Reverse Interpretation + Mutation Analysis = Automatic Retargeting

Christian S. Collberg

Department of Computer Science
The University of Auckland
Private Bag 92019
Auckland, New Zealand.
collberg@cs.auckland.ac.nz

Abstract

There are three popular methods for constructing highly retargetable compilers: (1) the compiler emits abstract machine code which is interpreted at run-time, (2) the compiler emits C code which is subsequently compiled to machine code by the native C compiler, or (3) the compiler's code-generator is generated by a back-end generator from a formal machine description produced by the compiler writer.

These methods incur high costs at run-time, compile-time, or compiler-construction time, respectively.

In this paper we will describe a novel method which promises 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 Unit* has been implemented. The current version 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. The tool is completely automatic and requires minimal input from the user: principally, the user needs to provide the internet address of the target machine and the command-lines by which the C compiler, assembler, and linker are invoked.

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 new hardware/operating system combinations 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 (CCCG), and Specification-Driven Code Generation (SDCG)), and then propose a new method (Self-Retargeting Code Generation (SRCG)) which overcomes these problems.

1.1 C Code Code Generation

The back-end of a CCCG 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 CCCG. Examples include early versions of the SRC Modula-3 compiler [2] and the ISE Eiffel compiler [7].

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 human programmers, 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 type, the CCCG compiled program will be very difficult to debug.

CCCG-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 [3].

[1] In some CCCG compilers the most expensive part of compilation is compiling the generated C code. For this reason both SRC Modula-3 and ISE Eiffel are moving away from CCCG. ISE Eiffel now uses a bytecode interpreter for fast turn-around time and reserves the CCCG-based compiler for final code generation. SRC Modula-3 now supports at least two SDCG back-ends, based on `gcc` and `burg`.

1.2 Specification-Driven Code Generation

The back-end of a SDCG 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` [17] machine specification, for example, can be several thousand lines long. Popular back-end generators such as BEG [6] and `burg` [9] 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. See Figure 15 for some example rules taken from a BEG machine description.

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. Examples: "... the manual says that the opcodes are named movsx..., but the assembler ... does not accept that. `(i386)`" "WARNING! There is a small i860 hardware limitation (bug?) which we may run up against ... we must avoid 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 the access to manuals, some specification writers seemed uncertain of exactly which constructs were legal. Examples: "Is this number right? `(mips)`," "Can this ever happen on i386? `(i386)`," "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. Example: "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. Example: "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/back-end generator is being targeted.

1.3 Self-Retargeting Code Generation

In this paper we will propose a new 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 (SRCG).

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.

The primary application of the architecture discovery unit 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 SRCG it will also become possible to build *self-retargeting compilers*, i.e. 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 "`A`". Originally designed to produce code for architectures `A1` and `A2`, `ac` is able to retarget itself to the `A3` architecture. The user only needs to supply the Internet address of an `A3` machine and the command lines by which the C compiler, assembler, and linker are invoked.

The architecture discovery package will have other potential uses as well. For example, machine-independent tools for editing of executables (EEL [13]), decompilation (Cifuentes [4]), and dynamic compilation (DCG [8]) 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 have 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 an assembly-code producing C compiler, 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.

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

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

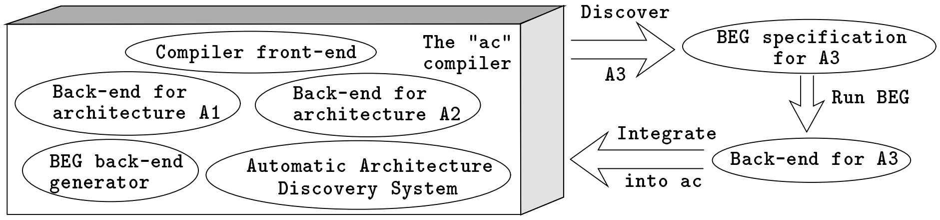


Figure 1: The structure of a self-retargeting compiler `ac` for some language A. The back-end generator BEG and the architecture discovery system are ntegrated into `ac`. The user can tell `ac` to retarget itself to a new architecture `A3` by giving the Internet address of an `A3` machine and the command lines by which the C compiler, assembler, and linker are invoked:
`ac -retarget -ARCH A3 -HOST kea.cs.auckland.ac.nz -CC 'cc -S -g -o %O %I' -AS 'as -o %O %I' -LD` ⋯.

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

The architecture discovery system 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 ⌜`main(){int b=5,c=6,a=b+c;}`⌝, conditionals ⌜`main(){int b=5,c=6,a=7; if(b<c)a=8;}`⌝, and procedure calls ⌜`main(){int b=5,a; a=P(b);}`⌝. 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 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⌝, and ⌜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.).

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

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 3 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`.

3.1 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; comments extend from a special comment-character to the end of the line; etc.

