Recovery of Jump Table Case Statements from Binary Code ⋆

Cristina Cifuentes ⋆⋆ and Mike Van Emmerik

Department of Computer Science and Electrical Engineering
The University of Queensland
Brisbane Qld 4072
Australia
{cristina,emmerik}@csee.uq.edu.au

Abstract

One of the fundamental problems with the analysis of binary (executable) code is that of recognizing, in a machine-independent way, the target addresses of n-conditional branches implemented via a jump table. Without these addresses, the decoding of the machine instructions for a given procedure is incomplete, leading to imprecise analysis of the code.

In this paper we present a technique for recovering jump tables and their target addresses in a machine and compiler independent way. The technique is based on slicing and copy propagation. The assembly code of a procedure that contains an indexed jump is transformed into a normal form which allows us to determine where the jump table is located and what information it contains (e.g. offsets from the table or absolute addresses).

The presented technique has been implemented and tested on SPARC and Pentium code generated by C, C++, Fortran and Pascal compilers. Our tests show that up to 90% more of the code in a text segment can be found by using this technique, when compared against the standard method of decoding. The technique was developed as part of our resourceable and retargetable binary translation framework UQBT; however, it is also suitable for other binary-manipulation and analysis tools such as binary profilers, instrumentors and decompilers.

Key words: Binary translation, binary analysis, slicing, jump tables.

⋆ This work was sponsored by the Australian Research Council under grant No. A49702762 and Sun Microsystems, Inc.

⋆⋆On sabbatical leave at Sun Microsystems, Inc.

1 Introduction

The ever growing reliance on software and the continued development of newer and faster machines has increased the need for machine-code tools that aid in the migration, emulation, debugging, tracing, and profiling of legacy code. Amongst these tools are: binary translators [19, 20], emulators/simulators [8, 23], code instrumentors [1, 17, 21], disassemblers [9, 15], and decompilers [5, 12].

The fundamental problem in decoding machine instructions is that of distinguishing code from data—both are represented in the same way in von Newmann machines. Given an executable program, the entry point to that program is given in the program's header. Information on text and data sections is also given in the header. However, data can also be stored in text areas, without such information being stored in the executable program's header. Therefore, there is a need to analyze the code that is decoded from the text area(s) of the program, and separate data from code.

The standard method of decoding machine code involves following all reachable paths from the entry point [19, 5]. This method does not give a complete coverage of the text space in the presence of indirect transfers of control such as indexed jumps and indirect calls. A common technique used to overcome this problem is the use of patterns. A pattern is generated for a particular compiler to cater for the way in which the compiler, or family of compilers, generate code for an indexed jump. This technique is extensively used as most tools deal with a particular set of compilers; for example, TracePoint only processes Windows binaries generated by the Microsoft C++ compiler [21]. In the presence of optimized code, patterns do not tend to work very effectively, even when the code is generated by a compiler known to the pattern recognizer.

In this paper we present a technique to recover the targets of indexed jumps on a variety of machines and languages. This technique is based on slicing of binary code and copy propagation into a normal form. Section 2 summarizes techniques described in the literature for the compilation of n-conditional branch statements into assembly code. Section 3 presents several Pentium and SPARC examples of code that use jump tables, Section 4 explains our normalization technique, Section 5 provides results on the use of this technique in existing benchmarks programs, and Section 6 gives previous work in this area. The paper contains an appendix with additional interesting examples.

2 Compiler Code Generation for N-Conditional Branches

N-conditional branches were first suggested by Wirth and Hoare in 1966 [24, 25] as a useful extension to the Algol language. An n-conditional branch allows a programming language to determine one of n branches in the code. This extension was implemented in Algol 68 in a form that allowed its use as a statement or an expression. In other words, the result of the `case` statement could be assigned to a variable. This high-level statement has evolved to the well known `switch` statement in C and the `case` statement in Pascal, where labels are used for the different arms of the conditional branch, and a default arm is allowed, as per Fig. 1. The C code shows the indexed variable `num` which is tested against the values in the range 2 to 7 for individual actions, and if not successful, defaults to the last `default` action.

```
#include <stdio.h>
int main()
{ int num;
    printf("Input a number, please: ");
    scanf("%d", &num);
    switch(num) {
        case 2:
            printf("Two!\n"); break;
        case 3:
            printf("Three!\n"); break;
        ...
        case 7:
            printf("Seven!\n"); break;
        default:
            printf("Other!\n"); break;
    }
   return 0;
}
```

Fig. 1. Sample switch program written in the C language.

