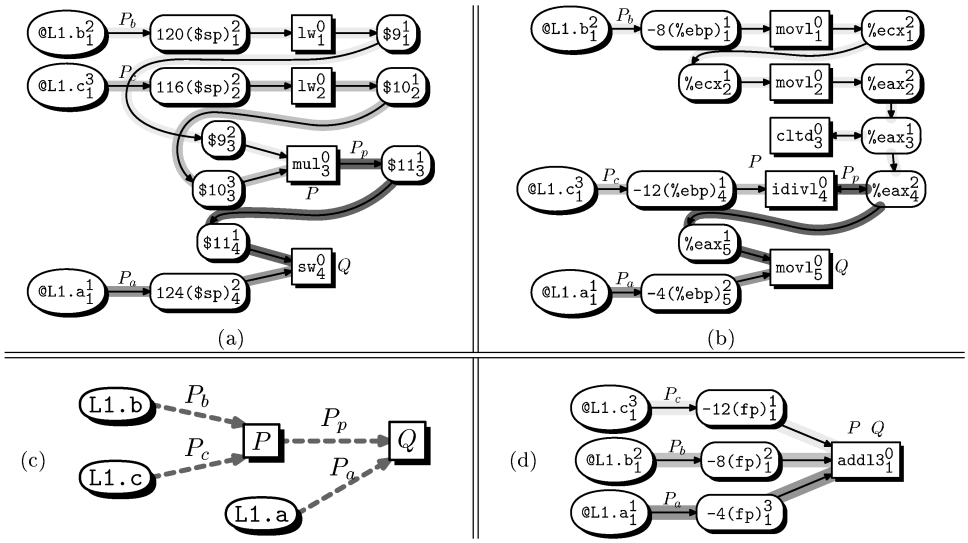

Fig. 15. Data-flow graphs after graph matching. (a) is MIPS multiplication, (b) is x86 division, and (d) is VAX addition. (c) is the pattern-matching template.

and operand. Referring back to Figure 2, the extractor takes a set of data-flow graphs as input (Figure 2(d)) and produces a set of instruction semantics (Figure 2(e)) as output.

In this section we will describe two of the many techniques that can be employed: *Graph Matching*, which is a simple and fast approach that works well in many cases, and *Reverse Interpretation*, which is a more general (and much slower) method.

5.1 Graph Matching

To extract the information of interest from the data-flow graphs, we need to make the following observation: for a binary arithmetic sample $\mathtt{a} = \mathtt{b} \oplus \mathtt{c}$, the graph will have the general structure shown in Figure 15(c). That is, the graph will have paths P_b and P_c originating in `@L1.b` and `@L1.c` and intersecting at some node P. Furthermore, there will be paths P_p and P_a originating in P and `@L1.a` that intersect at some node Q. P_p may be empty, while all other paths will be nonempty.

P marks the point in the graph where $\oplus$ is performed. The paths P_b and P_c represent the code that loads the r-values of `b` and `c`, respectively. Similarly, P_a represents the code that loads `a`'s l-value. Q, being the point where the paths computing $\mathtt{b} \oplus \mathtt{c}$ and `a`'s l-value meet, marks the point where the value computed from $\mathtt{b} \oplus \mathtt{c}$ is stored in `a`.

Hence, we can draw the following conclusions from the samples in Figure 15:

(1) On the MIPS, $P = \mathtt{mul}_3^0$, $Q = \mathtt{sw}_4^0$. Hence, `lw` is responsible for loading the r-values of `b` and `c`, `mul` performs multiplication, and `sw` stores the result.

(2) On the x86, `b`'s r-value is loaded by the operator sequence $[\mathtt{movl}_1^0, \mathtt{movl}_2^0, \mathtt{cltd}_1^0]$ contained in P_b. The division is performed by $P = \mathtt{idivl}_4^0$, which

also loads c's r-value (since P_c does not contain any operators). $Q = \texttt{movl}_5^0$, finally, stores the computed value.

(3) On the VAX, P_a, P_b, and P_c contain no operators, and $P = Q = \texttt{addl3}_1^0$. Hence, addl3 loads the r-values of b and c and stores their sum in a.

5.2 Reverse Interpretation

Graph Matching is fast and simple but it fails to analyze some graphs. Particularly problematic are samples that perform multiplication by a constant, since these are often expanded to sequences of shifts and adds. Figure 16 shows such a case. Note that node $\texttt{109}_1^3$, which represents the literal multiplicand, is completely disconnected from the rest of the graph. As a result, matching the graph against the template in Figure 15(c) will fail.

The method we will describe next, on the other hand, is completely general and can handle this and other kinds of nonstandard graphs, but suffers from a worst-case exponential time complexity.

In the following, we will take an interpreter I to be a function

$$I :: \texttt{Sem} \times \texttt{Prog} \times \texttt{Env}_{\texttt{in}} \longrightarrow \texttt{Env}_{\texttt{out}}.$$

Sem is a mapping from instructions to their semantic interpretation, Env a mapping from memory locations, registers, etc., to values, and Prog is the sequence of instructions to be executed. The result of the interpretation is a new environment, with (possibly) new values in memory and registers. The example in Figure 17(a) adds 5 to the value in memory location 10 (M[10]) and stores the result in memory location 20.

A *reverse interpreter* R, on the other hand, is a function that, given a program and an initial and final environment, will return a semantic interpretation that turns the initial environment into the final environment. R has the signature

$$R :: \texttt{Sem}_{\texttt{in}} \times \texttt{Env}_{\texttt{in}} \times \texttt{Prog} \times \texttt{Env}_{\texttt{out}} \longrightarrow \texttt{Sem}_{\texttt{out}}.$$

In other words, R extends $\texttt{Sem}_{\texttt{in}}$ with new semantic interpretations, such that the program Prog transforms $\texttt{Env}_{\texttt{in}}$ to $\texttt{Env}_{\texttt{out}}$. In the example in Figure 17(b), the reverse interpreter determines that the add instruction performs addition.

5.2.1 *The Algorithm.* We will devote the remainder of this section to a detailed discussion of reverse interpretation. Particularly, we will show how a probabilistic search strategy (based on expressing the *likelihood* of an instruction having a particular semantics) can be used to implement an effective reverse interpreter.

The idea is simply to interpret each sample, choosing (nondeterministically) new interpretations of the operators and operands until every sample produces the required result. The reverse interpreter will start out with an empty semantic mapping ($\texttt{Sem}_{\texttt{in}} = \{\}$), and, on completion, will return a $\texttt{Sem}_{\texttt{out}}$ mapping each operator and addressing mode to a semantic interpretation.

The reverse interpreter has a small number of semantic primitives (arithmetic, comparisons, logical operations, loads, stores, etc.) to choose from. RISC-type instructions will map more or less directly into these primitives, but