We use two fully automated techniques for discovering the details of a particular assembler: we can textually scan the assembly code produced by the C compiler or we can draw conclusions based on whether a particular assembly program is accepted or rejected by the assembler. For example, to discover the syntax of integer literals (Which bases are accepted? Which prefixes do the different bases use? Are upper, lower, and/or mixed case hexadecimal literals accepted?) we compile ⌜`main(){int a=1235;}`⌝ and scan the resulting assembly code for the constant 1235, in all the common bases. To discover the comment-character accepted by the assembler we start out with the assembly code produced from ⌜`main(){}`⌝, add an obviously erroneous line preceded by a suspected comment character, and submit it to the assembler for acceptance or rejection.

These techniques can be used for a number of similar tasks. In particular, we discover the syntax of addressing modes and registers, and the types of arguments (literals, labels, registers, memory references) each operator can take. 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].

Some assembly languages can be quite exotic. The Tera[5], 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.

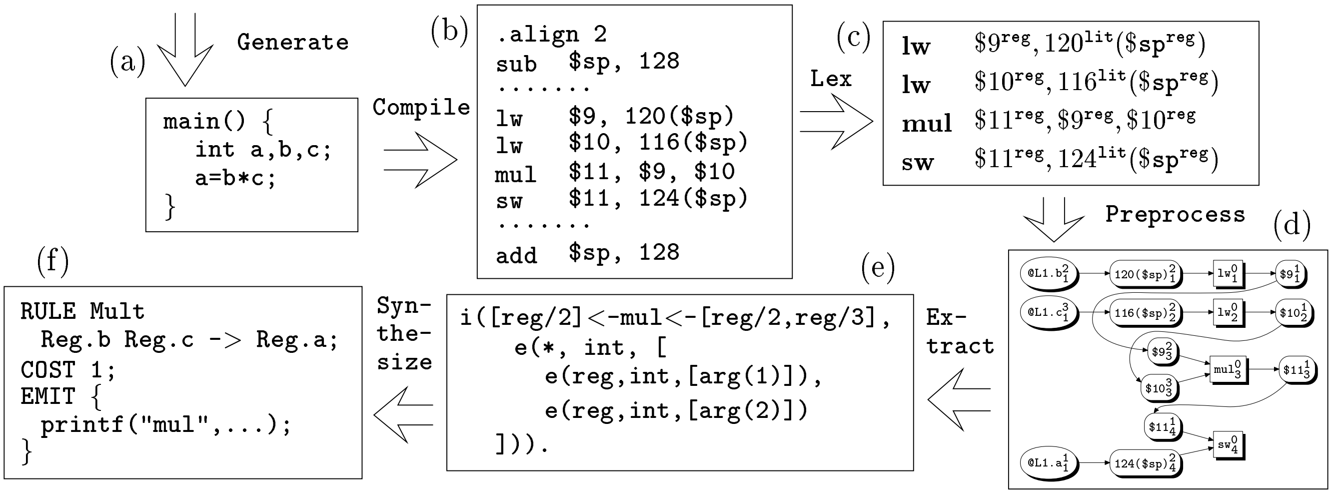


Figure 2: An overview of the major components of the architecture discovery system. The Generator produces a large number of small C programs (a) and compiles them to assembly on the target machine. The Lexer analyzes the raw assembly code (b) and extracts and tokenizes the instructions that are relevant 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.

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

```
/* 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;
}
```

(a)

```
/*
ADD @L1.a @L1.b @L1.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);
}
```

(b)

```
   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:
```

(c)

Figure 3: 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.

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.

The data-flow graph makes explicit the exact flow of information between individual instructions in a sample. This means that, for every instruction in every sample, we need to know where it takes its arguments and where it deposits its result(s). There are several major sources of confusion, some of which are illustrated in Figure 4.

For example, an instruction 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 8) takes its input argument and delivers its result in register `%eax`), as well as more recent ones when procedure call arguments are passed in registers (Figure 4(a)). If we cannot identify implicit arguments we obviously cannot

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

ld    [%fp+-0x8],%o0
ld    [%fp+-0xc],%o1
call  .mul,2
nop
st    %o0,[%fp+-0x4]
```

(a)

```
main(){int b,c,a=P(b,c);}
movl  -12(%ebp),%eax
pushl %eax
movl  -8(%ebp),%eax
pushl %eax
call  P
addl  $8,%esp
movl  %eax,%eax
movl  %eax,-4(%ebp)
```

(b)

```
main(){int a=P(34);}

call  P,1
mov   34,%o0
st    %o0,[%fp-4]
```

(c)

