Assembly to High-Level Language Translation

Assembly to High-Level Language Translation[*]

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

Technical Report 439
August 1998

Abstract

Translation of assembly code to high-level language code is of importance in the maintenance of legacy code, as well as in the areas of program understanding, porting, and recovery of code.

We present techniques used in the *asm2c* translator, a SPARC assembly to C translator. The techniques involve data and control flow analyses. The data flow analysis eliminates machine dependencies from the assembly code and recovers high-level language expressions. The control flow analysis recovers control structure statements. Simple data type recovery is also done.

The presented techniques are extensions and improvements on previously developed CISC techniques. The choice of intermediate representation allows for both RISC and CISC assembly code to be supported by the analyses.

We tested *asm2c* against SPEC95 SPARC assembly programs generated by a C compiler. Results using both unoptimized and optimized assembly code are presented.

1 Introduction

Recovery of high-level language code from assembly and machine code is an area of research that has not been widely researched in recent years, partly

[*]This work was sponsored by The University of Queensland under grant No.UQNSRG009G and Sun Microsystems Laboratories.

due to the complexity of the problem as well as the lack of techniques available in this field. In the context of fixing the year 2000 bug, the Gartner Group estimated that many organizations are missing 3% to 5% of their source code portfolios. In some cases, as much as 30% of source code is missing [21]. This means that a medium-sized information systems organization with a software portfolio of 30 to 50 million lines of code could easily be missing a million lines or more [9]. Further, some large organisations have thousands of lines of code written in assembly code and cannot benefit from state of the art object-oriented technicques unless the assembly code is reengineered into some high-level language.

These facts point to the need for research into ways of translating machine and assembly code to high-level language code, based on sound compiler technology. In essence, the problem becomes one of program comprehension; being able to understand existing code generated by assembler and compiler tools, and to apply techniques which will "undo" what the assembler/compiler has done. This transformation converts low-level assembly code into a higher level of abstraction that resembles today's high-level language (HLL) programs. Several techniques used in industry involve proprietary techniques that have not been published (e.g. Source Recovery's IBM 360 decompiler [9]). In this paper, we concentrate on the translation from SPARC assembly code to an imperative procedural language like C. Other target languages can benefit from the same techniques; we chose C due to its popularity in the Unix community. The reader is referred to [19, 8] for ways of translating machine code to assembly.

Translation of assembly to HLL code requires a series of analyses in order to abstract away from the hardware features of assembly languages, and to recover the high-level features available in most common procedural programming languages. The aim of this process is to recover HLL code which is comparable to that produced by native programmers rather than assembly code written in C which emulates features of the machine such as registers and the stack. The main types of analyses are:

- *data flow analysis* to recover high-level language expressions and statements (other than control transfer statements), actual parameters, and function return values, and to remove hardware references from the code, such as registers, pipeline references and stack references;

- *control flow analysis* to recover control flow structure information, such as loops and conditional statements, as well as their nesting level; and

- *type analysis* to recover high-level type information for variables, formal and actual parameter types, and function return types.

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.

Recovery of the control structure in a program is based on graph theory and *control flow analysis*. In assembly code, control transfer is expressed in the form of conditional or unconditional jumps, returns and procedure calls. Potentially, all jumps (conditional and unconditional) can be implemented by `goto` statements in a HLL program. However, control flow analysis can determine whether such jumps induce a loop, are redundant jumps, or will require `goto` statements. The aim is to minimize the number of `goto` statements used throughout the code without increasing the complexity of the program, in order to improve the quality of the generated code.

Type analysis deals with the recovery of HLL type information. This is an area that has traditionally been studied in the literature in the context of functional and untyped object-oriented languages. In such languages, the type of a variable is inferred based on the context of the use of the variable (e.g. Self [22]). 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 that makes use not only of base data types (e.g. byte, char, integer, and real) but also compound types (e.g. array, structure or record, and class).