Although not commonly documented in compiler textbooks, compiler writers generate different types of machine code for n-conditional branches. These ways of generating n-conditional branches are determined by talking to compiler writers or reverse engineering executable code. Several techniques for generating n-conditional branches from a compiler were documented in the 1970s and 1980s, when optimization for space and speed was an important issue. The most common techniques are described here based on [18].

The simplest way of generating code for an n-conditional branch is as a *linear sequence* of comparisons against each arm in the statement. This form is efficient for a small number of arms, typically 4 or less. A more sophisticated technique is the *if-tree*, where the selection is accomplished by a nested set of comparisons organized into a tree. The most common implementation is a *jump table*, which may hold labels or offsets from a particular label. This implementation requires a range test to determine the membership of values on the table. Although jump tables are the fastest method when there are many arms in the n-conditional branch, jump tables are space-wise inefficient if the case values are sparse. In such cases, a search tree is the most convenient implementation. When the arms of the n-conditional branch are sparse but yet can be clustered in ranges, a common technique used is to combine search trees and jump tables to implement each cluster of values [13, 10]. This paper deals with the issue of recovering code from generated *jump tables*, in such a way that the target addresses of an indexed jump are determined. This paper does not attempt to recover high-level n-conditional branch statements, but rather the information necessary to translate an indirect branch indexing a jump table.

For an n-conditional branch implemented using a jump table, an indexed table is set up with addresses or offsets for each of the cases of the branch. The table itself is located in a read-only data section, or mixed in with the text section. In the interest of efficiency, range tests for such jump tables need to be concise. The most common way of doing both tests is as follows [2]:

```
k <- case_selector - lower_bound
compare k with (upper_bound - lower_bound)
if unsigned_greater goto out_of_range
assertion: lower_bound <= case_selector <= upper_bound
```

If the case selector value is within the bounds of the upper and lower bounds, an offset into the jump table is calculated based on the size of each entry in the table; typically 4 bytes for a 32-bit machine. Based on the addressing modes available to a machine, either an indirect jump on the address of the table plus the offset, or an indexed jump on the same values is generated. The machine then continues execution at the target of the indirect/indexed jump.

Retargetable compilers also use these techniques. A brief description for the code generation of an indirect jump through a jump table for a retargetable C compiler is given in [11] by the following specification:

```
if t1 < v[l] goto lolab  ; l=lower bound
if t1 > v[u] goto hilab  ; u=upper bound
goto *table[t1-v[l]]
```

Overall, compiler writers use a variety of heuristics to determine which code to generate for a given n-conditional branch based on the addressing modes and instructions available on the target machine. It is also common for a compiler to have more than one way of emitting code for such a construct, based on the number of arms in the conditional branch and the sparseness of the values in such arms.

3 Examples of Existing Indexed Jumps in Binary Code

This section presents examples of Pentium and SPARC code that make use of jump tables. The examples aim to familiarize the reader with a variety of ways of encoding an n-conditional branch in assembly code, as well as to show the degree of complexity of such code. The assembly code for the examples was generated by the Unix utility `dis`. This disassembler uses the convention of placing the destination operand on the right of the instruction. The examples show annotated native Pentium and SPARC assembly code, and where relevant, the address for the assembly instructions or the indexed table. The annotations were included in these examples for ease of readability; they are not part of the produced disassembly.

```
movl -8(%ebp),%eax          ! Read index variable
subl $0x2,%eax              ! Minus lower bound
cmpl $0x5,%eax              ! Check upper bound
ja   0xffffffd9 <80489dc>   ! Exit; out of range
jmp  *0x8048a0c(,%eax,4)    ! Indexed, scaled jump

8048a0c:  64 89 04 08       ! Table of addresses
8048a10:  78 89 04 08       ! to code handling
8048a14:  8c 89 04 08       ! the various switch
8048a18:  a0 89 04 08       ! cases
...
```

Fig. 2. Pentium assembly code for sample switch program, produced by the Sun cc compiler.