```
main(){int a=503<<a;}

ldq   $1, 184($sp)
addl  $1, 0, $2
ldil  $3, 503
sll   $3, $2, $4
addl  $4, 0, $4
stq   $4, 184($sp)
```

(d)

Figure 4: 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⌝.

accurately describe the flow of information in the samples.

As shown in Figure 4(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 can not 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 4. 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:

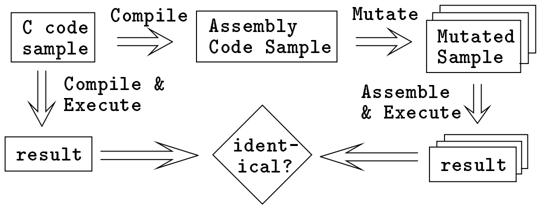


Figure 5 lists the available mutations.

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.

Even for trivial mutation like this, we must take special care for the mutation not to succeed by chance. As illustrated in Figure 6, countering random successes often involves judicious use of register clobbering.

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.

We will next consider three more profound preprocessing tasks (Live-Range Splitting, Implicit Argument Detection, and Register Definition/Use Computation) in detail.

4.3 Splitting Register Live-Ranges

Some samples (such as Figure 4(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 7 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 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 4(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.[4] Therefore the mutation in Figure 8(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, i.e. R is not an implicit argument of O (See Figure 8(b)). For those operator/register-pairs that fail the first step, we use *move* mutations to show that the registers are actually implicit arguments (Figure 8(c)).

[4] 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.

Original sample			

(1) $\mathtt{OP}_1$ A_1^1, R1, A_1^3
(2) $\mathtt{OP}_2$ A_2^1, A_2^2, A_2^3
(3) $\mathtt{OP}_3$ R1, A_3^2, A_3^3
	(2) $\mathtt{OP}_2$ A_2^1, A_2^2, A_2^3
	(1) $\mathtt{OP}_1$ A_1^1, R1, A_1^3
	(3) $\mathtt{OP}_3$ R1, A_3^2, A_3^3
	`move((1),after,(2))`
		(1) $\mathtt{OP}_1$ A_1^1, R1, A_1^3
		(2) $\mathtt{OP}_2$ A_2^1, A_2^2, A_2^3
		MOV -13,R1
		(3) $\mathtt{OP}_3$ R1, A_3^2, A_3^3
		`clobber(R1,after,(2))`
			(1) $\mathtt{OP}_1$ A_1^1, R1, A_1^3
			(2) $\mathtt{OP}_2$ A_2^1, A_2^2, A_2^3
			(3) $\mathtt{OP}_3$ R1, A_3^2, A_3^3
			(1') $\mathtt{OP}_1$ A_1^1, R1, A_1^3
			`copy((1),after,(3))`

	(1) $\mathtt{OP}_1$ A_1^1, R1, A_1^3 (3) $\mathtt{OP}_3$ R1, A_3^2, A_3^3 `delete((2))`	(1) $\mathtt{OP}_1$ A_1^1, R1, A_1^3 (2) $\mathtt{OP}_2$ A_2^1, A_2^2, A_2^3 (3) $\mathtt{OP}_3$ R2, A_3^2, A_3^3 `rename(R1,R2,(3))`	(1) $\mathtt{OP}_1$ A_1^1, R3, A_1^3 (2) $\mathtt{OP}_2$ A_2^1, A_2^2, A_2^3 (3) $\mathtt{OP}_3$ R3, A_3^2, A_3^3 `renameAll(R1, R3)`

Figure 5: 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. 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 the new register name chosen for a `rename` mutation.

Original Sample	`[delete((1))]`	`[clobber($1,before,(1)), ···, delete((1))]`	`[clobber($1,before,(1)), ···, delete((1))]`

`(1) ldq $1, 184($sp)`
`(2) addl $1, 0, $2`
`(3) ldil $3, 503`
`(4) sll $3, $2, $4`
`(5) addl $4, 0, $4`
`(6) stq $4, 184($sp)`
	`(2) addl $1, 0, $2`
	`(3) ldil $3, 503`
	`(4) sll $3, $2, $4`
	`(5) addl $4, 0, $4`
	`(6) stq $4, 184($sp)`
		`ldiq $1, -28793`
		`ldiq $2, 2556`
		`ldiq $3, 137`
		`ldiq $4, -22136`
		`(2) addl $1, 0, $2`
		`(3) ldil $3, 503`
		`(4) sll $3, $2, $4`
		`(5) addl $4, 0, $4`
		`(6) stq $4, 184($sp)`
			`ldiq $1, 234`
			`ldiq $2, -45256`
			`ldiq $3, 33135`
			`ldiq $4, 97`
			`(2) addl $1, 0, $2`
			`(3) ldil $3, 503`
			`(4) sll $3, $2, $4`
			`(5) addl $4, 0, $4`
			`(6) stq $4, 184($sp)`

(a)	(b)	(c)	(d)

Figure 6: Removing redundant instructions. In this example we are interested in whether instruction (1) is redundant. In (b) we delete the instruction; assemble, link, and execute the sample; and compare the result with that produced from the original sample in (a). If they are the same, (1) is redundant and can be removed permanently. Unfortunately, this simple mutation will succeed if register $\$1$ happens to contain the correct value. 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 ((c) and (d)) are constructed using different clobbering values. Both variants must succeed for the mutation to succeed.

4.5 Computing Definition/Use

Once 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 (e.g. on the VAX ⌜addl2 5,r1⌝ increments `R1` by 5), 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` (%$\mathtt{edx}_1$) is a pure definition, the second reference (%$\mathtt{edx}_2$) a use-definition, and the last one (%$\mathtt{edx}_3$) a pure use:

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

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

Figure 9 shows how we use the `copy` and `rename` mutations[5] to create a separate path from the first definition of a register `R1` to a reference of `R1` that is 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`.

[5]Necessary register clobbers have been omitted for clarity.

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.

