Interprocedural Data Flow Recovery of High-Level Language Code from Assembly*

Cristina Cifuentes and Antoine Fraboulet†
Department of Computer Science and Electrical Engineering
The University of Queensland
Brisbane, Qld 4072, Australia
Email: {cristina,afrab}@csee.uq.edu.au

Technical Report 421
December 1997

Abstract

We evaluate a CISC interprocedural data flow technique for the recovery of high-level language code from assembly code; extended register copy propagation; on RISC assembly code, and provide a method for the partial construction of a RISC assembly to high-level language tool. The data flow technique is suitable for eliminating machine dependencies from the assembly code, such as registers, condition codes and stack references, and for the recovery of high-level language expressions and parameters, as well as statements.

The presented method is useful for recovery of code, maintenance, porting and program understanding.

The method has been implemented as part of *asm2c*, an assembly to C translator for SPARC. The program was tested against SPEC95 assembly programs generated by a C compiler. Results using both unoptimized and optimized cases are presented.

1 Introduction

Recovery of high-level language code from assembly code is an area of research that has not been widely researched in recent years, partly due to the complexity of the problem, and the lack of techniques available in this field. Nevertheless, it is well known within the software maintenance community that 10% of existing operational software is written in assembly and that only few people know how to modify such programs in organizations that still use them [12, 1]. Even further, recent studies have shown that 30% of existing software cannot be recompiled due to lost source code or the unavailability of the compiling tools and libraries used to create the software in the first place. In both cases, changes to the system need to be done at the assembly level. These figures point to the

*This work was sponsored by The University of Queensland.
†Current address: ENS, Lyon, France. Email: antoine.fraboulet@ens-lyon.fr

need for research into ways of translating machine and assembly code to high-level languages, based on sound compiler technology. In essence, the problem becomes one of program comprehension; being able to understand existing code generated by compiler tools, and to apply techniques which will "undo" what the compiler/assembler has done, therefore transforming low-level assembly-like code into a higher level of abstraction that resembles today's high-level language (HLL) programs.

Translation of assembly to high-level language code requires a series of analyses in order to abstract away from the hardware features of assembly language itself, and to recover the high-level features available in most common programming languages. The aim of this process is to recover high-level language code which is comparable to that produced by native programmers. The main types of analyses are:

1. *data flow analysis* to recover high-level language expressions, parameters, and function return values, and to remove hardware references from the code, such as registers, pipeline references and the stack;

2. *control flow analysis* to recover control structure information, such as loops and conditionals, and their nesting level; and

3. *type analysis* to recover high-level type information for variables and parameters in the code.

Data flow analysis deals with the recovery of high-level language expressions and parameters, and the removal of hardware references from the assembly code. The analysis aims at machine-independent techniques for solving this problem on CISC and RISC machines. This analysis is the purpose of this paper and is based on compiler optimization theory.

Recovery of control structure in a program is based on graph theory [8] and *control flow analysis*. In assembly code, control transfer is expressed in the form of conditional or unconditional jumps, and procedure calls. Potentially, all unconditional jumps are equivalent to `goto` statements in a high-level language program. However, control flow analysis can determine whether such a jump is a backward jump into the code (i.e. it creates a loop in the program), a redundant jump, or a needed `goto`. Clearly, minimizing the number of `goto` statements used throughout the code improves the understanding of the program; these techniques are explained in [4].

Type analysis deals with the recovery of HLL type information. This is an area that has been studied in the context of functional and object oriented languages, which infer the type of variables based on the context of the use of the variable (e.g. Self [14]). This type of analysis has not been studied in relation to translations from assembly code to a higher level language, but it is clearly desirable to regenerate HLL code which makes use not only of base data types (byte, char, integer, real) but also higher level ones (arrays, structures).

In this paper we concentrate on the *data flow analysis* techniques as applied to RISC assembly code. We show that existing CISC techniques need minor modifications in order to work on RISC code. We test the techniques using a

large set of assembly programs created by compiling the SPEC95 benchmarks to asssembly code. The rest of this paper is structured in the following way: §2 discusses previous work in the area, §3 explains the intermediate representation used in this analysis, §4 explains the extended register copy propagation technique and its applicatio to assembly code, §5 shows results obtained with the assembly version of the SPEC95 benchmarks. Finally, §6 provides conclusions on this work.

2 Previous Work

Recovery of high-level language code from assembly code has not been widely studied in the literature. A few techniques were studied in the 70s, mainly in what we would consider nowadays toy languages: MIXAL and Varian Data Machine's assembler. Both techniques were studied within a decompilation of assembly language framework. *Text compression* was used to eliminate intermediate loads and stores [10], and *expression condensation* was used to combine two or more intermediate (assembly-like) instructions into an equivalent expression [9]. Both these methods were based on standard forward substitution analysis. The assembly languages used did not have to deal with propagation of interprocedural information, use of register arguments, or pipeline considerations. There was not even a mention of push and pop type of instructions.