In this paper we concentrate on the *data flow* and *control flow* analysis techniques as applied to RISC assembly code. Simple type analysis is performed implicitly in the algorithms used. We express properties of the techniques used, as well as the intermediate representation used in *asm2c*. We tested the techniques using a large set of assembly programs created by compiling the SPEC95 benchmarks to assembly code. The rest of this paper is structured in the following way: §2 explains the intermediate representation used in this analysis, §3 summarizes the properties used by the techniques to handle RISC assembly code, §4 shows sample code obtained from these techniques on both optimized and unoptimized code, §5 shows results obtained with the assembly version of the SPEC95 benchmarks and §6 discusses previous work in the area. Finally, §7 provides conclusions on this work.

2 Intermediate representation

Assembly programs are parsed and stored in an intermediate representation suitable for HLL recovery analysis. There are two levels to the representation in order to perform different types of analyses: a control flow graph for each procedure, and a high-level instruction for a sequence of assembly instructions. During the parsing of assembly code, machine idioms are checked for, in order to remove particularities of the machine itself. In the case of SPARC, most of the idioms relate to the synthetic instructions described in the SPARC manual [20]; 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 with a compare instruction (as register `%g0` is hardwired to zero).

2.1 Control flow graph

The control flow graph (CFG) of a procedure stores information about its instructions in 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 instruction.

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, it is not always possible to determine the target address(es) of the branch using static analysis. Two techniques are commonly used to solve this problem: intraprocedural (backwards) slicing [7] and pattern matching. The former technique is performed on the register branch, in order to obtain the set of instructions the indexed branch depends on. If the register can take a series of different values, these values are determined by a forward walk of the sliced instructions, making it possible to determine the size of the indexed table and the offsets to search for. On the other hand, the pattern matching technique requires patterns to be determined (for example, from different ways compilers generate `switch` statements), and matching to be performed against those patterns. In the cases were an indexed 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) for that procedure. The CFG generation technique, which has been used in RISC binary profilers like qpt [14] and in decompiling CISC code [7], 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 CFG needs to remove dependencies on the hardware pipeline by analysing delayed instructions. On SPARC [20], instructions that transfer flow of control work in combination with the next instruction; the delayed instruction. The delayed instruction may or may not be executed based on the type of branch. If enabled, it is executed prior to the target of the branch being reached. This semantic behaviour is documented in the SPARC manual [20], and needs to be removed from the graph in order to build a machine-independent graph. In our framework, the removal of delayed instructions used to be dealt with by using pattern matching and idioms. However, we now support a transformational approach which is proved to produce the right code for all combinations of delayed instructions [17].

2.2 High level opcodes

Assembly mnemonic instructions used in applications programs 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 (`jcond`): 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 instructions requires one traversal of the code. Some use-definition analysis of condition codes is required to merge compare and conditional jump instructions into `jcond` instructions. Note that at this stage, expressions that form part of an assignment are very simple and machine-dependent (as they use registers), e.g. `%l5 = %i0 + 10`. Throughout the

```
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;
}
```

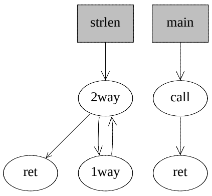

Figure 1: String length (strlen) program in C and corresponding control flow graph

analysis, expressions of several instructions are merged to regenerate more complex expressions and remove register references.

It is important to point out that since no control structure recovery analysis has been performed at this stage, it is unknown whether the conditional `jcond` instruction is the header of an `if..then`, an `if..then..else`, or a `while` 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 a C program which implements the string length function `strlen` and uses it in its `main`. The corresponding, unoptimized, assembly code is shown in Figure 2. In Figure 2, extraneous text lines of data have been removed and the code has been annotated with ud and du chains.