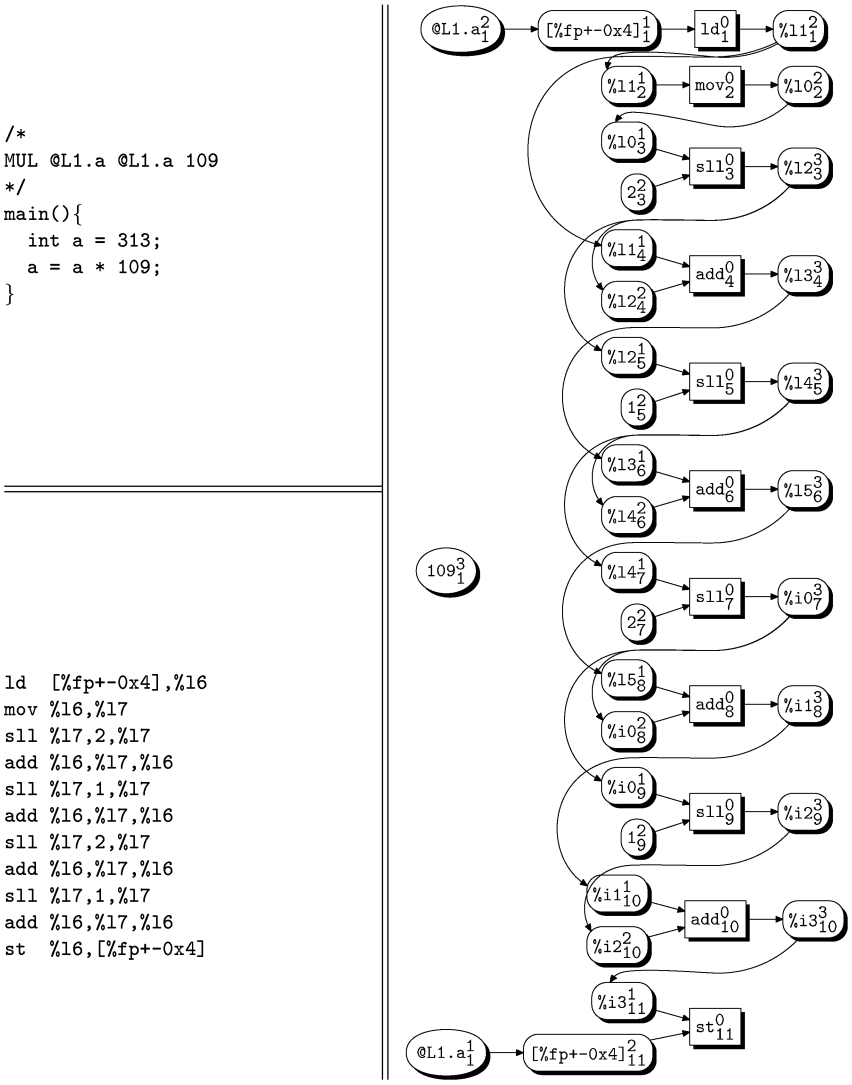

Fig. 16. SPARC multiplication sample. Note how the register live-ranges have been split to simplify building the data-flow graph.

they can be combined to model arbitrarily complex machine instructions. For example,

$$\texttt{addl3}_{\epsilon \leftarrow \texttt{a,a,a}}(a, b, c) = \mathit{store}\,(a,\, \mathit{add}\,(\mathit{load}\,(b),\, \mathit{load}\,(c)))^{7}$$

[7]To distinguish between instructions with the same mnemonic but different semantics (such as ⌜`addl $4,%ecx`⌝ and ⌜`addl -8(%ebp),%edx`⌝ on the x86), instructions are indexed by their signatures. In these signatures, `a` represents an address, `r` a register, `c` a literal, and ϵ a null argument. $\texttt{lw}_{\texttt{r} \leftarrow \texttt{a}}$, for example, is an instruction that takes an address as argument and returns a result in a register.

$$\text{(a)}\quad I\left(\begin{array}{l}\{\texttt{add}(x,y)=x+y;\texttt{load}(a)=\texttt{M}[a];\texttt{store}(a,b)=\texttt{M}[a]\leftarrow b\},\\ [\![\texttt{store}(20,\texttt{add}(\texttt{load}(10),5))]\!],\\ \{\texttt{M}[10]=7,\texttt{M}[20]=9\}\end{array}\right)\longrightarrow\begin{array}{l}\{\texttt{M}[10]=7,\\ \texttt{M}[20]=12\}\end{array}$$

$$\text{(b)}\quad R\left(\begin{array}{l}\{\texttt{load}(a)=\texttt{M}[a];\texttt{store}(a,b)=\texttt{M}[a]\leftarrow b\},\\ \{\texttt{M}[10]=7,\texttt{M}[20]=9\},\\ [\![\texttt{store}(20,\texttt{add}(\texttt{load}(10),5))]\!],\\ \{\texttt{M}[10]=7,\texttt{M}[20]=12\}\end{array}\right)\longrightarrow\begin{array}{l}\{\texttt{add}(x,y)=x+y;\\ \texttt{load}(a)=\texttt{M}[a];\\ \texttt{store}(a,b)=\texttt{M}[a]\leftarrow b\}\end{array}$$

Fig. 17. (a) shows the result of an interpreter I evaluating a program $[\![\texttt{store}(20, \texttt{add}(\texttt{load}(10), 5))]\!]$, given an environment $\{\texttt{M}[10] = 7, \texttt{M}[20] = 9\}$. The result is a new environment in which memory location 20 has been updated. (b) shows the result of a reverse interpretation. R is given the same program and the same initial and resulting environments as I. R also knows the semantics of the `load` and `store` instructions. Based on this information, R will determine that in order to turn $\texttt{Env}_{\texttt{in}}$ into $\texttt{Env}_{\texttt{out}}$, the `add` instruction should have the semantics $\texttt{add}(x, y) = x + y$.

$$R\left(\begin{array}{l}\{\texttt{am}^1_{\texttt{a}\leftarrow\texttt{r,c}}(a,b)=\textit{loadAddr}(\textit{add}(a,b)),\\ \texttt{lw}_{\texttt{r}\leftarrow\texttt{a}}(a)=\textit{load}(a),\\ \texttt{sw}_{\epsilon\leftarrow\texttt{r,a}}(a,b)=\textit{store}(a,b)\},\\ \{\texttt{M}[\texttt{@L1.b}]=313,\texttt{M}[\texttt{@L1.c}]=109\},\\ [\![\begin{array}{llll}(1)&a_1&\leftarrow\ \texttt{am}^1_{\texttt{a}\leftarrow\texttt{r,c}}&\leftarrow\ (120,\$\texttt{sp})\\ (2)&\$\texttt{9}&\leftarrow\ \texttt{lw}_{\texttt{r}\leftarrow\texttt{a}}&\leftarrow\ (a_1)\\ (3)&a_2&\leftarrow\ \texttt{am}^1_{\texttt{a}\leftarrow\texttt{r,c}}&\leftarrow\ (116,\$\texttt{sp})\\ (4)&\$\texttt{10}&\leftarrow\ \texttt{lw}_{\texttt{r}\leftarrow\texttt{a}}&\leftarrow\ (a_2)\\ (5)&\$\texttt{11}&\leftarrow\ \texttt{mul}_{\texttt{r}\leftarrow\texttt{r,r}}&\leftarrow\ (\$\texttt{9},\$\texttt{10})\\ (6)&a_3&\leftarrow\ \texttt{am}^1_{\texttt{a}\leftarrow\texttt{r,c}}&\leftarrow\ (124,\$\texttt{sp})\\ (7)&&\texttt{sw}_{\epsilon\leftarrow\texttt{r,a}}&\leftarrow\ (\$\texttt{11},a_3)\end{array}]\!],\\ \{\texttt{M}[\texttt{@L1.b}]=313,\texttt{M}[\texttt{@L1.c}]=109,\texttt{M}[\texttt{@L1.a}]=34117\}\end{array}\right)\longrightarrow\begin{array}{l}\{\texttt{am}^1_{\texttt{a}\leftarrow\texttt{r,c}}(a,b)=\textit{loadAddr}(\textit{add}(a,b)),\\ \texttt{lw}_{\texttt{r}\leftarrow\texttt{a}}(a)=\textit{load}(a),\\ \texttt{sw}_{\epsilon\leftarrow\texttt{r,a}}(a,b)=\textit{store}(a,b),\\ \texttt{mul}_{\texttt{r}\leftarrow\texttt{r,r}}(a,b)=\textit{mul}(a,b)\}\end{array}$$

Fig. 18. Reverse interpretation example. The data-flow graph in Figure 14(d) has been broken up into seven primitive instructions, each one of the form `result` ← `operator` ← (`arguments`). $\texttt{am}^1_{\texttt{a}\leftarrow\texttt{r,c}}$ represents the "register + offset" addressing mode. The a_i's are "pseudo-registers" which hold the result of address calculations.

models the VAX add instruction, and

$$\texttt{madd}_{\texttt{r}\leftarrow\texttt{r,r,r}}(a,b,c)=\textit{add}\,(a,\textit{mul}\,(b,c))$$

the MIPS multiply-and-add. Figure 19 lists the most important primitives, and in Section 5.2.3 we will discuss the choice of primitives in detail.