More recently, a method for optimization of condition codes based on reach analysis was used to translate microprocessor object code for the i8085 into PL/1 code [7]. In the area of decompilers, a method of *extended register copy propagation* was used in the *dcc* decompiler to recover C code from binary executable files [3]. The technique was tested against small benchmark programs, and due to the nature of 80286 code, there were not many optimizations in the binary code itself. Both these methods were also based on forward substitution.

This paper is an analysis of the extended register copy propagation technique, as applied to SPARC (RISC) code, and the testing of the technique in both an unoptimized and optimized environment. The aim of the technique is for a machine-independent analysis; as such, this article tests the generality of the *dcc* (CISC) method and identifies areas of further research for optimized code.

3 Intermediate Representation

Assembly programs are parsed and stored in an intermediate representation suitable for high-level language recovery analysis. There are two parts to the representation: a control flow graph for each procedure, and a high-level opcode for each instruction. However, during the parsing of the code, machine idioms are checked for in order to remove particularities of the machine itself. Most of these idioms relate to the synthetic instructions described in the SPARC manual [13]; we replace such instructions by more expressive ones. For example, the increment instruction is replaced by an addition instruction with explicit operands rather than implicit ones, and a subtract with carry using destination

register `%g0` is replaced by a compare instruction (as register `%g0` is hardwired to zero). We also consider the removal of instructions such as `save`, `nop`, and `restore` an idiom as these instructions do not provide us with information that needs to be expressed in a high-level language; they are mere conventions used by the machine.

Control Flow Graph

The control flow graph (CFG) of a procedure stores information about its basic blocks. There are several types of nodes based on the last instruction on that block or the existence of a label on the target block; these are:

- 1-way: block that ends with an unconditional branch,
- 2-way: block that ends with a conditional branch,
- n-way: block that ends with an unconditional branch on a register,
- call: block that ends with a procedure call,
- ret: block that ends on a return/restore, and
- fall: (fall through) block that is followed by a labelled block (i.e. there is transfer of control to the next instruction of this block).

Constructing the CFG is straight-forward, except for extra analysis needed in the case of n-way nodes. When a branch on a register is met, intraprocedural (backwards) slicing [15] is performed on the instruction using that register, in order to obtain the set of instructions the indirect branch depends on. If the register can take a series of different values, these values are determined by a forward walk of the slice conditions, making it possible to determine the size of the table and the offsets to search for. On the few cases were an indirect jump cannot be resolved, we flag the basic block as having zero out-edges and the graph as being an incomplete one (a non-connected graph given that all the assembly code is parsed into basic blocks). The CFG generation technique, which has been used in RISC binary profilers like qpt [11, 2] and in decompiling of CISC code [5], is easily implemented on RISC machines due to the stylized code fragments that compilers generate for indexed jumps. In contrast, CISC code is much harder to analyse for indexed jumps.

The generation of the control flow graph removes dependencies on the hardware pipeline, by analysing delayed instructions. On SPARC [13], instructions that transfer flow of control work in combination with a delayed instruction (that follows it in the assembly code). The transfer instruction is first checked to determine if a branch is to be taken or not, and the delayed instruction is executed before the flow of control is transferred to the target destination. In this case, the delayed instruction is executed whether the branch is taken or not, hence it can be moved before the transfer of control instruction.

There are also annulled delayed instructions, which annul the execution of the delayed instruction when the false branch of the instruction is taken. In this case, the delayed instruction can only be copied to a new basic block that

falls through to the target address of the transfer of control instruction. The new basic block is needed in case other instructions jump to the target address directly, and may be removed later on in the analysis if no other basic block transfers control to it. This technique has been used in the past in binary profilers like qpt, and are explained in detail in [11]. In our framework, the removal of delayed instructions is just another machine idiom.

High Level Opcodes

Assembly mnemonic instructions can be represented by a set of five different high-level opcode instructions:

- assignment (`:=`): assigns the right-hand side expression to the left-hand side location,
- conditional (`if`): checks the value of an expression to determine which is the target address for the branch,
- unconditional (`jump`): unconditional transfer of control; equivalent to a `goto`,
- call (`call`): procedure call invocation, and
- return (`ret`): return from a procedure.

An almost one to one mapping of assembly instructions to the above high-level opcodes requires one traversal of the code. Some use-definition analysis of condition codes is required to merge compare and conditional jump instructions into `if` instructions. Note that at this stage, expressions that form part of an assignment are very simple, e.g. `%l5 = %i0 + 10`, but throughout the analysis, expressions are merged to regenerate more complex expressions and remove the register references.