The 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 assembly code operators and operands and there is an edge $A \rightarrow B$ if B uses a value stored in or computed by A.

For ease of identification, each node is labeled $\mathtt{N}_{\mathtt{i}}^{\mathtt{a}}$, where `N` is the operator or operand, `i` is the instruction number, and `a` is `N`'s argument number in the instruction. In Figure 10(b), for example, $\mathtt{\$11}_3^1$ refers to the use of register `$11` as the first argument to the third instruction (`mul`). `@L1.a` is a *data descriptor* [11] referring to the variable `a` at static nesting level 1.

Given the information provided by the various mutation analyses, building the graph for a sample S is straightfor-

Mutated Samples \ Mutation	`[rename(%eax,%ebx,(4)), clobber(%ebx,before,(4)]`	`[rename(%eax,%ebx,(4)), rename(%eax,%ebx,(3)), clobber(%ebx,before,(3)]`	`[rename(%eax,%ebx,(4)), rename(%eax,%ebx,(3)), rename(%eax,%ebx,(2)), clobber(%ebx,before,(2)]`
`(1) movl -12(%ebp), %eax`	`(1) movl -12(%ebp), %eax`	`(1) movl -12(%ebp), %eax`	`(1) movl -12(%ebp) %eax`
`(2) pushl %eax`	`(2) pushl %eax`	`(2) pushl %eax`	`movl -123, %ebx`
`(3) movl -8(%ebp), %eax`	`(3) movl -8(%ebp), %eax`	`movl -123, %ebx`	`(2) pushl %ebx`
`(4) pushl %eax`	`movl -123, %ebx`	`(3) movl -8(%ebp), %ebx`	`(3) movl -8(%ebp), %ebx`
`(5) call P`	`(4) pushl %ebx`	`(4) pushl %ebx`	`(4) pushl %ebx`
`(6) addl $8, %esp`	`(5) call P`	`(5) call P`	`(5) call P`
`(7) movl %eax, %edx`	`(6) addl $8, %esp`	`(6) addl $8, %esp`	`(6) addl $8, %esp`
`(8) movl %edx, -4(%ebp)`	`(7) movl %eax, %edx`	`(7) movl %eax, %edx`	`(7) movl %eax, %edx`
	`(8) movl %edx, -4(%ebp)`	`(8) movl %edx, -4(%ebp)`	`(8) movl %edx, -4(%ebp)`
(a)	(b)	(c)	(d)

Figure 7: Splitting the sample from Figure 4(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 4(e) shows the sample after regions have been split and registers renamed.

Original Sample	`[rename(%ecx,%eax,(1)), rename(%ecx,%eax,(2))]`	`[move((4),after,(5))]`
`(1) movl -8(%ebp),%ecx`	`(1) movl -8(%ebp), %ebx`	`(1) movl -8(%ebp),%ecx`
`(2) movl %ecx,%eax`	`(2) movl %ebx ,%eax`	`(2) movl %ecx,%eax`
`(3) cltd`	`(3) cltd`	`(3) cltd`
`(4) idivl -12(%ebp)`	`(4) idivl -12(%ebp)`	`(5) movl %eax,-4(%ebp)`
`(5) movl %eax,-4(%ebp)`	`(5) movl %eax,-4(%ebp)`	`(4) idivl -12(%ebp)`
(a)	(b)	(c)

Figure 8: 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) mutation will fail, since `%eax` is an implicit argument to `idivl`.

ward. A node is created for every operator and operand that occurs explicitly in S. 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.

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 towards a real understanding of the machine. From the graph in Figure 10(d), 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 analyse from the data-flow graphs and extract the function computed by each individual operator and operand. 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 11(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 non-empty.

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`.

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. The method we will describe next, on the other hand, is completely general but suffers from a worst-case exponential time complexity.

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

$$I \;::\; \mathtt{Sem} \times \mathtt{Prog} \times \mathtt{Env_{in}} \longrightarrow \mathtt{Env_{out}}$$

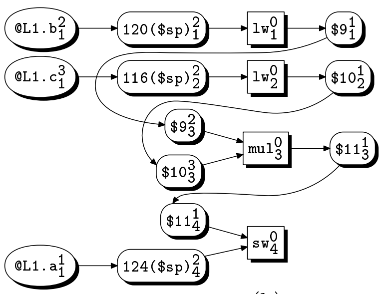
`Sem` is a mapping from instructions to their semantic interpretation, `Env` a mapping from memory locations, registers, etc., to values, and `Prog` is the sequence of instructions to be executed. The result of the interpretation is a new environment, with (possibly) new values in memory and registers. The example in Figure 12(a) adds 5 to the value in memory location 10 (`M[10]`) and stores the result in memory location 20.

Original Sample	[copy((1),after,(1)), rename(R1,R2,⟨(1'),(2)⟩)]	[copy((1),after,(1)), copy((2),after,(1')), rename(R1,R2,⟨(1'),(2'),(3)⟩)]
(1) $\texttt{OP}_1$ ···, $\texttt{R1}^{\texttt{D}}$, ···	(1) $\texttt{OP}_1$ ···, $\texttt{R1}^{\texttt{D}}$, ···	(1) $\texttt{OP}_1$ ···, $\texttt{R1}^{\texttt{D}}$, ···
(2) $\texttt{OP}_2$ ···, $\texttt{R1}^{\texttt{U or U/D?}}$, ···	(1') $\texttt{OP}_1$ ···, $\texttt{R2}^{\texttt{D}}$, ···	(1') $\texttt{OP}_1$ ···, $\texttt{R2}^{\texttt{D}}$, ···
(3) $\texttt{OP}_3$ ···, $\texttt{R1}^{\texttt{U or U/D?}}$, ···	(2) $\texttt{OP}_2$ ···, R2, ···	(2') $\texttt{OP}_2$ ···, $\texttt{R2}^{\texttt{U/D}}$, ···
(4) $\texttt{OP}_4$ ···, $\texttt{R1}^{\texttt{U}}$, ···	(3) $\texttt{OP}_3$ ···, $\texttt{R1}^{\texttt{U or U/D?}}$, ···	(2) $\texttt{OP}_2$ ···, $\texttt{R1}^{\texttt{U/D}}$, ···
	(4) $\texttt{OP}_4$ ···, $\texttt{R1}^{\texttt{U}}$, ···	(3) $\texttt{OP}_3$ ···, R2, ···
		(4) $\texttt{OP}_4$ ···, $\texttt{R1}^{\texttt{U}}$, ···
(a)	(b)	(c)

Figure 9: Computing definition/use information. The first occurence 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.