The example in Figure 18 shows the reverse interpretation of the sample in Figure 14 (c) and (d). The data-flow graph has been converted into a list of instructions to be interpreted. In this example, we have already determined the semantics of the `sw` and `lw` instructions and the $\texttt{am}^1_{\texttt{a}\leftarrow\texttt{r,c}}$ register + offset addressing mode. All that is left to do is to fix the semantics of the `mul` instruction such that the resulting environment contains $\texttt{M}[\texttt{@L1.a}] = 34117$. The reverse interpreter does this by enumerating all possible semantic interpretations of `mul`, until one is found that produces the correct $\texttt{Env}_{\texttt{out}}$.

Before we can arrive at an effective algorithm, there are a number of issues that need to be resolved.

First of all, it should be clear there will be an infinite number of valid semantic interpretations of each instruction. In the example in Figure 18, $\texttt{mul}_{\texttt{r}\leftarrow\texttt{r,r}}$ could get any one of the semantics $\texttt{mul}_{\texttt{r}\leftarrow\texttt{r,r}}(a,b) = a*b$, $\texttt{mul}_{\texttt{r}\leftarrow\texttt{r,r}}(a,b) = a*b*1$, $\texttt{mul}_{\texttt{r}\leftarrow\texttt{r,r}}(a,b) = b^2*a/b$, etc. Since most machine instructions have very simple semantics, we should strive for the simplest (shortest) interpretations.

Second, there may be situations where a set of samples will allow several conflicting interpretations. To see this, let $S = \ulcorner\texttt{main()\{int b=2,c=1,a=b*c;\}}\urcorner$ be a sample, and let the multiplication instruction generated from S be named `mul`. Given that $\texttt{Env}_{\texttt{in}} = \{\texttt{b} = 2, \texttt{c} = 1\}$ and $\texttt{Env}_{\texttt{out}} = \{\texttt{b} = 2, \texttt{c} = 1, \texttt{a} = 2\}$, the reverse interpreter could reasonably conclude that $\texttt{mul}(a,b) = a/b$, or even $\texttt{mul}(a,b) = a - b + 1$. A wiser choice of initialization values (such as `b=34117,c=109`) would avoid this problem. A Monte Carlo algorithm can help us choose wise initialization values: generate pairs of random numbers (a,b) until a pair is found for which none of the interpreter primitives (or simple combinations of the primitives) yield the same result.

Third, the reverse interpreter might produce the wrong result if its arithmetic is different from that of the target architecture. We use `enquire` [Pemberton 1991] to gather information about word sizes on the target machine, and simulate arithmetic in the correct precision.

A further complication is how to handle addressing mode calculations such as $a_1 \leftarrow \texttt{am}^1_{\texttt{a}\leftarrow\texttt{r,c}} \leftarrow (120, \$\texttt{sp})$ which are used in calculating variable addresses. These typically rely on stack or frame pointer registers which are initialized outside the sample. How is it possible for the interpreter to determine that in Figure 18 `@L1.a`, `@L1.b`, and `@L1.c` are addressed as $124 + \$\texttt{sp}$, $120 + \$\texttt{sp}$, $116 + \$\texttt{sp}$, respectively, for some unknown value of $\$\texttt{sp}$? We handle this by initializing every register not initialized by the sample itself to a unique value ($\$\texttt{sp} \leftarrow \perp_{\$\texttt{sp}}$). The interpreter can easily determine that a symbolic value $124 + \perp_{\$\texttt{sp}}$ must correspond to the address `@L1.a` after having analyzed a couple of samples such as $\ulcorner\texttt{main()\{int a=1231;\}}\urcorner$ in Figure 14(b).

However, the most difficult problem of all is how the reverse interpreter can avoid combinatorial explosion. We will address this issue next.

5.2.2 *Guiding the Interpreter.* Reverse interpretation is essentially an exhaustive search for a workable semantics of the instruction set. Or, to put it differently, we want the reverse interpreter to consider *all* possible semantic interpretations of every operator and addressing mode encountered in the samples, and then choose an interpretation that allows all samples to evaluate to their expected results. As noted before, there will always be an infinite number of such interpretations, and we want the interpreter to favor the simpler ones.

Any number of heuristic search methods can be used to implement the reverse interpreter. There is, however, one complication. Many search algorithms require a *fitness function* which evaluates the goodness of the current search position, based on the results of the search so far. This information is used to guide the direction of the continued search. Unfortunately, no such fitness

function can exist in our domain. To see this, let us again consider the example interpretation in Figure 18. The interpreter might guess that

$$\texttt{mul}_{\texttt{r}\leftarrow\texttt{r,r}}(a,b) = \mathit{mul}\,(a,\ \mathit{add}\,(100,b)),$$

and, since $313 * 100 + 109 = 31409$ is close to the real solution (34117) the fitness function would give this solution a high goodness value. Based on this information, the interpreter may continue along the same track, perhaps trying

$$\texttt{mul}_{\texttt{r}\leftarrow\texttt{r,r}}(a,b) = \mathit{mul}\,(a,\ \mathit{add}\,(110,b)).$$

This is clearly the wrong strategy. In fact, an unsuccessful interpretation (one that fails to produce the correct $\texttt{Env}_{\texttt{out}}$) gives us no new information to help guide our further search.

Fortunately, we can still do much better than a completely blind search. The current implementation is based on a *probabilistic best-first search*. The idea is to assign a *likelihood* (or *priority*) to each possible semantic interpretation of every operator and addressing mode. The interpreter will consider more likely interpretations (those that have higher priority) before less likely ones. Note the difference between likelihoods and fitness functions: the former are static priorities that can be computed before the search starts, the latter are evaluated dynamically as the search proceeds.

Let I be an instruction, S the set of samples in which I occurs, and R a possible semantic interpretation of I. Then the likelihood that I will have the interpretation R is

$$\begin{aligned} \mathit{likelihood}(S,I,R) \;=\; & c_1 \cdot \mathit{match}(S,I,R) + c_2 \cdot \mathit{sem}(S,R) \\ & + c_3 \cdot \mathit{sig}(I,R) + c_4 \cdot \mathit{name}(I,R) \end{aligned}$$

where the c_i's are implementation specific weights and *match*, *sem*, *sig*, and *name* are functions defined below.

—*match*(S, I, R) This function represents information gathered from successful (or even partially successful) graph matchings. Let S be the MIPS multiplication sample in Figure 15(a). After graph matching we know that the operators and operands along the P_b path will be involved in loading the value of `@L1.b`. Therefore *match*$(S, \texttt{lw}_{\texttt{r}\leftarrow\texttt{a}}, \mathit{load})$ will be very high. Similarly, since the paths from `@L1.b` and `@L1.c` convene in the $\texttt{mul}_0^3$ node, `mul` is highly likely to perform a multiplication, and therefore *match*$(S, \texttt{mul}_{\texttt{r}\leftarrow\texttt{r,r}}, \mathit{mul})$ will also be high.

When available, this is the most accurate information we can come by. It is therefore weighted highly in the *likelihood*(S, I, R) function.

—*sem*(S, R) The semantics of the sample itself is another important source of information, particularly when combined with an understanding of common code generation idioms.