It is important to point out that since no control structure recovery analysis has been performed, it is unknown at this stage whether the conditional `if` instruction is an if-then or an if-then-else structure.

For analysis purposes, definition-use (du) and use-definition (ud) chains are constructed for each register and condition code in an instruction. These chains are at the procedure level; i.e. they do not cross procedure boundaries, however, they do take into account summary information provided by the called procedure. Figure 1 shows where our sample assembler program of Figure 2 came from: a C program which implements the string length function `strlen` and uses it in its `main`. Its assembly code representation in Figure 2 is our starting point for analysis. Extraneous text lines of data have been removed from this figure and the code has been annotated with ud and du chains.

4 Interprocedural Register Copy Propagation

The technique described herein is an extension of the data flow technique used in *dcc* [6], an 80286 decompiler, for the recovery of high-level language code

```
int strlen (char *s)
{  char *t = s;
       while (*s != '\0')
               s++;
       return s-t;
}

int main (void)
{ char *s = "hello";
  int size;
       size = strlen(s);
       return 0;
}
```

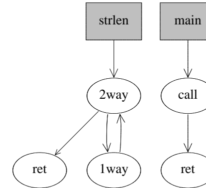

Figure 1: String length (strlen) program in C and corresponding control flow graph

```
strlen:                          LiveIn={%o0->%i0}  LiveOut={%i0->%o0}
12      save %sp,-120,%sp
13      st %i0,[%fp+68]          ud(%i0)=-1
14      ld [%fp+68],%o0          du(%o0)=15
15      st %o0,[%fp-20]          ud(%o0)=14
.LL2:
17      ld [%fp+68],%o0          du(%o0)=18
18      ldub [%o0],%o1           ud(%o0)=17 du(%o1)=19
19      sll %o1,24,%o2           ud(%o1)=18 du(%o2)=20
20      sra %o2,24,%o0           ud(%o2)=19 du(%o0)=21
21      cmp %o0,0                ud(%o0)=20 du(cc)=22
22      bne .LL4                 ud(cc)=21
24      b .LL3
.LL4:
27      ld [%fp+68],%o1          du(%o1)=28
28      add %o1,1,%o0            ud(%o1)=27 du(%o0)=29
29      mov %o0,%o1              ud(%o0)=28 du(%o1)=30
30      st %o1,[%fp+68]          ud(%o1)=29
31      b .LL2
.LL3:
34      ld [%fp+68],%o0          du(%o0)=36
35      ld [%fp-20],%o1          du(%o1)=36
36      sub %o0,%o1,%o0          ud(%o0)=34 ud(%o1)=35 du(%o0)=37
37      mov %o0,%i0              ud(%o0)=36
38      b .LL1
.LL1:
41      ret
main:                            LiveIn={}  LiveOut={}
58      sethi %hi(.LLC0),%o1
59      mov .LLC0,%o0            du(%o0)=60
60      st %o0,[%fp-20]          ud(%o0)=59
61      ld [%fp-20],%o0          du(%o0)=62
62      call strlen,0            ud(%o0)=61 du(%o0)=64
64      st %o0,[%fp-24]          ud(%o0)=62
65      mov 0,%i0
66      b .LL5
.LL5:
69      ret
70      restore
```

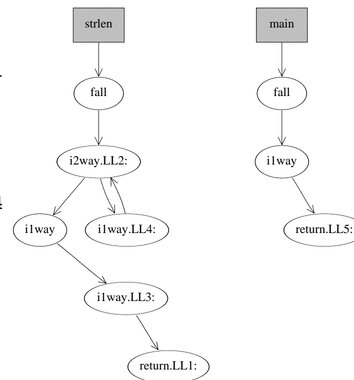

Figure 2: Unoptimized SPARC assembly code for strlen program, annotated with definition-use and use-definition chains

from DOS executables. The aim of this article is to test the suitability and generality of such a technique for the recovery of RISC code, particularly in the context of large programs and optimized code.

Throughout the analysis we treat registers and condition codes in the same way—as registers. Each condition code (cc) is treated as a different register. This is needed as instructions may set n condition codes and use m condition codes. However, for the purposes of illustration in Figure 2, all condition codes have been grouped into one as the use of m overlaps with the definition of n (for the sample instructions).