3 Data and control flow analysis techniques

The techniques described in this section are an extension of the data and control flow techniques used in *dcc* [8], an 80286 CISC decompiler for the recovery of C code from DOS binary executables. In this paper, we extend the techniques to deal with RISC code by stating the properties underlying the techniques, which lead to improvements on the algorithms used. The testing of these techniques is done in the context of large programs and optimized code; the CISC techniques had only been tested with small, unoptimized code, and hence were not exposed to very large unstructured

```
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
```

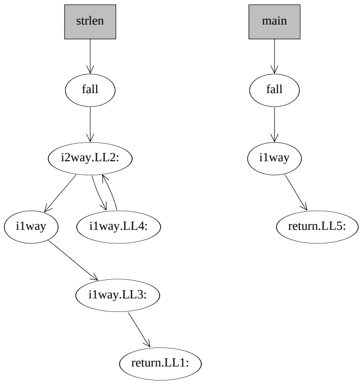

Figure 2: Unoptimized SPARC assembly code for strlen program, annotated with definition-use and use-definition chains. The CFGs for strlen and main are also shown.

examples which make the restructuring process harder to achieve.

3.1 Data flow analysis

High-level language statements can be translated to a series of assembly load and store instructions that make use of the machine's registers. Registers are normally used as carriers of intermediate values to calculate a new result or transfer it to a new memory location. To reverse this process, forward substitution on registers is used.

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:

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

and the need for the instruction at i would disappear. The previous relationship is partly captured by the du and 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-clear$_{i \rightarrow j}$ relationship for register r. More formally,

$$\begin{aligned} s = f_2(\{f_1(\{a_k\}, i), \ldots\}, j) \text{ iff } \quad & |ud(r,j)| = 1 \wedge \\ & ud(r,j) = i \wedge \\ & j \in du(r,i) \wedge \\ & \forall a_k \bullet a_k\text{-clear}_{i \rightarrow j} \end{aligned}$$

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, instruction i can potentially be substituted into n different instructions j_k, provided they satisfy the r-clear$_{i \rightarrow j_k}$ property.

Throughout the analysis we treat registers and condition codes in the same way—as registers. Each condition code 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 example.

Forward substitution analysis uses the r-clear$_{i \rightarrow j_k}$ relationship in an intra and interprocedural way. Hence, it is able to eliminate condition codes and intermediate registers, and places actual arguments into the procedure call argument lists, as well as determine the return value of functions. This can be achieved by noting that the high-level instructions use or define registers or condition codes in the following way:

- assignment: defines registers or condition codes on the left-hand side, and uses registers or condition codes on the right-hand side,
- conditional: uses condition codes and registers,
- call: uses registers passed as actual arguments, and defines returned registers.

In the case of register-windowed architectures like SPARC, the summarized LiveIn and LiveOut information for a procedure needs to make the mapping of callee registers to caller registers; as seen in Figure 2 for the `strlen` function. A step by step example of the use of forward substitution for Figure 2 is given in [6].

3.2 Control flow analysis

Transfers of control are modelled in assembly code via conditional jumps (based on the value of one or more condition codes), and unconditional jumps (i.e. `goto`'s). Forward and backward jumps denote traditional 2- or n-way conditionals, and loops respectively. The control flow graph of a procedure, $CFG = (N, E)$, captures the transfers of control relationships between basic blocks in the program, hence it is used as the basis for the analysis. The process involves the determination of conditionals and loops, followed by the restructuring of conditionals.

The node that contains the control transfer instruction is called the *header* node. The *follow* node is the node that contains the first instruction to be executed after the equivalent HLL control structure has been executed. If h is the header node and f the follow, we denote conditionals by $h\Diamond f$. Loops are denoted by the header h and latching node l as $h \circlearrowleft l$. A latching node is the "last" node in the loop; the one with the back-edge to the header node.

Structuring conditionals

For a subgraph headed by a 2- or n-way conditional node, there exists a unique node such that all paths from each of the out-edges from the header converge upon it. We define the follow node of a conditional to be that convergence node, and point out that the header's immediate post-dominator satisfies this property.

The set of nodes on all paths starting from one successor of a conditional header is called a *branch*. More formally:

$\forall s \in (\mathrm{succ}(h) \backslash \{f\})$, branch of $h \Diamond f$ is the set $\{n \in N \mid n$ is on a path $s \rightarrow f \wedge n \notin \{h, f\}\}$.

This definition implies that a branch may be empty.