The first two examples in Figs. 2 and 3 were generated by the `cc` compiler on a Solaris Pentium and SPARC machine respectively, from the sample program in Fig. 1. In Fig. 2, register `eax` is used as the index variable; its value is read from a local variable on the stack (`[ebp-8]`, the case selector). The lower bound and the range of the table are checked (2 and 5 respectively); the code exits if the value of the index variable is out of bounds. If within bounds, an indexed scaled jump on (`eax*4`) is performed, offset from the start of the indexed table at `0x8048a0c`. The contents of the values of the table are of addresses; each is displayed in little-endian format.

Fig. 3 performs the same logical steps as Fig. 2 using SPARC assembly code, where indexed jumps do not exist but indirect jumps on registers are allowed. In the example, the indexed variable is initially in `o0`, which gets set from a local variable on the stack (`[fp-20]`, the case selector). The lower bound

```
ld    [%fp - 20], %o0        ! Read index variable
add   %o0, -2, %o1           ! Minus lower bound
cmp   %o1, 5                 ! Check upper bound
bgu   0x10980                ! Exit if out of range
sethi %hi(0x10800), %o0      ! Set table address
or    %o0, 0x108, %o0        ! (continued)
sll   %o1, 2, %o1            ! Multiply by 4
ld    [%o0 + %o1], %o0       ! Fetch from table
jmp   %o0                    ! Jump
nop

10908:  0x1091c              ! Table of pointers
1090c:  0x10930
10910:  0x10944
10914:  0x10958
...
```

Fig. 3. SPARC assembly code for sample switch program, produced by the Sun cc compiler.

is computed and the indexed variable is set to `o1`. The range of the table is checked; if out of bounds, the code exits to address `0x10980`. If within bounds, the address of the table is computed to `o0` (by the `sethi` and `or` instructions), the indexed register is multiplied by 4 to get the right 4-byte offset into the indexed table, and the value of the table (`o0`) indexed at `o1` is fetched into `o0`. A jump to `o0` is then performed.

Fig. 4 presents a SPARC example that uses a hash function to determine how to index into the table. The code comes from the Solaris 2.5 `vi` program. The index variable is set as `o0`, and it is normalized by subtracting its lower bound. The range of the table is checked; if the value is out of range, a jump to the end of the case statement is performed (`0x18804`). If within bounds, the table's address is set in register `o2`. The indexed register is hashed into `o1` and multiplied by 8 (into `o4`) to get the right offset into the table (as the table contains two 4-byte entries per case). A word is loaded from the table into register `o3` and its value is compared against the hash function key (the normalized index variable `o0`). If the value matches, the code jumps to address `0x1885c`, where a second word is read from the table into `o0`, and a register jump is performed to that address. In the case where the value fetched from the table does not match the key, an end-of-hashing comparison is performed against the value `-1`. If `-1` is found, the code exits (`0x18804`), otherwise, the indexed register (`o4`) is set to point to the next value in the table (wrapping the offset into the table from the end of the table to the start) and the process is repeated at address `0x18554`. Note that this table contains 2 entries per case; the first one is the normalized index value, and the second one is the target address for the code associated with that case entry.

```
18524:  ld          [%fp - 20], %o0   ! Read indexed variable
18528:  sub         %o0, 67, %o0      ! Subtract lower bound
1852c:  cmp         %o0, 53           ! Compare with range-1
18530:  sethi       %hi(0x18800), %o2 ! Set upper table addr
18534:  bgu         0x18804           ! Exit if out of range
18538:  nop
1853c:  or          %o2, 108, %o2     ! Set lower table addr
18540:  srl         %o0, 4, %o1       ! Hash...
18544:  sll         %o1, 1, %o1       ! ...
18548:  add         %o1, %o0, %o1     ! ...function
1854c:  and         %o1, 15, %o1      ! Modulo 16
18550:  sll         %o1, 3, %o4       ! Multiply by 8
18554:  ld          [%o4 + %o2], %o3  ! First entry in table
18558:  cmp         %o3, %o0          ! Compare keys
1855c:  be          0x1885c           ! Branch if matched
18560:  cmp         %o3, -1           ! Unused entry?
18564:  be          0x18804           ! Yes, exit
18568:  nop                           ! (delay slot)
1856c:  add         %o4, 8, %o4       ! No, linear probe
18570:  and         %o4, 120, %o4     ! with wraparound
18574:  ba          0x18554           ! Continue lookup
18578:  nop

1885c:  add         %o4, %o2, %o4     ! Point to first entry
18860:  ld          [%o4 + 4], %o0    ! Load second entry
18864:  jmp         %o0               ! Jump there
18868:  nop

! Each entry is a (key value, code address) pair
1886c:  0x0
18870:  0x187b4        ! Case 'C'+0
18874:  0xffffffff     ! Unused entries have -1 (i.e. 0xffffff) as
                       ! the first entry
18878:  0x18804
1887c:  0x10
18880:  0x185b8        ! Case 'C'+0x10 = 'S'
18884:  0x2f
18888:  0x18630        ! Case 'C'+0x2f = 'r'
...
```