A definition of a register r at instruction i in terms of a set of a_k registers, $r = f_1(\{a_k\}, i)$, can be forward substituted at the use of that register on another instruction j, $s = f_2(\{r, \ldots\}, j)$, if the definition at i is the unique definition of r that reaches j along all paths in the program, and no register a_k has been redefined along that path. The resulting instruction at j would then look as follows:

$$s = f_2(\{f_1(\{a_k\}, i), \ldots\}, j).$$

This relationship is partly captured by the definition-use (du) and use-definition (ud) chains of an instruction: a use of a register is uniquely defined if it is only reached by one instruction, that is, its ud chain set has only one element. This relationship is known as the $r\text{-clear}_{i \to j}$ relationship. More formally,

$$s = f_2(\{f_1(\{a_k\}, i), \ldots\}, j) \text{ iff } \begin{array}{l} |ud(r,j)| = 1 \land \\ ud(r,j) = i \land \\ j \in du(r,i) \land \\ \forall a_k \bullet a_k\text{-clear}_{i \to j} \end{array}$$

Note that this definition does not place a restriction on the number of uses of the definition of r at i. Hence, if the number of elements on $du(r, i)$ is n, this instruction can potentially be substituted into n different instructions j_k, provided they satisfy the $r\text{-clear}_{i \to j_k}$ property.

We now look at the application of this relationship to the different parts of the assembly example in Figure 2, namely, elimination of condition codes, elimination of intermediate registers, recreation of high-level expressions, recovery of formal and actual parameters, and determining function return values.

Elimination of Condition Codes

On SPARC, condition codes are set when a later instruction will use them. Hence, for a use of a condition code or sets of condition codes, there is normally only one instruction which defines them, making it simple to eliminate them. From Figure 2:

```
21 cmp %o0,0      ud(%o0)=20 du(cc)=22
22 bne .LL4       ud(cc)=21
```

we can see that all condition codes used by the branch on not equal (instruction 22) are defined by the compare instruction at 21. These two instructions satisfy the $\text{cc-clear}_{21 \to 22}$ relationship, and hence can be merged. Prior to merging we need to capture the semantic meaning of those instructions in a logical expression; in this case,

```
22 if (%o0 != 0) jump .LL4
```

and instruction 21 is removed (as it's no longer needed).

Elimination of Intermediate Registers

On SPARC, all instructions operate on registers only, except for the load (`ld`) and store (`st`) instructions, which load a value from memory onto a register, or store a value of a register to memory, respectively. In order to perform an arithmetic operation, the values of variables are copied to registers, the operation(s) is performed, and the value of the result is copied back to the variable in memory—these registers are what we refer to as "intermediate registers". Intermediate registers can always be eliminated and the instructions involved can be replaced by a HLL expression. From Figure 2, rewritten in the form of assignments:

```
27  %o1 := [%fp+68]       du(%o1)=28
28  %o0 := %o1 + 1        ud(%o1)=27 du(%o0)=29
29  %o1 := %o0            ud(%o0)=28 du(%o1)=30
30  [%fp+68] := %o1       ud(%o1)=29
```

We replace the right-hand side of a register definition on the use of that register at a later instruction. Hence, `[%fp+68]` is replaced in the place of `%o1` at instruction 28, leading to:

```
28  %o0 := [%fp+68] + 1
```

and removing instruction 27 as its use is unique. This new instruction is replaced at instruction 29 replacing `%o0`, leading to

```
29  %o1 := [%fp+68]+1
```

We then replace `%o1` at instruction 30, leading to the final high-level instruction:

```
30  [%fp+68] := [%fp+68] + 1
```

and eliminating the other 3 instructions in this process. This process was possible as all instructions satisfied the $r\text{-clear}_{i \to j}$ condition.

Subroutine Signature

The signature or declaration of a subroutine is composed of its name, formal parameter list, and return value (if any). The latter two pieces of information are recovered from interprocedural live register analysis: a register that is used before definition in a subroutine must be defined by it's caller. A register that is used after a procedure call and has been set by the subroutine called, is a register that must be returned by the subroutine. This information is captured in the *LiveIn and* LiveOut sets of each routine in the program. From Figure 2:

```
strlen:   LiveIn={%o0->%i0}  LiveOut={%i0->%o0}
```

which not only provides the register information needed but also the mapping of registers across register-window calls on SPARC. Within `strlen`, register `%i0` is a formal parameter, but it's caller would have had to set `%o0` (i.e. output registers become input registers). Within `main`, after the call to `strlen`, register `%o0` is used. This register was set within `strlen` as register `%i0`. We summarize this information as follows:

```
%i0 strlen (%i0)
```

Actual Parameters

Actual parameters to a procedure call are registers that are live on entry to the called routine (i.e. LiveIn registers). We annotate call instructions with a ud-chain on all registers that are LiveIn in the called routine. From Figure 2:

```
62  call strlen,0            ud(%o0)=61
```

as `LiveIn(strlen)={%o0->%o1}`, and the call instruction is now viewed as:

```
62  call strlen,0 (%o0)      ud(%o0)=61
```

We can now substitute the definition of `%o0` in this instruction. From Figure 2:

```
61  ld [%fp-20],%o0          du(%o0)=62
62  call strlen,0 (%o0)      ud(%o0)=61 du(%o0)=64
```

which gives the high-level code:

```
62  call strlen,0 ([%fp-20])
```

i.e., a local variable is used as the actual argument.