If a subgraph represents a *structured* conditional $h \Diamond f$, it satisfies the following two properties:

1. $\forall b_i \in$ *branches of* $h \Diamond f \bullet \bigcap_i (b_i) = \emptyset$, and
2. $\forall b_i \in$ *branches of* $h \Diamond f \bullet (\nexists n \in \bigcup_i (b_i) \bullet (m, n) \in E \wedge m \notin (\{h\} \cup \bigcup_i (b_i)))$.

The first property states that the membership of the branches must be mutually exclusive (i.e no node may belong to more than one branch of a given conditional). The second property states that the predecessor of any node within a branch must be either another node within the same branch or the conditional header. An *unstructured* conditional will violate at least one of these properties.

The type of a conditional is determined by its header node: an n-way node is equivalent to the start of a `switch` or `case` statement, and a 2-way node can be one of the following statements:

- `if..then..else` - neither successor of the header is the follow node or at least one successor is reached by a back edge.
- `if..then` - the false successor is the follow node.
- `if..else` - the true successor is the follow node.

Structuring loops

An ordering between the nodes in a CFG is given by a post-order walk of the graph during a depth-first search traversal. This ordering implies that leaf nodes in the underlying depth-first search tree (DFST) will have a smaller order than nodes higher up in the tree, and that the header of the CFG will have an order equal to the number of nodes within the graph. Once this partial-order has been established, a loop $h \circlearrowright l$ is determined by a back-edge from a node lower in the underlying DFST to a higher node in the tree; i.e. $\mathrm{order}(h) > \mathrm{order}\ (l)$ and the edge $(l, h) \in E$.

In our work, we have used interval theory [2] to determine the nesting level of loops and the nodes that belong to the loop. However, different methods can be used for this purpose, for example [3, 1, 10]. Using intervals, the nodes that belong to a loop can be determined by the following set:

$\mathrm{loopNodes}(h \circlearrowright l) = \{n \in N \bullet \mathrm{order}(l) \leqslant \mathrm{order}(n) < \mathrm{order}(h) \wedge n \in I(h)\}$

where $I(h)$ denotes the interval that h belongs to. Nodes that belong to a loop are tagged as belonging to the most nested loop they belong to.

Finally, the type of the loop is determined by its header and latching nodes in the following way:

- A *pre-tested* loop header will be a 2-way node that controls the loop iterations and will have a latching node that has a solitary edge back to the loop header. One of the edges from a pre-tested loop header will lead to the follow node of the header when it was previously structured as a 2-way conditional.
- A *post-tested* loop may have either a 2- or 1-way node as a header node and a 2-way latching node. The loop will have a 2-way header node if the first statement within the loop is a 2-way conditional.
- An *endless* loop will have either a 2- or 1-way header node for the same reasons as a post-tested loop, but it will only have a 1-way latching node. The difference between an endless loop with a 2-way header and a pre-tested loop is whether or not the follow of the conditional is a node within the loop or not.

Note that back-edges out of an n-way conditional are not structured as loops, as that would generate too many `goto`'s in the generated code (one for each arm of an n-way conditional).

Restructuring conditionals

To improve the code generated when a 2-way conditional overlaps with a loop or n-way conditional, we make use of the nodes' n-way header tag and their loop header tag. For a 2-way conditional $h \Diamond f$ that represents an unstructured jump into or out of a loop, h's loop header tag will be different to f's loop header tag. Similarly for an unstructured jump into or out of an n-way conditional, h and f's respective n-way header tags will be different. We use this information to modify the type of a 2-way conditional and its follow, so that a simple "if condition goto label" can be emitted during code generation.

For example, to restructure a 2-way conditional $h \Diamond f$ in which one branch makes an unstructured exit from a loop, we detect such branch by testing the following two conditions:

1. h has a different loop header than f, and
2. $\exists! \, s \in \text{succ}(h) \bullet$ there is a path from s to the latch node of the loop enclosing h (s may be the latch node).