```
/* MUL @L1.a @L1.b @L1.c */
main(){
  int b=313,c=109,a=b*c;
}
(1) lw  $9, 120($sp)
(2) lw  $10, 116($sp)
(3) mul $11, $9, $10
(4) sw  $11, 124($sp)
```

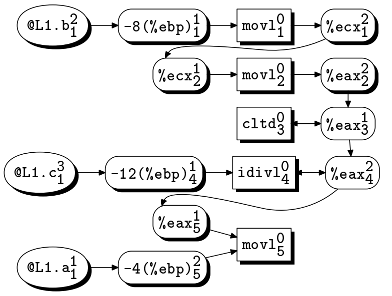
(a)



(b)

```
/*
DIV @L1.a @L1.b @L1.c
*/
main(){
  int b,c,a=b/c;
}
movl -8(%ebp),%ecx
movl %ecx,%eax
cltd
idivl -12(%ebp)
movl %eax,-4(%ebp)
```

(c)



(d)

Figure 10: MIPS multiplication (a-b) and x86 division (c-d) samples and their corresponding data-flow graphs. Rectangular boxes are operators, round-edged boxes are operands, and ovals represent source code variables. Note that, in (d), the implicit arguments to `cltd` and `idivl` are explicit in the graph (they are drawn unshaded). Also note that the edges $\texttt{\%eax}_4^2 \rightarrow \texttt{idivl}_4^0$ and $\texttt{idivl}_4^0 \rightarrow \texttt{\%eax}_4^2$ indicate that `idivl` reads *and* modifies `%eax`. All the graph drawings shown in this paper were generated automatically as part of the documentation produced by the architecture discovery system.

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

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

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 12(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 (non-deterministically) 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 they can be combined to model arbitrarily complex machine instructions. For example, $\texttt{addl3}_{\texttt{c}\leftarrow\texttt{a,a,a}}(a,b,c) = \textit{store}(a, \textit{add}(\textit{load}(b), \textit{load}(c)))$ 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

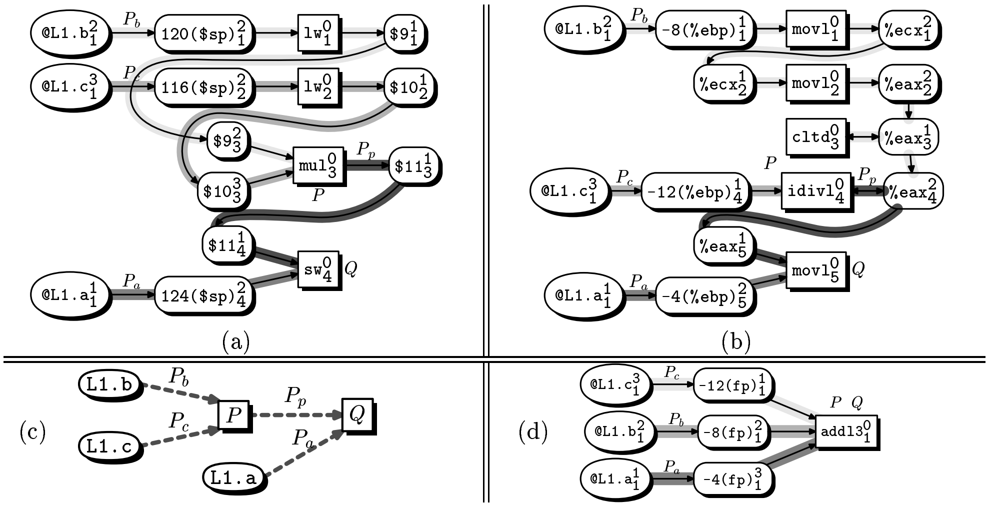


Figure 11: Data-flow graphs after graph matching. (a) is MIPS multiplication, (b) is x86 division, and (d) is VAX addition. In (a), $P = \mathtt{mul}_3^0$, $Q = \mathtt{sw}_4^0$. Hence, on the MIPS, `lw` is responsible for loading the r-values of `b` and `c`, `mul` performs multiplication, and `sw` stores the result.

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

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

Figure 12: (a) shows the result of an interpreter I evaluating a program $[\![\mathtt{store}(20,\mathtt{add}(\mathtt{load}(10),5))]\!]$, given an environment $\{\mathtt{M}[10]=7,\mathtt{M}[20]=9\}$. 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 $\mathtt{Env_{in}}$ into $\mathtt{Env_{out}}$, the `add` instruction should have the semantics $\mathtt{add}(x,y)=x+y$.

MIPS multiply-and-add. Figure 14 lists the most important primitives, and in Section 5.2.3 we will discuss the choice of primitives in detail.

The example in Figure 13 shows the reverse interpretation of the sample in Figure 10(a-b). 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 $\mathtt{am}^1_{\mathtt{a}\leftarrow\mathtt{r,c}}$ 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 $\mathtt{M}[\mathtt{@L1.a}] = 34117$. The reverse interpreter does this by enumerating all possible semantic interpretations of `mul`, until one is found that produces the correct $\mathtt{Env_{out}}$.

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 13, $\mathtt{mul}_{\mathtt{r}\leftarrow\mathtt{r,r}}$ could get any one of the semantics $\mathtt{mul}_{\mathtt{r}\leftarrow\mathtt{r,r}}(a,b) = a*b$, $\mathtt{mul}_{\mathtt{r}\leftarrow\mathtt{r,r}}(a,b) = a*b*1$, $\mathtt{mul}_{\mathtt{r}\leftarrow\mathtt{r,r}}(a,b) = b^2*a/b$, etc. Since most machine instructions have very simple semantics, we should strive for the simplest (shortest) interpretations.

Secondly, there may be situations where a set of samples will allow several conflicting interpretations. To see this, let S=⌜`main(){int b=2,c=1,a=b*c;}`⌝ be a sample, and let the multiplication instruction generated from S be named `mul`. Given that $\mathtt{Env_{in}} = \{\mathtt{b}=2,\mathtt{c}=1\}$ and $\mathtt{Env_{out}} = \{\mathtt{b}=2,\mathtt{c}=1,\mathtt{a}=2\}$, the reverse interpreter could reasonably conclude that $\mathtt{mul}(a,b) = a/b$, or even $\mathtt{mul}(a,b) = a-b+1$. 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:[6] 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.

Thirdly, the reverse interpreter might produce the wrong result if its arithmetic is different from that of the target architecture. We use `enquire` [16] 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 $a_1 \leftarrow \mathtt{am}^1_{\mathtt{a}\leftarrow\mathtt{r,c}} \leftarrow (120,\$\mathtt{sp})$ 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 13 `@L1.a`, `@L1.b`, and `@L1.c` are