Function Return

In a similar way, a subroutine that returns a value (i.e. LiveOut $\neq \{\}$) is viewed as an assignment to the register(s) that it defines. From Figure 2:

```
62  %o0 = call strlen,0 (%o0)
```

and its du-chain on `%o0` would be calculated: du(%o0)=64. Substituting this instruction on instruction 64 gives:

```
64  [%fp-24] := call strlen,0 (%o0)
```

that is, the result of the call is stored in local variable `[%fp-24]`.

Applying this extended register copy propagation technique leads to the pseudo-C code on the left-hand side of Figure 3. In this figure, local variables have been given the names `loc1` and `loc2` as the names were not available from the assembly code. A few registers remain, these are of three types: registers in the formal parameter list (which can be removed by naming the formal parameters), registers that are redundant (i.e. not used and hence can be eliminated via dead register analysis), and registers that cannot be eliminated and are equivalent to register-variables (e.g. `i0:=loc1-loc2` in `strlen`), hence they are replaced by new local variables in the procedure.

Analysis of the flow of control of the program in Figure 3 and simple transformations on instructions mentioned above would lead to the right-hand side code of Figure 3, which is comparable to the initial C code of Figure 1.

We applied the same techniques to optimized level O2 code of the `strlen` program. The optimized assembly code is shown in Figure 4, annotated with ud and du chains, and the CFG of each routine. The generated code after analysis is in the left-hand side of Figure 5. Control flow analysis and variable renaming would lead to the right-hand side code. It is clear from this code that although it is longer and different to the one generated for the unoptimized case, both programs are functionally equivalent to the initial code of Figure 1. Further, this code is more efficient as the `while` loop is only executed when needed.

```
%i0 strlen(%i0 )
{
    loc1 := %i0
    loc2 := loc1
.LL2:
    if ( (([loc1] << 24) >>a 24) != 0 ) jump .LL4
    jump .LL3
.LL4:
    loc1 := loc1 + 1
    jump .LL2
.LL3:
    %i0 := loc1 - loc2
    jump .LL1
.LL1:
    ret
}

main( )
{
    %o1 := %hi(.LLC0)
    loc1 := .LLC0
    loc2 := strlen (loc1)
    %i0 := 0
    jump .LL5
.LL5:
    ret
}
```

```
strlen (arg0)
{
    loc1 := arg0
    loc2 := loc1
    while (*loc1 != 0)
        loc1 := loc1 + 1
    res0 := loc1 - loc2
    ret (res0)
}

main()
{
    loc1 := &strLabel
    loc2 := strlen(loc1)
    ret
}
```

Figure 3: Generated pseudo-C code after analysis for unoptimized assembly code (left-hand side) and after control flow analysis (right-hand side)

4.1 Sample Generated Code

We show snippets of two other programs to illustrate the usefulness of the technique, and the need for type recovery of compound types.

The left-hand side of Figure 6 shows the pseudo-C code generated for the recursive fibonacci routine, which was compiled with O2 optimization level. The right-hand side shows what the code would look like after control flow analysis. In this case, due to the base types used by the routine (only integers) being the same as those available in the machine (word), the generated code would trivially annotate the types of the variables as integers.

The left-hand side of Figure 7 shows the snippet of generated code for the matrix multiplication C statement:

```
A[i,j] = A[i,j] + B[i,k] * C[k,j]
```

and the right-hand side shows what each of the individual statements are equivalent to if they were to be transformed into array notation.

In this case, the `.umul` function is not part of the assembly source code, hence, our algorithm cannot calculate LiveIn or LiveOut sets for this routine. In these cases, we perform a conservative analysis and do not forward substitute across those procedure calls, leading to the uncertainty of `loc2` after the procedure code—we cannot assert if it has been changed or not. In our example, we know for a fact that it is changed, as the `.umul` function takes two arguments and returns the multiplication of their values. This analysis can be

```
strlen:                         LiveIn={%o0}  LiveOut={}
12      ldsb [%o0],%g2          du(%g2)=13
13      cmp %g2,0               ud(%g2)=12 du(cc)=14
14      be _delayed_l14c1       ud(cc)=13
        ba _delayed_l14c2
_delayed_l14c1:
        mov %o0,%g3             du(%g3)=24
        ba .LL3
_delayed_l14c2:
15      mov %o0,%g3             du(%g3)=24
16      add %o0,1,%o0           du(%o0)=18 du(%o0)=21 du(%o0)=24
.LL6:
18      ldsb [%o0],%g2          ud(%o0)=16 ud(%o0)=21 du(%g2)=19
19      cmp %g2,0               ud(%g2)=18 du(cc)=20
20      bne_a _annulled_l20c3   ud(cc)=19
        ba _annuled_l20c4
_annulled_l20c3:
21      add %o0,1,%o0           ud(%o0)=16 ud(%o0)=21
                                du(%o0)=18 du(%o0)=21 du(%o0)=24
        ba .LL6
_annuled_l20c4:
.LL3:
24      sub %o0,%g3,%o0         ud(%o0)=16 ud(%o0)=21
                                ud(%g3)=-1 ud(%g3)=15
23      retl
main:                           LiveIn={}  LiveOut={}
40      sethi %hi(.LLC0),%o0
42      mov .LLC0,%o0           du(%o0)=41
41      call strlen,0           ud(%o0)=42
43      ret
```