If both conditions hold, then s is determined to be the new follow for the conditional headed by h. If s is the successor of the true branch of h, then the type of the conditional is changed to be `if..else` otherwise it is set to `if..then`. This allows for the generation of a `goto` as part of that branch.

Similar rules are created for unstructured entries into a loop and n-way structures. We point out that unstructured forward exits from an n-way structure are not possible as any conditional within the n-way structure will have the same follow as the n-way node (by construction).

Lastly, 2-way nodes that have a successor s reached by a back-edge that was not structured as the latch node for a loop are restructured. This case includes that of an unstructured back-edge out of an n-way structure. In this case, we set the new follow of the 2-way to be the successor not reached by the back-edge. The type of the node is changed to be `if..then` or `if..else`, based on whether s is the successor along the true or false branch.

Concrete details of the implementations of these algorithms and examples are given in [18].

4 Sample generated code

Applying the data flow method to the example of Figure 2 leads to the pseudo-C code on the left-hand side of Figure 3. The right-hand side shows the code after control flow analysis. This code is comparable to the initial C code of Figure 1. In Figure 3, 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 named local variables in the procedure.

We applied the same techniques to optimized level O2 assembly code of the `strlen` program. The optimized assembly code is shown in Figure 4, annotated with ud and du chains, and the CFG for each routine. The generated code after data flow analysis is on the left-hand side of Figure 5, and after control flow analysis and variable renaming on the right-hand side. 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.

```
%i0 strlen(%i0 )
{
    loc1 := %i0
    loc2 := loc1
.LL2:
    jcond ((([loc1]<<24)>>a24) != 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 := &labelLLC0
    loc2 := strlen(loc1)
    ret
}
```

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

4.1 Sample snippets of generated code

We show snippets of two other programs to illustrate the usefulness of the technique, and the need for type recovery of compound data types. Basic types are trivially supported by the algorithms, and type propagation is performed wherever possible. No attempt is made to recover 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 the code after control flow analysis. In this case, due to the base types used by the program (only integers) being the same as those available in the machine (word), the generated code trivially supports 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]
```

```
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
```

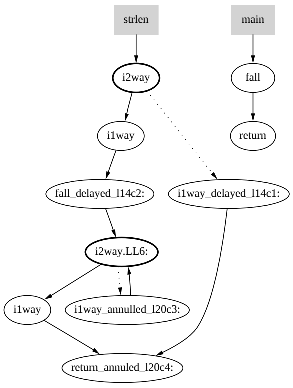

Figure 4: Optimized level O2 SPARC assembly code for strlen program, annotated with definition-use and use-definition chains

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 rou-