Fig. 4. SPARC assembly code from the vi program, produced by the Sun cc version 2.0.1 compiler.

Our last example, Fig. 5, is from the `m88ksim` SPEC95 benchmark suite. This example shows 3 groups of tests on bits of a field within a structure, which get stored in registers `edx`, `ecx` and `eax`. The three partial results are then or'd together to get the resultant indexed variable in register `ecx`. The upper bound is checked (7) and, if within bounds, a branch to address `0x8058045` is

```
8057d90:  movb   38(%eax),%al        ! Get struct member
8057d93:  testb  $0x2,%al            ! Test bit
8057d95:  setne  %edx                ! To boolean
8057d98:  andl   $0xff,%edx          ! To byte
8057d9e:  testb  $0x4,%al            ! Test another bit
8057da0:  setne  %ecx
8057da3:  andl   $0xff,%ecx          ! Save in cl
8057da9:  testb  $0x8,%al            ! Test third bit
8057dab:  setne  %eax
8057dae:  andl   $0xff,%eax          ! Save in al
8057db3:  shll   $0x2,%edx           ! To bit 2
8057db6:  shll   %ecx                ! To bit 1
8057db8:  orl    %edx,%ecx           ! Combine these two
8057dba:  orl    %eax,%ecx           ! Combine all three
8057dbc:  cmpl   $0x7,%ecx           ! Upper bound compare
8057dbf:  jbe    0x280 <8058045>     ! Branch if in range
...
8058045:  jmp    *0805f5e8(,%ecx,4) ! Table jump
...
805f5e8:  f8 7d 05 08                ! table of addresses of code to
805f5ec:  01 80 05 08                ! handle switch cases
...
```

Fig. 5. Pentium assembly code from the m88ksim program, produced by the Sun cc version 4.2 compiler.

taken, where an indexed branch is made on the contents of register `ecx`, scaled by the right amount (4), and the table address. Note that the branch (`jbe`) is the opposite of that normally found in `switch` statements (i.e. `ja`). This illustrates the danger of relying on patterns of instruction to recover indexed branch targets; such a piece of code could not be well specified in a pattern. For the interested reader, this code was produced from the C macro in Fig. 6. The appendix illustrates more examples.

```
#define FPSIZE(ir)    ((((ir->p->flgs.dest_64) ? 1 : 0) << 2) | \
                      (((ir->p->flgs.s1_64) ? 1 : 0) << 1) | \
                      ((ir->p->flgs.s2_64) ? 1 : 0))

    switch (FPSIZE(ir)) {
      case SSS: /* other code */
      /* other cases */
    }
```

Fig. 6. C source code for example in Fig. 5

4 Our Technique

We have developed a technique to recover jump table branches from disassembled code. The technique is architecture, compiler and language independent, and has been tested on CISC and RISC machines with a variety of languages and compilers (or unknown compiler, when dealing with precompiled executables). Development of general techniques is an aim in our work as analysis of executable code should not rely on particular compiler knowledge; this knowledge prevents the techniques from working with code generated by other compilers, and in most cases, for other machines.

There are 3 steps to our technique:

(1) Slice the code at the indexed/indirect register jump,
(2) Perform copy propagation to recover pseudo high-level statements, and
(3) Check against indexed branch normal forms to determine the type of jump table.

4.1 Slicing of Binary Code

Our executable code analysis framework allows for the disassembly of the code into an intermediate representation composed of register transfer lists (RTL) [6] and control flow graphs for each decoded procedure in the program. The RTL describes the effects of machine instructions in terms of register transfers, and is general enough to support RISC and CISC machine descriptions.