[6] Thanks to John Hamer for pointing this out.

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

Figure 13: Reverse interpretation example. The data-flow graph in Figure 10(a) has been broken up into seven primitive instructions, each one of the form `result` $\leftarrow$ `operator` $\leftarrow$ (`arguments`). $\mathtt{am}^1_{\mathtt{a}\leftarrow\mathtt{r,c}}$ represents the "register+offset" addressing mode. The a_i's are "pseudo-registers" which hold the result of address calculations. 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. $\mathtt{lw}_{\mathtt{r}\leftarrow\mathtt{a}}$, for example, takes an address as argument and returns a result in a register.

addressed as $124 + \$\mathtt{sp}$, $120 + \$\mathtt{sp}$, $116 + \$\mathtt{sp}$, respectively, for some unknown value of $\$\mathtt{sp}$? We handle this by initializing every register not initialized by the sample itself to a unique value ($\$\mathtt{sp} \leftarrow \perp_{\$\mathtt{sp}}$). The interpreter can easily determine that a symbolic value $124+\perp_{\$\mathtt{sp}}$ must correspond to the address `@L1.a` after having analyzed a couple of samples such as ⌜`main(){int a=1452;}`⌝.

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 13. The interpreter might guess that $\mathtt{mul}_{\mathtt{r}\leftarrow\mathtt{r,r}}(a,b) = \mathit{mul}\,(a, \mathit{add}\,(100,b))$, 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 $\mathtt{mul}_{\mathtt{r}\leftarrow\mathtt{r,r}}(a,b) = \mathit{mul}\,(a, \mathit{add}\,(110,b))$.

This is clearly the wrong strategy. In fact, an unsuccessful interpretation (one that fails to produce the correct $\mathtt{Env}_{\mathtt{out}}$) 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

$$L(S,I,R) = c_1M(S,I,R)+c_2P(S,R)+c_3G(I,R)+c_4N(I,R)$$

where the c_i's are implementation specific weights and M, P, G, and N are functions defined below.

$M(S,I,R)$ This function represents information gathered from successful (or even partially successful) graph matchings. Let S be the MIPS multiplication sample in Figure 11(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 $M(S, \mathtt{lw}_{\mathtt{r}\leftarrow\mathtt{a}}, \mathit{load})$ will be very high. Similarly, since the paths from `@L1.b` and `@L1.c` convene in the $\mathtt{mul}^3_0$ node, `mul` is highly likely to perform a multiplication, and therefore $M(S, \mathtt{mul}_{\mathtt{r}\leftarrow\mathtt{r,r}}, \mathit{mul})$ will also be high.

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

$P(S,R)$ 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=⌜`main(){int b,c,a=b*c}`⌝. 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, $P(S, \mathit{mul}) > P(S, \mathit{add}) \gg P(S, \mathit{branch})$.

$G(I, R)$ 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 $\mathtt{sw}_{\epsilon \leftarrow \mathtt{r,a}}$ in Figure 11(a) or $\mathtt{addl3}_{\epsilon \leftarrow \mathtt{a,a,a}}$ in Figure 11(d)) that returns no result is likely to perform (some sort of) store operation.

$N(I, R)$ Finally, we take into account the name of the instruction. 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, so $N(I, R)$ 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 $M(S, I, R)$ 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 14, 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 instruction ⌜POLY⌝ or the HP 2100 [10] series computers' "alter-skip-group." The latter contains 19 basic opcodes that can be combined (up to 8 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`.

6 The Synthesizer

The Synthesizer collects all the information gathered by previous phases and converts it into a BEG specification. If the discovery system is part of a self-retargeting compiler, the machine description would be fed directly into BEG and the resulting code generator would be integrated into the compiler. If the discovery system 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. As an example, consider a compiler which emits an intermediate code instruction $\mathtt{BranchEQ}(a, b, L) =$ `IF` $a = b$ `GOTO` L. Using the primitives in Figure 14, 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 straight-forward.