As an example, let $S = \ulcorner$`main(){int b,c,a=b*c;}`$\urcorner$. Then we know that the corresponding assembly-code sample is much more likely to contain *load*, *store*, *mul*, *add*, or *shiftLeft* instructions, than (say) a *div* or a *branch*. Hence, for this example, $\mathit{sem}(S, \mathit{mul}) > \mathit{sem}(S, \mathit{add}) \gg \mathit{sem}(S, \mathit{branch})$.

—*sig*(I, R) The signature of an instruction can provide some clues as to the function it performs. For example, if I takes an address argument, it is quite likely to perform a *load* or a *store*, and if it takes a label argument, it probably does a *branch*. Similarly, an instruction (such as $\texttt{sw}_{\epsilon \leftarrow \texttt{r,a}}$ in Figure 15(a) or $\texttt{addl3}_{\epsilon \leftarrow \texttt{a,a,a}}$ in Figure 15(d)) that returns no result is likely to perform (some sort of) store operation.

—*name*(I, R) Finally, we take into account the name of the operator. This is based on the observation that, if I's mnemonic contains the string `"add"` or `"plus,"` it is more likely to perform (some sort of) addition than (say) a left shift. Unfortunately, this information can be highly inaccurate,[8] so *name*(I, R) is given a low weighting.

For many samples, these heuristics are highly successful. Often the reverse interpreter will come up with the correct semantic interpretation of an instruction after just one or two tries. In fact, while previous versions of the system relied exclusively on graph matching, the current implementation now mostly uses matching to compute the *match*(S, I, R) function.

There are still complex samples for which the reverse interpreter will not find a solution within a reasonable time. In such cases, a time-out function interrupts the interpreter and the sample is discarded.

5.2.3 *Primitive Instructions.* The instruction primitives used by the reverse interpreter largely determine the range of architectures that can be analyzed. A comprehensive set of complex primitives might map cleanly into a large number of instruction set architectures, but would slow down the reverse interpreter. A smaller set of simple primitives would be easier for the reverse interpreter to deal with, but might fail to provide a semantic interpretation for some instructions. As can be seen from Figure 19, the current implementation employs a small, RISC-like instruction set, which allows us to handle current RISCs and CISCs. It lacks, among other things, conditional expressions. This means that we currently cannot analyze instructions like the VAX's arithmetic shift (`ash`), which shifts to the left if the count is positive, and to the right otherwise.

In other words, the reverse interpreter will do well when analyzing an instruction set that is at the same or slightly higher level than its built-in primitives. However, dealing with micro-code-like or very complex instructions may well be beyond its capabilities. The reason is our need to always find the *shortest* semantic interpretation of every instruction. This means that when analyzing a complex instruction we will have to consider a very large number of short (and wrong) interpretations before we arrive at the longer, correct one. Since the number of possible interpretations grows exponentially with the length of the semantic interpretation, the reverse interpreter may quickly run out of space and time.

Although very complex instructions are currently out of favor, they were once very common. Consider, for example, the VAX's polynomial evaluation

[8]For example, consider that `s8addq` (Alpha) and `madd` (Mips R10000) are multiply-and-add instructions, and that, on the IBM/360, the mnemonic for addition is `A` and for load `L`.

Signature	Semantics	Comments
add $(\mathbb{I} \times \mathbb{I}) \rightarrow \mathbb{I}$	$add(a,b) = a + b$	Also *sub*, *mul*, *div*, and *mod*.
abs $\mathbb{I} \rightarrow \mathbb{I}$	$abs(a) = \mid a \mid$	Also *neg*, *not* and *move*.
and $(\mathbb{I} \times \mathbb{I}) \rightarrow \mathbb{I}$	$and(a,b) = a \wedge b$	Also *or*, *xor*, *shiftLeft*, and *shiftRight*.
ignore1 $(\mathbb{I} \times \mathbb{I}) \rightarrow \mathbb{I}$	$ignore1(a,b) = b$	Ignore first argument. Also *ignore2*.
compare $(\mathbb{I} \times \mathbb{I}) \rightarrow \mathbb{C}$	$compare(a,b) = \langle a < b, a = b, a > b \rangle$	Return the result of comparing a and b. Example: $compare(5,7) = \langle \mathbb{T}, \mathbb{F}, \mathbb{F} \rangle$.
isLE $\mathbb{C} \rightarrow \mathbb{B}$	$isLE(a) = a \neq \langle \_, \_, \mathbb{T} \rangle$	Return true if a represents a less-than-or-equal condition. Also *isEQ*, *isLT*, etc.
brTrue $(\mathbb{B} \times \mathbb{L})$	$brTrue(a,b) =$ *if* a *then* $PC \leftarrow b$	Branch on true. Also *brFalse*.
nop		No operation.
load $\mathbb{A} \rightarrow \mathbb{I}$	$load(a) = \mathtt{M}[a]$	Load an integer from memory.
store $(\mathbb{A} \times \mathbb{I})$	$store(a,b) = \mathtt{M}[a] \leftarrow b$	Store an integer into memory.
loadLit Lit $\rightarrow \mathbb{I}$	$loadLit(a) = a$	Load an integer literal.
loadAddr Addr $\rightarrow \mathbb{A}$	$loadAddr(a) = a$	Load a memory address.

Fig. 19. Reverse interpreter primitives. Available types are `Int` ($\mathbb{I}$), `Bool` ($\mathbb{B}$), `Address` ($\mathbb{A}$), `Label` ($\mathbb{L}$), and `Condition Code` ($\mathbb{C}$). `M[ ]` is the memory. $\mathbb{T}$ is True and $\mathbb{F}$ is False. $\mathbb{C}$ is an array of booleans representing the outcome of a comparison. While the current implementation only handles integer instructions, future versions will handle all standard C types. Hence the reverse interpreter will have to be extended with the corresponding primitives.

instruction ⌜`POLY`⌝ or the HP 2100 [Hewlett-Packard 1968] series computers' "alter-skip-group." The latter contains 19 basic opcodes that can be combined (up to eight at a time) into very complex statements. For example, the statement ⌜`CLA,SEZ,CME,SLA,INA`⌝ will `clear A`, `skip if E=0`, `complement E`, `skip if LSB(A)=0`, and then `increment A`. It is unlikely that any automated techniques would be able to reverse-engineer such instructions.

5.3 Analyzing Procedures and Procedure Calls

To generate code for procedures, we need to know which information needs to go in procedure headers and footers. Typically, the header will contain instructions or directives that reserve space on the runtime stack for new activation records. Similarly, to generate code for procedure calls we need to know how the caller passes actual arguments to the callee.

We use a technique we call *Difference Analysis* to deduce this type of information. The idea is simply to observe and draw conclusions from the differences between the assembly code generated from a sequence of increasingly more complex samples.

For example, to analyze procedure declarations we compile

```
void P(){};
void P(){int a;};
void P(){int a,b;};
void P(){int a,b,c;};
        ...
```