```
%o0 strlen(%o0 )
{
    jcond ( [%o0] == 0 )
         jump _delayed_l14c1
    jump _delayed_l14c2
_delayed_l14c1:
    %g3 := %o0
    jump .LL3
_delayed_l14c2:
    %g3 := %o0
    %o0 := %o0 + 1
.LL6:
    jcond ( [%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 (&labelLLC0)
    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
    jcond ( %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)

```
%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 ()                                          l2 := .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

tine. In these cases, we perform a conservative analysis and do not forward substitute values across those procedure calls, as we cannot assert whether the values have been changed or not. However, if the complete assembly code is available, forward substitution in this example would lead to determine that the function `.umul` takes two parameters (the contents of registers `%o0` and `%o1`) and returns a result in register `%o0`. 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 compound data types such as arrays and structures.

5 Results

We tested *asm2c* against the assembly version of the integer SPEC95 benchmark programs, and gathered statistical data on the number and type of assembly instructions that were transformed into high-level instructions. The statistical information aids in comparing the complexity of recovery of HLL 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
- compress: compresses and decompresses file in memory
- li: LISP interpreter
- ijpeg: graphic compression and decompression
- perl: manipulates strings (anagrams) and prime numbers in Perl.

The benchmark programs were compiled with gcc on a SPARC V9 machine using the -S option to produce assembly code. We compiled to assembly to have large test assembly programs as we were unable to get sample programs from industry. Two compilations to assembly were made, one without optimization, and another with optimization level O2. Figure 8 shows the number of assembly lines for each benchmark program, for both unoptimized and optimized cases.

Benchmark	O0 Assembly LOC	O2 Assembly LOC
go	108995	55035
m88ksim	41307	22361
gcc	384546	153094
li	16697	10250
ijpeg	56440	28691
perl	58757	35885

Figure 8: Number of assembly lines of code (LOC) processed by the asm2c tool.

5.1 Data flow results

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 HLL assignment and call statements, as well as simple conditionals. The average reduction rate was 66.63%, which clearly shows the code explosion generated by the compiler when emitting assembly code without optimizations. These results are comparable to results on an 80286 CISC machine, where a reduction in the number of instructions was 70% [4], and with results achieved by the text compression method on MIXAL code; 40% [13].

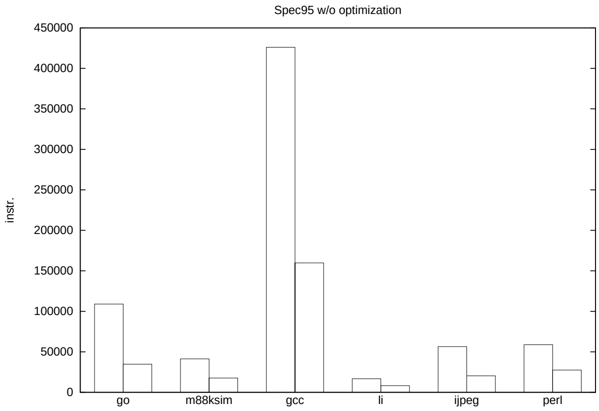

Figure 9: Comparison results for unoptimized assembly code

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 for optimized code is needed to improve the quality of the recovered code, however, the code is valid code as for generated. 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 would improve the quality of the generated code. Nevertheless, the generated code is of a high-level of abstraction and does not resemble assembly code.

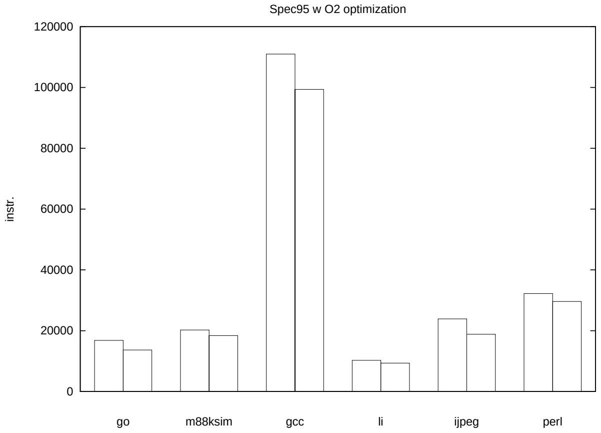

Figure 10: Results for level O2 optimized assembly code

5.2 Control flow results

We collected data for the following types of control transfer statements: 2-way conditionals, loops, n-way conditionals, and goto's. Figures 11, 12, 13, and 14 provide comparative data based on the number of a given control transfer statement; 2-way statements, loops, n-way statements and goto's respectively. 3 bars are used to represent different information about a program, in left to right order, the unoptimized regenerated C code, the optimized regenerated C code, and the original C code. The data for the regenerated C code was gathered by *asm2c* itself, and the data for original C programs was gathered with the aid of scripts.