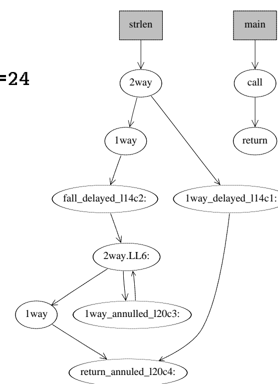

Figure 4: Optimized level O2 SPARC assembly code for strlen program, annotated with definition-use and use-definition chains

done in binary executables as the routine `.umul` would have been either statically linked in, in which case the code is in the file, or dynamically linked in, in which case the code is in a dynamic library pointed to by the binary executable file. Nevertheless, the example illustrates the need for the third type of analysis in the recovery of HLL code from assembly: type recovery. Such analysis may use lattices or type theory to attempt to recover high-level language types such as arrays and structures.

5 Results

In order to test the register copy propagation method, we tested *asm2c* against the integer SPEC95 benchmark programs, and gathered statistical data on the number of assembly instructions that were transformed into high-level instructions. The statistical information aids in comparing the complexity of recovery of high-level language code, as explained in further paragraphs.

The benchmarks that were used for testing purposes were:

- go: artificial intelligence; plays the game of "Go"
- m88ksim: moto 88K chip simulator; runs test program
- gcc: GNU C compiler; builds SPARC code

```
%o0 strlen(%o0 )
{
    if ( [%o0] == 0 )
         jump _delayed_l14c1
    jump _delayed_l14c2
_delayed_l14c1:
    %g3 := %o0
    jump .LL3
_delayed_l14c2:
    %g3 := %o0
    %o0 := %o0 + 1
.LL6:
    if ( [%o0] != 0 )
         jump _annulled_l20c3
    jump _annulled_l20c4
_annulled_l20c3:
    %o0 := %o0 + 1
    jump .LL6
_annulled_l20c4:
.LL3:
    %o0 := %o0 - %g3
    ret
}

main( )
{
    %o0 := %hi(.LLC0)
    %o0 := strlen (.LLC0)
    ret
}
```

```
strlen (arg0)
{
    if (*arg0 == 0)
        glb1 := arg0
    else
    {
        glb1 := arg0
        arg1 := arg0 + 1
        while (*arg0 != 0)
            arg0 := arg0 + 1
    }
    arg0 := arg0 - glb1
    ret (arg0)
}

main ()
{
    loc1 := strlen (&strLabel)
    ret
}
```

Figure 5: Generated pseudo-C code from optimized assembly code (left-hand side) and after control flow analysis (right-hand side)

```
%i0 fib(%i0 )
{
    %l0 := %i0
    if ( %i0 <= 2 ) jump _delayed_l15c1
    jump _delayed_l15c2
_delayed_l15c1:
    %i0 := 1
    jump .LL4
_delayed_l15c2:
    %i0 := 1
    %i0 := fib (%l0 - 1) + fib (%l0 - 2)
.LL4:
    ret
}
```

```
fib (arg0)
{
    loc1 := arg0
    if (arg0 <= 2)
        arg0 := 1
    else
        arg0 := fib(loc1-1)
              + fib(loc1-2)
    ret (arg0)
}
```

Figure 6: Generated pseudo-C code for the recursive fibonacci function from optimized code (left-hand side), and after control flow analysis (right-hand side)

- compress: compresses and decompresses file in memory
- li: LISP interpreter
- ijpeg: graphic compression and decompression
- perl: manipulates strings (anagrams) and prime numbers in Perl.

```
%l0 := (loc6 << 2) + [(loc5 << 2) + loc2]        l0 := &A[i,j]
%l1 := (loc6 << 2) + [(loc5 << 2) + loc2]        l1 := &A[i,j]
%o0 := [(loc7 << 2) + [(loc5 << 2) + loc3]]      l2 := B[i,k]
%o1 := [(loc6 << 2) + [(loc7 << 2) + loc4]]      l3 := C[k,j]
.umul ()                                         .umul (l2,l3)
[%l0] := [%l1] + %o0                             A[i,j] := A[i,j] + l2
```