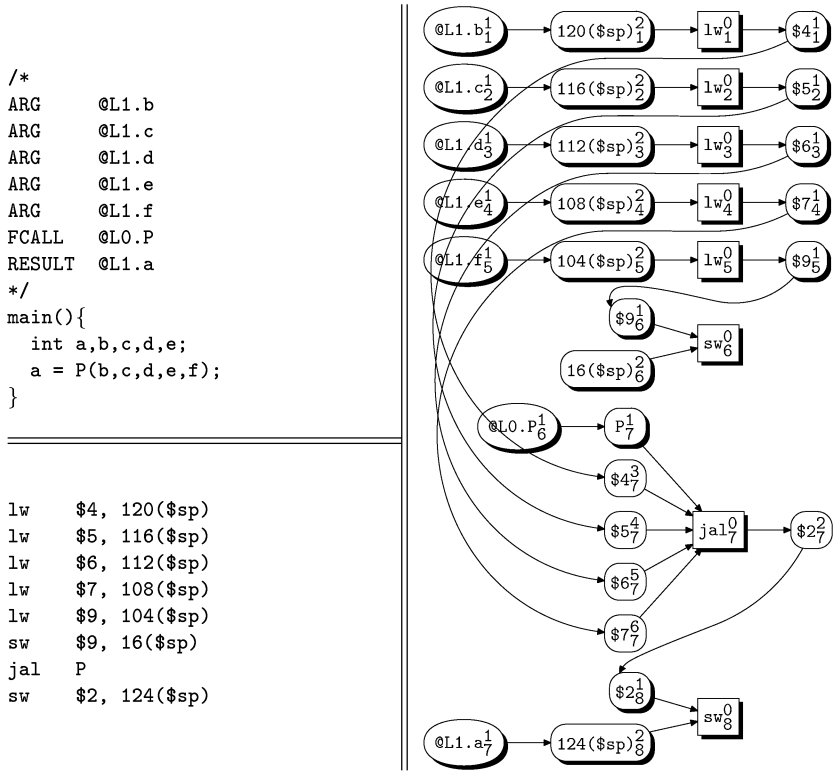

Fig. 20. MIPS function call sample. Note that, except for `@L1.f`, there is a path from all semantic operands to the procedure call instruction `jal`. This indicates that, on the MIPS, the first four arguments are passed in registers, the rest on the stack.

which will result in assembly-code samples which only differ in the amount of stack space allocated for activation records.

Unfortunately, things can get more complicated. On the VAX, for example, the procedure header must contain a register mask containing the registers that are used by the procedure and which need to be saved on procedure entry. Even if ADT were able to deduce these requirements, BEG has no provision for expressing them.

We use a similar technique to analyze procedure calls. We generate samples

```
P();
P(a);
P(a,b);
P(a,b,c);
   ...
```

and compare the resulting assembly code samples for differences. For the MIPS (see Figure 20), for example, we will find that the first four arguments are passed in registers `$4`–`$7`, and that any further arguments are passed on the stack. Strictly speaking, ADT does not understand anything about procedure

calling conventions. All it learns from analyzing the samples is an operational specification, such as

> *"To pass the first argument to a procedure on the MIPS, I move the argument into register* `$4`*. The second argument I move into* `$5`*. . . . The fifth argument, I move into* `16($sp)`*. . . ."*

The current version of ADT does not learn about floating point instructions, so the complications introduced by the interaction between floating point and integer arguments are currently ignored. We would deal with this problem the same way as outlined above, by drawing conclusions from the differences between successively more complex examples involving arguments of different types:

```
int i,j;
float x,y;
P(i,x);
P(x,i);
P(i,x,j);
P(x,i,y);
   ...
```

6. THE SYNTHESIZER

The synthesizer collects all the information gathered by previous phases and converts it into a machine description. Referring back to Figure 2, the synthesizer takes a description of instruction semantics as input (Figure 2(e)) and produces a BEG specification (Figure 2(d)) as output. The input to the synthesizer is realized as a set of Prolog predicates. This is illustrated in Figure 21.

If the ADT is part of a self-retargeting compiler, the generated machine description would be fed directly into BEG and the resulting code generator would be integrated into the compiler. If, instead, the ADT is used to speed up a *manual* compiler retargeting effort, the machine description could first be refined by the compiler writer.

The main difficulty of this phase is that there may not be a simple mapping from the intermediate code instructions emitted by the compiler into the machine code instructions provided by the ISA we have discovered. As an example, consider a compiler which emits an intermediate code instruction $\texttt{BranchEQ}(a, b, L) \equiv \texttt{IF } a = b \texttt{ GOTO } L$. Using the primitives in Figure 19, the semantics of `BranchEQ` can be described as $\mathit{brTrue}(\mathit{isEQ}(\mathit{compare}(a_1, a_2)), L)$. This, incidentally, is the exact semantics we derive for the MIPS' `beq` instruction. Hence, in this case, generating the appropriate BEG pattern matching rule is straightforward.

However, on most other machines the `BranchEQ` instruction has to be expressed as a combination of two machine code instructions. For example, on the Alpha we derive $\texttt{cmpeq}(a, b) \equiv \mathit{isEQ}(\mathit{compare}(a, b))$ and $\texttt{bne}(a, L) \equiv \mathit{brTrue}(a, L)$. To handle this problem, a special synthesizer phase (the *Combiner*) attempts to combine machine code instructions to match the semantics of

```
% Syntax of assembler comments. FORMAT: commentChar(prefix,suffix).
commentChar('/*', '*/'). commentChar('!', '').

% Syntax of integer literals supported by the assembler.
% FORMAT: intLitteral(base,lower/upper case,prefix,suffix).
intLitteral(16,lower,[0,x],[]). intLitteral(10,none,[],[]).

% Sizes and ranges of C types.
size('int', 4). size('float', 4). bitsPerByte(8).
intRange('int', -2147483647, 2147483648). intRange('short', -32767, 32768).

% A grammar describing assembler register syntax.
reg(1) --> ['%'], [f], [p], [].
reg(2) --> ['%'], [g], digit(0,0).
reg(3) --> ['%'], [i], digit(0,5).
reg(4) --> ['%'], [l], digit(0,7).
reg(5) --> ['%'], [o], digit(0,7).
reg(6) --> ['%'], [s], [p], [].

% A grammar describing assembler addressing modes.
addrMode(am(1,[int(I1)])) --> pInt(I1).
addrMode(am(2,[lab(L1)])) --> pLab(L1).
addrMode(am(3,[reg(R1)])) --> pReg(R1).
addrMode(am(4,[reg(R2),int(I2)])) --> ['['], pReg(R2), addsub, pInt(I2), [']'].
addsub --> [+]; [-].

% Information on how to pass actual arguments in function calls.
% FORMAT: actualArgument(argument number, register/memory, location).
actualArgument(1,passInRegister,'%o0').
actualArgument(6,passInRegister,'%o5').
actualArgument(7,passInMemoryLocation,am(4,[reg('%sp'),int('0x5c')])).
actualArgument(8,passInMemoryLocation,am(4,[reg('%sp'),int('0x60')])).

% Instructions that have a delay slot.
hasDelaySlot(be). hasDelaySlot(call).

% Argument restrictions. FORMAT:intCondition(opcode, argument number, min, max).
intCondition(add,2,-4096,4095).

% Instruction timings. FORMAT:instructionTime(opcode,addressing modes,normalized time).
instructionTime(add,[3,1,3],1). instructionTime(add,[3,3,3],1).

% Instruction semantics.
% FORMAT: instruction(output operands<-opcode<-input operands, expr).
% The 'expr' argument is an arbitrary expression-tree describing the
% computation performed by the instruction.
% In the add-example below, the expression describes that the Sparc add
% instruction computes the addition of the first and second operands and
% stores the result in the third. All operands are registers.
instruction([reg/3]<-add<-[reg/1,reg/2],
            e(+, int, [e(arg,int,[arg(1)]), e(arg,int,[arg(2)])])).
```

Fig. 21. A subset of the information gathered by ADT for the Sparc. The data is expressed as Prolog predicates. The comments are not part of the generated code.

intermediate code instructions. Again, we resort to exhaustive search. We consider any combination of instructions to see if combining their semantics will result in the semantics of one of the instructions in the compiler's intermediate code.[9] Any such combination results in a separate BEG pattern matching rule. See Figure 24(c) for an example.

Depending on the complexity of the machine description language, the synthesizer may have to contend with other problems as well. BEG, for example, has a powerful way of describing the relationship between different addressing modes, so called *chain rules*. A chain rule expresses under which circumstances two addressing modes have the same semantics. The chain-rules in Figure 24(a) to (b) express that the SPARC's register + offset addressing mode is the same as the register immediate addressing mode when the offset is 0. To construct the chain-rules, we consider the semantics S_A and S_B of every pair of addressing modes A and B. For each pair, we exhaustively assign small constants (such as 0 or 1) to the constant arguments of S_A and S_B, and we assign registers with hardwired values (such as the SPARC's `%g0`) to S_A's and S_B's register arguments. If the resulting semantics S'_A and S'_B are equal, we emit the corresponding chain-rule.