Recovered loops are a combination of 2-way conditionals and `goto` statements. As can be seen in Figure 12, the number of loops in the original C programs were normally higher than those recovered from unoptimized and optimized assembly code. However, Figure 11 shows that the generated code contains more conditionals and Figure 14 states that the generated code contains more `goto` statements than in the original C code. These figures imply that our loop recovery algorithm can be improved in order to deal with some of the 2-way conditionals and `goto`s that can be transformed into a loop. However, these loops are the unstructured ones and design decisions will most invariably lead to different C code.

The large number of `goto` statements is also due to the fact that we are not recoverying short-circuit evaluated expressions from the assembly code. We have also noticed that redundant `goto` statements are generated in the code—these are a legacy of the labels found in the original assembly code and need to be removed by an extra pass through the code.

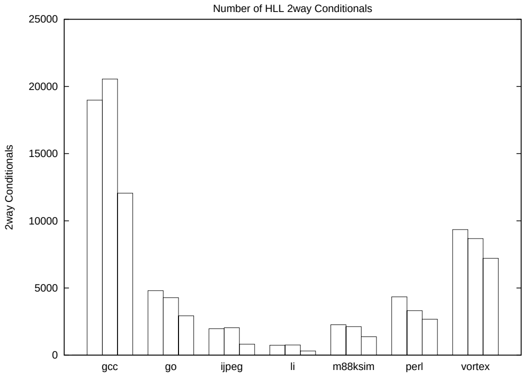

Figure 11: Comparison of number of 2-way Statements

5.3 Maintainability of the generated code

As can be seen from the examples in this section, the generated code resembles the original C code in that high-level statements are used throughout the program and no emulation of the original machine exists in such generated code. However, the lack of meaningful variable names and the lack of comments makes it hard to understand the functionality of the program. Clearly, if such names and comments do not exist in the assembly code, they cannot exist in the recovered HLL code.

Companies that provide a translation service will make use of human resources to aid in this area. As reported by Freeman [9], the Source Recovery company makes use of any supporting information that clients can provide

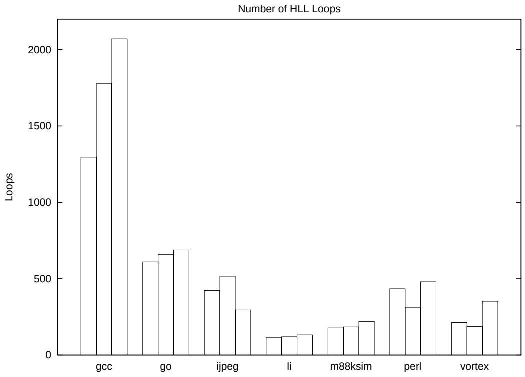

Figure 12: Comparison of number of Loop Statements

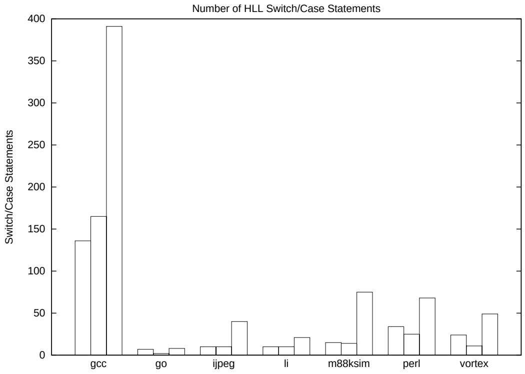

Figure 13: Comparison of number of n-way Statements

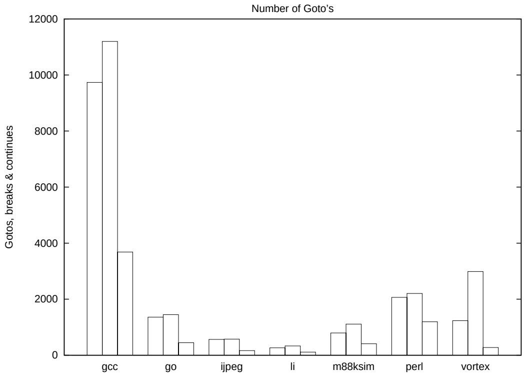

Figure 14: Comparison of number of Goto Statements

them with, such as file layouts, source of copybooks, JCL programs, documentation, and program and application specifications, to recover meaningful variable names. This clearly makes the code more maintainable and has no impact on the performance of the application.

Informal comments received from software reengineers who are familiar with both assembly and C code points to the usefulness of the translation scheme as it saves hours of hand-crafted translation time. When asked which code would they rather maintain, they pointed to the generated code as it was smaller than the assembly version and it had control structures. They thought that the generated code needs improvements in the form of data types for variables (particularly compound data types) and variable naming conventions. However, there was no general agreement as to which naming convention to use, and it was thought that changing names of variables by hand would be needed. Also, comments in the code are needed and the maintainer would need to add these. It was generally believed that an interactive tool would be helpful for making changes to the generated code, so that the tool could check the impact of changes on data types and the like. Overall, the generated code was rated useful and it was acknowledged that documentation, data types and name changes are required.

6 Previous work

Recovery of HLL 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. Housel [13] used text compression to eliminate intermediate loads and stores, and Hopwood [12] used expression condensation to combine two or more intermediate assembly-like instructions into equivalent expressions. More recently, Cifuentes [4] used forward substitution to eliminate intermediate assembly instructions from disassembled 80286 code.