Figure 7: Generated pseudo-C code for matrix multiplication statement from unoptimized code

Benchmark	Unoptimized Assembly LOC	Optimized Assembly LOC
go	108995	55035
m88ksim	41307	22361
gcc	384546	153094
compress	1363	-
li	16697	10250
ijpeg	56440	28691
perl	58757	35885

Figure 8: Number of assembly lines processed by the asm2c tool

These programs were compiled with gcc on a SPARC V9 machine using the -S option to produce assembly code. Two compilations to assembly were made, one without optimization, and another with level O2 optimization. Figure 8 shows the number of assembly lines for each benchmark program, for both, unoptimized and optimized cases.

Figure 9 shows the results for the unoptimized version of the benchmarks. For each program, the first column shows the original number of assembly instructions, and the second column shows the generated number of high-level language statements. The average reduction rate was of 66.63%, which clearly shows the code explosion generated by the compiler when emitting assembly code without optimizations. These results are inline with results on a CISC machine, 80286, were a reduction in the number of instructions was 70% [3], and with results achived by the text compression method on MIXAL code; 40% [10].

Figure 10 shows the results for the level O2 optimization case. It is clear from the figures that the final difference in the number of instructions was small; with only a 5.58% reduction rate. This small difference on its own shows how well the compiler is doing at generating assembly code and that there is an almost 1:1 correlation between the average number of assembly instructions and high-level instructions. Checking the generated high-level code, it is clear that extra analysis is needed to improve the quality of the recovered code. For example, a loop that was optimized by loop unrolling, will be recovered as a loop with duplicated code throughout it; checking for duplicated code and then folding the code would be an extension to the analysis that will improve the quality of the generated code. Nevertheless, the generated code is of a high-level of abstraction and does not resemble assembler. Once these results are combined with the recovery of control structures in the program, the final program would be more readable and would contain less number of statements,

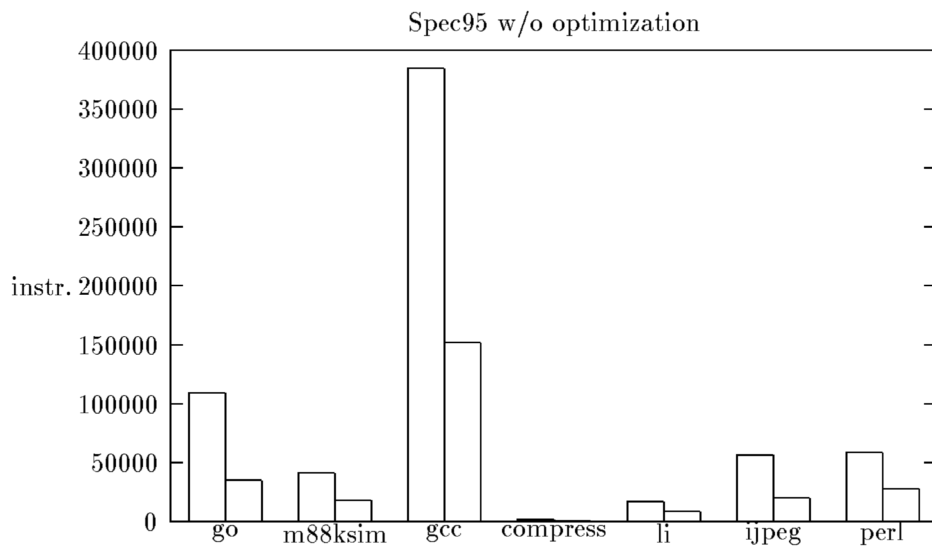

Figure 9: Comparison results for unoptimized assembly code

as at present they contain a percentage of redundant jumps. Figures based on the comparison of generated HLL statements (including control constructs such as loops and if-then-else's) cannot be created until the data flow analysis technique has been integrated with the control flow analysis technique; such results will be reported in the near future.