Figures 22, 23, 24, and 25 show parts of a BEG specification for the SPARC, generated by ADT:

—Figure 22 defines the nodes of the intermediate code trees that the generated back-end accepts. This part of the specification is currently hard-coded.

—Figure 23 lists the registers and addressing modes found by the ADT. `AddrMode4`, for example, is the SPARC's register + offset addressing mode. Note that ADT failed to find some registers.

—Figure 24 shows BEG chain-rules and rules for arithmetic and branches. Four points are noteworthy:

(1) The ADT has discovered that branch instructions take a delay slot. This information is acquired through yet another mutation analysis during the preprocessing stage.

(2) The ADT has discovered the 13-bit ranges of the SPARC's `cmp`- and `add`-immediate instructions. This is done through an assembler error analysis: Each instruction that takes a literal argument is submitted to the assembler with increasingly larger constants. When the constant is out of range the assembler will emit an error message. We continue this process using a binary-search-type algorithm, homing in on the upper and lower limits of each constant argument.

(3) The ADT has computed a constant *cost* for each instruction. We use a simplistic algorithm which executes and times each instruction (with each allowable combination of arguments) a large number of times, and computes the instructions' relative execution time costs.

[9]This is somewhat akin to Massalin's [1987] superoptimizer. The difference is that the superoptimizer attempts to find a *smallest* program, whereas the Combiner looks for *any* combination of instructions with the required behavior. The back-end generator is then responsible for selecting the best (cheapest) instruction sequence at compile-time.

```
CODE_GENERATOR_DESCRIPTION SPARC;

INTERMEDIATE_REPRESENTATION
NONTERMINALS IntExp, UsrLabel, ArgExp;
OPERATORS
   IntLiteral (val:  INTEGER)            -> IntExp;
   BaseReg                               -> IntExp;
   LoadInt                               IntExp -> IntExp;
   StoreInt                              IntExp * IntExp;
   Plus                                  IntExp + IntExp -> IntExp;
   Sub                                   IntExp * IntExp -> IntExp;
   Mult                                  IntExp + IntExp -> IntExp;
   Div                                   IntExp * IntExp -> IntExp;
   Mod                                   IntExp * IntExp -> IntExp;
   ShiftLeft                             IntExp * IntExp -> IntExp;
   ShiftRight                            IntExp * IntExp -> IntExp;
   UserLabel (lab:  INTEGER)             -> UsrLabel;
   DeclareLabel                          UsrLabel;
   BranchEQ                              UsrLabel * IntExp * IntExp;
   BranchGE                              UsrLabel * IntExp * IntExp;
   BranchGT                              UsrLabel * IntExp * IntExp;
   BranchLE                              UsrLabel * IntExp * IntExp;
   BranchLT                              UsrLabel * IntExp * IntExp;
   BranchNE                              UsrLabel * IntExp * IntExp;
   Call (id:STRING; ArgCount:CARDINAL) ArgExp;
   FunCall (id:STRING; ArgCount:CARDINAL) ArgExp -> IntExp;
   ActualArgInt (nr :  CARDINAL) IntExp  -> ArgExp;
   ConsArg ArgExp * ArgExp               -> ArgExp;
   NullArg                               -> ArgExp;

   REGISTERS AND ADDRESSING MODES

   RULES

INSERTS
   HARDWIRED CODE
END CODE_GENERATOR_DESCRIPTION SPARC.
```

Fig. 22. First part of the BEG specification for the SPARC. The `OPERATOR` section is hard-coded by ADT and provides an interface (the intermediate code) by which the compiler front-end communicates with the ADT/BEG-generated back-end.

(4) The ADT has discovered the implicit input (`%o0` and `%o1`) and output arguments (`%o0`) to the SPARC's ⌜`call .mul`⌝ instruction.

—Figure 25, finally, gives rules that handle procedure calls on the SPARC. `ConsArg` and `NullArg` are used to build up lists of actual arguments. Note that the ADT has discovered that the first six arguments should be passed in registers, and the remaining ones on the stack.

7. DISCUSSION AND SUMMARY

It is interesting to note that many of the techniques presented here have always been used manually by compiler writers. The fastest way to learn about code-generation techniques for a new architecture is to compile some small C

```
REGISTERS
   Reg_fp, Reg_g0, Reg_i0, Reg_i1, Reg_i2, Reg_i3, Reg_i4, Reg_i5,
   Reg_l0, Reg_l1, Reg_l2, Reg_l3, Reg_l4, Reg_l5, Reg_l6, Reg_l7,
   Reg_o0, Reg_o1, Reg_o2, Reg_o3, Reg_o4, Reg_o5, Reg_o6, Reg_o7,
   Reg_sp;

AVAIL (Reg_g0, Reg_i0, Reg_i1, Reg_i2, Reg_i3, Reg_i4, Reg_i5, Reg_l0,
   Reg_l1, Reg_l2, Reg_l3, Reg_l4, Reg_l5, Reg_l6, Reg_l7, Reg_o0,
   Reg_o1, Reg_o2, Reg_o3, Reg_o4, Reg_o5, Reg_o6, Reg_o7);

NONTERMINALS
   Register REGISTERS (
      Reg_fp, Reg_g0, Reg_i0, Reg_i1, Reg_i2, Reg_i3, Reg_i4, Reg_i5,
      Reg_l0, Reg_l1, Reg_l2, Reg_l3, Reg_l4, Reg_l5, Reg_l6, Reg_l7,
      Reg_o0, Reg_o1, Reg_o2, Reg_o3, Reg_o4, Reg_o5, Reg_o6, Reg_o7,
      Reg_sp
   );

   IntConstant   COND_ATTRIBUTES (val :  INTEGER);
   AddrMode2     COND_ATTRIBUTES (lab2_1 :  INTEGER);
   AddrMode4     ADRMODE
                    COND_ATTRIBUTES (int4_1 :  INTEGER)
                    (reg4_1 :  Register);
   Label         COND_ATTRIBUTES (lab :  INTEGER);
   Arg           ADRMODE;
```

Fig. 23. BEG definitions of SPARC registers and addressing modes. Note that register %g0 has been included in the list of registers available to the compiler for allocation. This is due to a limitation of the current implementation which does not include a test for registers with hardwired values.

or FORTRAN programs and examine the resulting assembly code. The ADT automates this task.

One of the major sources of problems when writing machine descriptions by hand is that the documentation describing the ISA, the implementation of the ISA, the assembler syntax, etc., is notoriously unreliable. Our system bypasses these problems by dealing directly with the hardware and system software. Furthermore, our system makes it cheap and easy to keep machine descriptions up to date with hardware and system software updates.

We will conclude this paper with a discussion of the *generality*, *completeness*, and *implementation status* of the ADT.

7.1 Generality

What range of architectures can an architecture discovery system possibly support? Under what circumstances might it fail?

As we have seen, our analyzer consists of three major modules: the Lexer, the Preprocessor, and the Extractor. Each of them may fail when attempting to analyze a particular architecture. The Lexer assumes a relatively standard assembly language, and will, of course, fail for unusual languages such as the one used for the Tera. The extractor may fail to analyze instructions with very complex semantics, since the reverse interpreter is worst-case exponential.