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 $\mathtt{cmpeq}(a, b) = \mathit{isEQ}(\mathit{compare}\,(a, b))$ and $\mathtt{bne}(a, L) = \mathit{brTrue}\,(a, L)$. To handle this problem, a special Synthesizer phase (the *Combiner*) attempts to combine machine code instructions to match the semantics of 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.[7] Any such combination results in a separate BEG pattern matching rule. See figure Figure 15(d) 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 15(b-c) 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 and S_B's register arguments. If the resulting semantics S'_A and S'_B are equal, we produce the corresponding chain-rule.

7 Discussion and Summary

It is interesting to note that many of the techniques presented here have always been used manually by compiler

[7] This is somewhat akin to Massalin's [14] 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.

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.

Figure 14: 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.

(a)
```
NONTERMINALS AddrMode4 ADRMODE
   COND_ATTRIBUTES (int4_1 :  INTEGER) (reg4_1 :  Register);
```

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

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

(d)
```
RULE BranchEQ Label.a1 Register.a2 IntConstant.a3 ;
   CONDITION{(a3.val>=-4096) AND (a3.val<=4095)};
   COST 2;
   EMIT{print "cmp", a2 "," a3.val; print "be", "L" a1.lab; print "nop"}
```

(e)
```
RULE Mult Register.a3(Reg_o0) Register.a4(Reg_o1) -> Register.res(Reg_o0);
   COST 15; TARGET a3;
   EMIT{print "call .mul, 2"; print "nop"}
```

Figure 15: Part of a BEG specification for the SPARC, generated automatically by the architecture discovery system. (a) shows the declaration of the "register+offset" addressing mode. (b) and (c) 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 (d) a comparison and a branch instruction have been combined to match the semantics of the intermediate code instruction `BranchEQ`. Note how the architecture discovery system has detected that the integer argument to the `cmp` instruction has a limited range. (e), finally, describes the SPARC's software multiplication routine `".mul"`. Note that we have discovered the implicit input (`%o0` and `%o1`) and output argument (`%o0`) to the `call` instruction.

writers. The fastest way to learn about code-generation techniques for a new architecture is to compile some small C or FORTRAN program and examine the resulting assembly code. The architecture discovery unit 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 architecture discovery system.

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 (being worst-case exponential) may simply "run out of time."

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 the data-flow graph cannot be built, and that sample cannot be further analyzed. There are basically four different 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.

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.

Furthermore, there is no guarantee that either CCCG or SDCG will work for all architecture/compiler/language combinations. We have already seen that some C compilers will be unsafe as CCCG back-ends for languages with garbage collection. SDCG-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.

7.1.1 Completeness and Code Quality

The quality of the code generated by an SRCG compiler will depend on a number of things:

The quality of the C compiler. Obviously, if the C compiler does not generate a particular instruction, then we will 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 SRCG compiler will be able to make use of them. Note that this is no different from a CCCG-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 architecture discovery system. 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.

7.2 Implementation Status and Future Work

The current version of the prototype implementation of the architecture discover system is general enough to be able to discover the instruction sets of common RISC and CISC architectures. It has been tested on the integer[8] 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. As an example, consider the symbol table information needed by symbolic debuggers (`".stabs"` entries).