In the area of control flow analysis, there have been algorithms to structure HLL code that uses `goto`'s, into a more structured form. Baker [3] restructured the control flow of Fortran programs to remove `goto`'s. Lichtblau [15] made use of graph transformations to restructure the flow of control of a program. More recently, Cifuentes [5] recovered the underlying control structures of disassembled assembly code with general structuring algorithms. Proebsting [16] recovered the underlying control structured of Java bytecode programs by using pattern matching techniques.

A few commercial assembler to C or COBOL translation tools are available in the market. For example, ASM370C Translator by Micro-Processor Services translates mainframe assembler 360 or 370 into C, and XTRAN by Pennington Systems translates PDP 11 assembler to C. More recently, the Source Recovery company offers machine and assembly code translations to COBOL for IBM 360 machines [11]. Pattern matching techniques are used to recover COBOL code based on the code that the compiler generates.

7 Summary and conclusions

We have presented the properties underlying algorithms for the translation of assembly code to imperative high-level language code. The aim of the techniques is to remove all low-level dependencies on the machine, such as registers, stacks and condition codes, and recover high-level abstractions such as control statements, assignments, expressions, parameters, and function calls. Simple type recovery is also supported.

The results show that a high percentage of high-level language statements and control structures are recovered. There is room for improvements with the recovery of unstructured code though. From a maintenance point of view, *asm2c* is at the initial stage of a source code recovery framework. Human resources are required to enhance the documentation and readability of the recovered code, and lift it up to a maintainable level. *asm2c* provides maintainers with a quick way of translating a large percentage of their assembly code into a high-level language representation.

References

[1] O. Agesen. Design and implementation of Pep, a Java just-in-time translator. *Theory and Practice of Object Systems*, 3(2):127–155, 1997.

[2] F.E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, July 1970.

[3] B.S. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

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

[5] 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.

[6] C. Cifuentes and A. Fraboulet. Interprocedural data flow recovery of high-level language code from assembly. Technical Report 421, The University of Queensland, Department of Computer Science and Electrical Engineering, December 1997.

[7] 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.

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

[9] L. Freeman. *Don't let Missing Source Code stall your Year 2000 Project.* 1997. Year 2000 Survival Guide.

[10] P. Havlak. Nesting of reducible and irreducible loops. *Transactions on Programming Languages and Systems*, 19(4):557–567, July 1997.

[11] T. Hoffman. Recovery firm hot on heels of missing source code. *Computer World*, March 24 1997.

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

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

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

[15] U. Lichtblau. Decompilation of control structures by means of graph transformations. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, Berlin, 1985.

[16] T.A. Proebsting and S.A. Watterson. Krakatoa: Decompilation in java (does bytecode reveal source?). In *Proceedings of the Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, USA, June 1997. USENIX.

[17] N. Ramsey and C. Cifuentes. A transformational approach to binary translation of delayed branches. Technical Report 440, The University of Queensland, Department of Computer Science and Electrical Engineering, September 1998.

[18] D. Simon. Structuring assembly programs. Honours thesis, The University of Queensland, Department of Computer Science and Electrical Engineering, 1997.

[19] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

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

[21] M. Swaine. A chat with Bob Bemer. *Dr. Dobb's Journal*, pages 115–117, May 1998.

[22] 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.