The Preprocessor's task is essentially to determine how pairs of instructions communicate with each other within a sample. Should it fail to do so for a

```
(a) RULE Register.a1 -> AddrMode4.res;
       COST 0;
       EVAL{res.int4_1 := 0;}
       EMIT{res.reg4_1 := a1;}

(b) RULE AddrMode4.a1 -> Register.res;
       CONDITION{(a1.int4_1 = 0)};
       COST 0;
       EMIT{res := a1.reg4_1;}

(c) RULE BranchEQ Label.a1 Register.a2 IntConstant.a3 ;
       CONDITION{(a3.val>=-4096) AND (a3.val<=4095)};
       COST 2;
       EMIT{printf("cmp %s,%i\n", a2, a3.val);
            printf("be L%i\n, a1.lab);
            printf("nop\n");
       }

(d) RULE Mult Register.a3(Reg_o0) Register.a4(Reg_o1) -> Register.res(Reg_o0);
       COST 15;
       TARGET a3;
       EMIT{printf("call .mul, 2"\n);
            printf("nop\n");
       }

(e) RULE Plus Register.a1 IntConstant.a2 -> Register.res;
       CONDITION{(a2.val>=-4096) AND (a2.val<=4095)};
       COST 1;
       EMIT {printf("add %s,%i,%s\n", a1, a2.val,res);}

(f) RULE Plus Register.a1 Register.a2 -> Register.res;
       COST 1;
       EMIT {printf("add %s,%s,%s\n", a1, a2,res);}
```

Fig. 24. Some of the BEG rules generated by the ADT for the SPARC. (a) and (b) are chain-rules that describe how to turn a *register + offset* addressing mode into a register (when the offset is 0), and vice versa. In (c) a comparison and a branch instruction have been combined to match the semantics of the intermediate code instruction `BranchEQ`. Rule (d), describes the SPARC's software multiplication routine `.mul`. Rules (e) and (f), finally, show the SPARC's two instructions for integer addition.

particular sample, the data-flow graph cannot be built, and that sample cannot be further analyzed. There are four basic ways for two instructions A and B to communicate:

Explicit registers. A assigns a value to a general-purpose register R. B reads this value.

Implicit registers. A assigns a value to a general purpose register R which is hardwired into the instruction. B reads this value.

Hidden registers. A and B communicate by means of a special-purpose register which is "hidden" within the CPU and not otherwise available to the user. Examples include condition codes and the `lo` and `hi` registers on the MIPS.

```
(a)  RULE NullArg -> Arg;
        COST 0;
(b)  RULE ConsArg Arg.a Arg.b -> Arg.c;
        COST 0;
(c)  RULE ActualArgInt.a Register({Reg_o0}) -> Arg;
        CONDITION {a.nr = 1};
        COST 0;
(d)  RULE ActualArgInt.a Register({Reg_o1}) -> Arg;
        CONDITION {a.nr = 2};
        COST 0;
(e)  RULE ActualArgInt.a Register({Reg_o2}) -> Arg;
        CONDITION {a.nr = 3};
        COST 0;
(f)  RULE ActualArgInt.a Register({Reg_o3}) -> Arg;
        CONDITION {a.nr = 4};
        COST 0;
(g)  RULE ActualArgInt.a Register({Reg_o4}) -> Arg;
        CONDITION {a.nr = 5};
        COST 0;
(h)  RULE ActualArgInt.a Register({Reg_o5}) -> Arg;
        CONDITION {a.nr = 6};
        COST 0;
(i)  RULE ActualArgInt.a Register.a1 -> Arg;
        CONDITION {(a.nr >= 7) AND (a.nr <= 11)};
        COST 4;
        EMIT {printf("st %s,[%%sp+%i]", a1, a.nr*4+64);}
(j)  RULE Call.a Arg;
        CONDITION{(a.ArgCount >= 0) AND (a.ArgCount <= 5)};
        COST 10;
        EMIT {printf("call %s%i\n",a.id,a.ArgCount); printf("nop\n");}
(k)  RULE Call.a Arg;
        CONDITION{(a.ArgCount >= 6) AND (a.ArgCount <= 12)};
        COST 10;
        EMIT {printf("call %s,%i\n",a.id,6); printf("nop\n");}
```

Fig. 25. BEG specification generated by ADT for SPARC procedure calls. Rules (a) and (b) are used to build up a list of actual arguments. Rules (c) through (h) handle the first six arguments to a function (which are passed in registers %o0–%o5), and rule (i) handles any further arguments which are passed on the stack. Rules (j) and (k), finally, are the top-level rules for procedure calls.

Memory. *A* and *B* communicate via the stack or main memory. Examples include stack-machines such as the Burroughs B6700.

The current implementation handles the first two, some special cases (such as condition codes) of the third, but not the last. For this reason, we are not currently able to analyze extreme stack-machines such as the Burroughs B6700.

There is no guarantee that either C Code Code Generation or Specification-Driven Code Generation will work for all combinations of architectures, compilers, and languages. We have already seen that some C compilers will be unsafe as back-ends for languages with garbage collection. Specification-Driven Code Generation-based compilers will also fail if a new ISA has features unanticipated when the back-end generator was designed. Version 1 of BEG, for example, did not support the passing of actual parameters in registers, and hence was unable to generate code for RISC machines. Version 1.5 rectified this.

7.2 Completeness and Code Quality

The quality of the code generated by a Self-Retargeting Code Generation-based compiler will depend on a number of issues:

The quality of the C compiler. Obviously, if the C compiler does not generate a particular instruction, then the ADT never find out about it.

The semantic gap between C and the target language. The architecture may have instructions that directly support a particular target language feature, such as exceptions or statically nested procedures. Since C lacks these features, the C compiler will never produce the corresponding instructions, and no Self-Retargeting Code Generation compiler will be able to make use of them. Note that this is no different from a C Code Code Generation-based compiler which will have to synthesize its own static links, exceptions, etc., from C primitives.

The completeness of the sample set. There may be instructions which are part of the C compiler's vocabulary, but which it does not generate for any of our simple samples. Consider, for example, an architecture with long and short branch instructions. Since our branching samples are currently very small (typically, `main(){int a,b,c; if (b<c) a=9;}`), it is unlikely that a C compiler would ever produce any long branches.

The power of the ADT. If a particular sample is too complex for us to analyze, we will fail to discover instructions present only in that sample.

The quality of the back-end generator. A back-end generated by BEG will perform no optimization, not even local common subexpression elimination. Regardless of the quality of the machine descriptions we produce, the code generated by a BEG back-end will not be comparable to that produced by a production compiler.

It is important to note that we are not trying to *reverse engineer* the C compiler's code generator. This is a task that would most likely be beyond automation. In fact, if the C compiler's back-end and the back-end generator use different code generation algorithms, the codes they generate may bear no resemblance to each other.

A somewhat surprising consequence is that we may be able to produce a *correct* back-end using input from an *incorrect* C compiler, provided that

(1) the compiler generates correct code for the trivial examples that we feed it, and
(2) the back-end generator that we use generates correct back-ends from correct specifications.

In other words, a compiler that generates the wrong code when optimizing complex expressions such as `(a*5+t/9)%r+x[66]` will work perfectly well for us, as long as it generates correct, unoptimized, code for simple samples such as `a=b+c`.

One potential problem with the ADT is that instructions are tested mostly in isolation. It is therefore conceivable that we would not pick up on complex

interactions between pairs of instructions, particularly for pairs that do not occur in any of our samples. To deal with this and similar issues, it is our intention to include an extensive *testing* phase in a future version of ADT. Since all of the information we collect is in a machine-readable form, such an implementation would be straight-forward. The idea would be to generate intermediate code for the back-end that would exercise every generated code-generation rule, and any combination of rules. The testing phase would potentially be able to discover that "the rule for `mulx` seems to work and the rule for `divx` seems to work, but when `mulx` is generated immediately after a `divx` the code no longer executes correctly."

7.3 Related Work

Many programs are distributed with "automatic configuration scripts" that run at installation time to provide the program with information about the environment in which it is running. AUTOCONF [MacKenzie and Elliston 1998], for example, generates configuration scripts that check for the presence of certain programs, libraries, header files, system services, and compiler characteristics that are necessary for a successful installation.