Furthermore, to generate code for a procedure we need to know which information needs to go in the procedure header and footer. Typically, the header will contain instructions or directives that reserve space on the runtime stack for new activation records. To deduce this information we can simply observe the differences between the assembly code generated from a sequence of increasingly more complex procedure declarations. For example, compiling ⌜`int P(){}`⌝, ⌜`int P(){int a;}`⌝, ⌜`int P(){int a,b;}`⌝, etc., will result in procedure headers 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 the architecture discover system were able to deduce these requirements, BEG has no provision for expressing them.

[8] At this point we are targeting integer instruction sets exclusively, since they generally exhibit more interesting idiosyncrasies than floating point instruction sets.

7.2.1 Hardware Analysis

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 [1] describes 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 [12] has it that SunSoft uses a compiler-construction time variant of this technique to tune their 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` [15] 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.

Finally, Pemberton's `enquire` [16] program (which determines endian-ness and sizes and alignment of data types) is already in use by compiler writers. Parts of `enquire` have been included into our system.

It is our intention to include more of these techniques in future versions of the architecture discovery system. At the present time only crude instruction timings are performed. More detailed information would not be useful at this point, since BEG would be unable to make use of it.

7.2.2 Current Status

The system is under active development. The implementation currently consists of 10000 non-blank, non-comment lines of Prolog, 900 lines of shell scripts (mostly for communicating with the machine being analyzed), 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).

Acknowledgments

Thanks to Peter Fenwick and Bob Doran for valuable information about legacy architectures, and to the anonymous referees for helping me to greatly improve the presentation.

References

[1] Henry G. Baker. Precise instruction scheduling without a precise machine model. *Computer Architecture News*, 19(6), December 1991.

[2] Digital Systems Research Center. Src modula-3: Release history. `http://www.research.digital.com/SRC/modula-3/html/history.html`, 1996.

[3] David Chase and Oliver Ridoux. C as an intermediate representation. comp.compilers article numbers 90-08-046 and 90-08-063, August 1990. Retrieve from `http://iecc.com/compilers/article.html`.

[4] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software – Practice & Experience*, 25(7):811–829, July 1995.

[5] Tera Computer Company. Major system characteristics of the Tera supercomputer, November 1995. `http://www.tera.com/hardware-overview.html`.

[6] Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. Beg - a generator for efficient back ends. In *SIGPLAN 89 Conference on Programming Language Design and Implementation*, pages 227–237, 1989.

[7] Interactive Software Engineering. ISE Eiffel in a nutshell. `http://eiffel.com/eiffel/nutshell.html`, 1996.

[8] D. R. Engler and Todd A. Proebsting. DCG: An efficient retargetable dynamic code generation system. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[9] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG – fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 24(7):68–76, April 1992.

[10] Hewlett-Packard. *A Pocket Guide to Hewlett-Packard Computers*, 1968.

[11] Richard C. Holt. Data descriptors: A compile-time model of data and addressing. *ACM Transactions on Programming Languages and Systems*, 9(3):367–389, 1987.

[12] David Keppel. Compiler back-ends. comp.compilers article number 95-10-136, October 1995. Retrieve from `http://iecc.com/compilers/article.html`.

[13] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *PLDI'95*, pages 291–300, La Jolla, CA, June 1995.

[14] Harry Massalin. Superoptimizer – a look at the smallest program. In *Proceedings Second International Conference on Architechtural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, California, October 1987.

[15] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis, 1996. In *USENIX Annual Technical Conference*, San Diego, California, January 1996.

[16] Steven Pemberton. Enquire 4.3, 1990.

[17] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., 2.7.2 edition, November 1995.
`http://www.gnu.ai.mit.edu/doc/doc.html`.