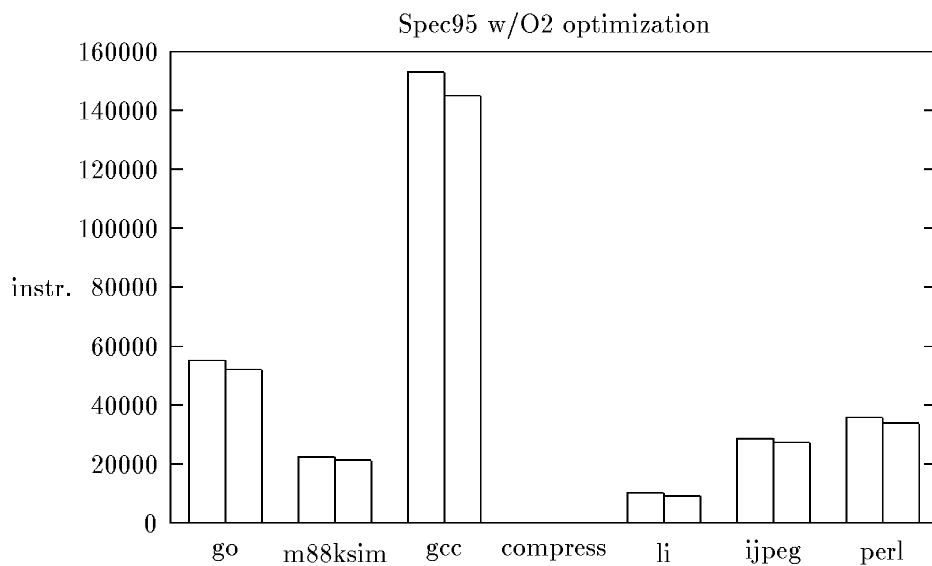

Figure 10: Results for level O2 optimized assembly code

6 Summary and Conclusions

We have shown the application of the extended register copy propagation technique to SPARC (RISC) assembly code in order to recover high-level language code. The aim of the technique is to remove all low-level dependencies on the machine, such as registers, stacks and condition codes, and recover high-level abstractions such as expressions, parameters, and function calls. The technique is an extension of the technique used in *dcc*, an 80286 (CISC) decompiler for DOS programs.

The results show that on both types of machines, the reduction on the number of assembly to high-level language instructions is quite large; 70% for *dcc* [6] and 66% for *asm2c*. This rate shows the code explosion when translating high-level language code to assembly. However, once optimized assembly code is analyzed, the reduction is quite small (5.58%). This is mainly due to the fact that the size of optimized assembly code is dependent on the types of optimizations performed (e.g. loop unrolling, inlining, global code scheduling) by the compiler. Nevertheless, the results show that extended register copy propagation itself can recover readable, high-level code that doesn't resemble assembly code. Once this technique is combined with the recovery of control structures, the final code resembles high-level language code.

Future Work

We are developing techniques for the recovery of high-level language code from assembly code for RISC and CISC machines. Two main areas have been tackled so far: recovery of control structures and recovery of high-level language statements. The next step is to recover data types associated to each of the statements in the program, hence making the generated code better from the user's perspective.

References

[1] W.S. Adolph. Cash cow in the tar pit: Reengineering a legacy system. *IEEE Software*, 13(3):41–47, May 1996.

[2] T. Ball and J.R. Larus. Optimally profiling and tracing programs. *Transactions of Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[3] C. Cifuentes. Interprocedural dataflow decompilation. *Journal of Programming Languages*, 4(2):77–99, June 1996.

[4] C. Cifuentes. Structuring decompiled graphs. In T. Gyimóthy, editor, *Proceedings of the International Conference on Compiler Construction*, Lecture Notes in Computer Science 1060, pages 91–105, Linköping, Sweden, 24-26 April 1996. Springer Verlag.

[5] C. Cifuentes and A. Fraboulet. Intraprocedural slicing of binary executables. In M.J. Harrold and G. Visaggio, editors, *Proceedings of the International Conference on Software Maintenance*, pages 188–195, Bari, Italy, 1–3 October 1997. IEEE-CS Press.

[6] C. Cifuentes and K.J. Gough. Decompilation of binary programs. *Software – Practice and Experience*, 25(7):811–829, July 1995.

[7] D.M. Dejean and G.W. Zobrist. A definition optimization technique used in a code translation algorithm. *Communications of the ACM*, 32(1):94–104, January 1989.

[8] M.S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, Inc, 52 Vanderbilt Avenue, New York, New York 10017, 1977.

[9] G.L. Hopwood. *Decompilation*. PhD dissertation, University of California, Irvine, Computer Science, 1978.

[10] B.C. Housel. *A Study of Decompiling Machine Languages into High-Level Machine Independent Languages*. PhD dissertation, Purdue University, Computer Science, August 1973.

[11] J.R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software – Practice and Experience*, 24(2):197–218, February 1994.

[12] T.M. Pigoski and L.E. Nelson. Software maintenance metrics: A case study. In H.A. Muller and M. Georges, editors, *Proceedings International Conference on Software Maintenance*, pages 392–401, Victoria, British Columbia, Canada, September 1994. IEEE Computer Society Press.

[13] Sparc. *The SPARC Architecture Manual – Version 8*. Sparc International, Menlo Park, California, 1992.

[14] D. Ungar and R.B. Smith. SELF: The power of simplicity. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 227–241. ACM Press, October 1987.

[15] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.