The work presented here is, in some sense, the ultimate extension of such programs. The ADT could be run as part of the installation of a compiler, determining not only aspects of the operating environment, but every important charcteristic of the underlying hardware as well.

There has been much work in the past on automatically determining the *runtime* characteristics of an architecture implementation. This information can be used to guide a compiler's code generation and optimization passes. Baker [1991] described a technique ("scheduling through self-simulation"), in which a compiler determines a good schedule for a basic block by executing and timing a few alternative instruction sequences. Rumor [Keppel 1995] has it that SunSoft uses a compiler-construction time variant of this technique to tune it schedulers. The idea is to derive a good scheduling policy by running and timing a suite of benchmarks. Each benchmark is run several times, each time with a different set of scheduling options, until a good set of options has been found.

In a similar vein, McVoy's `lmbench` [McVoy 1995] program measures the sizes of instruction and data caches. This information can be used to guide optimizations that increase code size, such as inline expansion and loop unrolling.

Pemberton's `enquire` [Pemberton 1991] program (which determines byte order and sizes and alignment of data types) is already in use by compiler writers. Parts of `enquire` have been included in our system.

Finally, Engler and Hsieh [2000] presented a natural extension to ADT, namely a tool (`DERIVE`) that derives instruction encodings. A user provides `DERIVE` with the assembler syntax of instructions, and `DERIVE` will return a C struct that specifies this encoding. In a vein similar to ADT, `DERIVE` extracts this information by exercising the native assembler with generated assembly-code samples.

7.4 Implementation Status

The current version of the prototype implementation of the ADT is general enough to discover the instruction sets of common RISC and CISC architectures. It has been tested on the integer instruction sets of five machines (Sun SPARC, Digital Alpha, MIPS, DEC VAX, and Intel x86), and has been shown to generate (almost) correct machine specifications for the BEG back-end generator. The areas in which the system is deficient relate to modules that are not yet implemented. For example, we currently do not test for registers with hardwired values (register `%g0` is always 0 on the SPARC), and so the BEG specification fails to indicate that such registers are not available for allocation.

In this paper we have described algorithms which deduce the register sets, addressing modes, and instruction sets of a new architecture. Obviously, there is much additional information needed to make a complete compiler, information which the algorithms outlined here are not designed to obtain. For example, consider the symbol table information needed by symbolic debuggers (`.stabs` entries), alignment directives (`.align`), and compiler hints (`.livereg` in Figure 3).

The implementation consists of 10,000 nonblank, noncomment lines of Prolog, 900 lines of shell scripts, 1500 lines of AWK (for generating the C code samples and parsing the resulting assembly code), and 800 lines of makefiles (to integrate the different phases). It is available for download from `http://www.cs.arizona.edu/~collberg/Research/AutomaticRetargeting`.

REFERENCES

BAKER, H. G. 1991. Precise instruction scheduling without a precise machine model. *Comput. Architect. News 19*, 6 (Dec.), 4–8.

BRĀZMA, A. 1995. Learning of regular expression by pattern matching. In *Computational Learning Theory 1995* (Barcelona, Spain). Lecture Notes in Computer Science, vol. 904. Springer Verlag, Berlin, Germany, 392–403.

CHASE, D. AND RIDOUX, O. 1990. C as an intermediate representation. comp.compilers articles numbers 90-08-046 and 90-08-063. Available online at `http://iecc.com/compilers/article.html`.

CIFUENTES, C. AND GOUGH, K. J. 1995. Decompilation of binary programs. *Softw. Pract. Exp. 25*, 7 (July), 811–829.

COLLBERG, C. 1997a. Automatic derivation of machine descriptions. In *Australasian Computer Science Conference 1997* (Sidney, Australia).

COLLBERG, C. S. 1997b. Reverse interpretation + mutation analysis = automatic retargeting. In *Conference on Programming Language Design and Implementation* (Las Vegas, NV). ACM Press, New York, NY, 57–70.

DIGITAL SYSTEMS RESEARCH CENTER. 1996. SRC Modula-3: Release history. Palo Alto, CA. Available on line at `http://www.research.digital.com/SRC/modula-3/html/history.html`.

EMMELMANN, H., SCHRÖER, F.-W., AND LANDWEHR, R. 1989. BEG—a generator for efficient back ends. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 227–237.

ENGLER, D. R. AND HSIEH, W. 2000. DERIVE: A tool that automatically reverse-engineers instruction encodings. *SIGPLAN Not. 35*, 7 (July), 12–22. (Special issue on SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization)

ENGLER, D. R. AND PROEBSTING, T. A. 1994. DCG: An efficient retargetable dynamic code generation system. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 4–7). *SIGPLAN Not. 29*, 11, 263–272.

FRASER, C. W., HENRY, R. R., AND PROEBSTING, T. A. 1992. BURG—fast optimal instruction selection and tree parsing. *SIGPLAN Not. 24*, 7 (Apr.), 68–76.

HEWLETT-PACKARD. 1968. *A Pocket Guide to Hewlett-Packard Computers*, Hewlett-Packard.

HOBBY, J. D. 1992. Introduction to MetaPost. In *EuroTEX '92 Proceedings*. Czechoslovak TeX Users Group, Prague, Czech Republic, 21–36.

HOLT, R. C. 1987. Data descriptors: A compile-time model of data and addressing. *ACM Trans. Program. Lang. Syst. 9*, 3, 367–389.

INTERACTIVE SOFTWARE ENGINEERING. 1996. ISE Eiffel in a nutshell. Interactive Software Engineering (now Eiffel Software, Inc.), Goleta, CA. Available online at `http://eiffel.com/eiffel/nutshell.html`.

KEPPEL, D. 1995. Compiler back-ends. comp.compilers article number 95-10-136. Available online at `http://iecc.com/compilers/article.html`.

LARUS, J. R. AND SCHNARR, E. 1995. EEL: Machine-independent executable editing. In *PLDI'95*. La Jolla, CA, ACM Press, New York, NY, 291–300.

MACKENZIE, D. AND ELLISTON, B. 1998. Autoconf, version 2.1.3. Available online at `http://www.gnu.org/manual/autoconf/ps/autoconf.ps.gz`.

MASSALIN, H. 1987. Superoptimizer—a look at the smallest program. In *Proceedings of the Second International Conference on Architechtural Support for Programming Languages and Operating Systems* (ASPLOS II), Palo Alto, CA, Oct. 5–8. ACM Press, New York, NY.

MCVOY, L. 1995. lmbench: Portable tools for performance analysis. Available online at `ftp://perelandra.cms.udel.edu/bench/lmbench.tar`.

PEMBERTON, S. 1991. The ergonomics of software porting: Automatically configuring software to the runtime environment—or—Everything you wanted to know about your C compiler, but didn't know who to ask. Available online at `http://www.cwi.nl/ftp/steven/enquire/enquire.ps`.

RAMSEY, N. AND JONES, S. P. 2000. A single intermediate language that supports multiple implementations of exceptions (abstract). In *Conference on Programming Language Design and Implementation*. *SIGPLAN Not. 35*, 5, 285–298.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM—a system for building customized program analysis tools. In *Symposium on Programming Language Design and Implementation* (Orlando, FL, June 20–24). *SIGPLAN Not. 29*, 6, 196–205.

STALLMAN, R. M. 1995. *Using and Porting GNU CC*, 2.7.2 ed. Free Software Foundation, Inc. Boston, MA. Available online at `http://www.gnu.ai.mit.edu/doc/doc.html`.

TERA COMPUTER COMPANY. 1995. Major system characteristics of the Tera supercomputer. Tera Computer Company (now Cray, Inc.) Seattle, WA. `http://www.tera.com/hardware-overview.html`.

Received January 2001; revised September 2001; accepted April 2002