When an indexed or indirect jump is decoded, we create an intraprocedural backward slice of the disassembled binary code [4]. Slicing occurs by following the transitive closure of registers and condition codes that are used in a given expression. The stop criterion for a given register along a path is when that register is loaded from memory (i.e. from a local variable, a procedure argument, or a global variable), it is returned by another function, or it reaches the start of the procedure without being defined (and hence it is a register parameter set by the caller).

For the purposes of determining jump tables, we have an extra stop criterion: if the lower bound of the indexed jump is found, and other relevant information has been found, no more slicing is performed. Of course, this condition is not always satisfied as indexed tables whose first entry corresponds to the register being zero do not need to check for the lower bound. In such cases, the slice finishes by means of the other stop conditions. In the case of slices across calls, we stop if the register is returned by the call (i.e. `eax` on Pentium or o0 on SPARC); in other cases we assume registers are preserved across calls and continue slicing. This is a heuristic that works well in practice and is used rarely. The heuristic works when the machine code conforms to the operating system's application binary interface [22].

For example, for Fig. 2, the following slice is created using RTL notation:

```
(1)  eax = m[ebp-8]
(2)  eax = eax - 2
(3)  ZF = (eax - 5) = 0 ? 1 : 0
(4)  CF = (~eax@31 & 5@31) | ((eax-5)@31 & (~eax@31 | 5@31))
(5)  PC = (~CF & ~ZF) = 1 ? <exit> : <step 6>
(6)  PC = m[0x8048a0c + eax * 4]
```

Register ebp points to the stack, therefore indexed variable eax fetches a value from the local memory for that procedure. The indexed register is normalized by subtracting the lower bound (2) and its range is checked against 5 (the difference between the upper and lower bounds). If within bounds, an indexed jump is performed at statement 6.

4.2 Copy Propagation

Once a slice has been computed, we perform copy propagation on registers and condition codes. This is a common technique used in reverse engineering when recovering higher-level statements from more elementary ones, such as assembly code [3, 7] and COBOL code [14].

As per [7], 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 copy propagated 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 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-clear$_{i \to j}$ relationship for register r. More formally,

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

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$_{i \to j_k}$ property.

In our example of Fig. 2, the application of copy propagation to the slice found in Section 4.1 gives the following pseudo high-level statements:

```
(3)   jcond ([ebp-8] > 7) <exit>
(4)   jmp [0x8048a0c + ([ebp-8] - 2) * 4]
```

where `jcond` stands for conditional jump and `jmp` stands for unconditional jump. Statement 3 checks if the case selector is outside the bounds of the jump, and statement 4 performs a jump to the content (i.e. an address) of memory location `0x8048a0c + ([ebp-8] - 2) * 4`.

4.3 Normal Form Comparison

Our previous example can be rewritten in the following way:

jcond (var > num$_u$) X
jmp [T + (var - num$_l$) * w]

where var is a local variable, for example `[ebp-8]`, num$_u$ is the upper bound for the n-conditional branch, for example 7, num$_l$ is the lower bound of the n-conditional branch, for example 2, T is the indexed table's address (and is of type address), for example `0x8048a0c`, and w is a constant equivalent to the size of the word of the machine; 4 in this example. Based on this information, we can infer that the number of elements in the indexed table is num$_u$ - num$_l$ + 1, for a total of 6 in the example. The example also shows that the elements of the indexed table are labels (i.e. addresses) as the jump is to the target address loaded from the address at `0x8048a0c + ([ebp-8] - 2) * 4` .

The previous example only shows one of several normal forms that are used to encode n-conditional branches using a jump table. We call the previous normal form type A. Fig. 7 shows the 3 different normal forms that we have

Type	Normal Form	Types of <expr> allowed
A	jcond (r[v] > num_u) X jmp [T + <expr> * w]	r[v] r[v] - num_l ((r[v] - num_l) << 24) >> 24)
O	jcond (r[v] > num_u) X jmp [T + <expr> * w] + T *or* jmp PC + [PC + (<expr> * w + k)] jmp PC + [PC + (<expr> * w + k)] + k	r[v] r[v] - num_l ((r[v] - num_l) << 24) >> 24) ((r[v] - num_l) << 24) >> 24)
H	jcond (r[v] > num_u) X jmp [T + ((<expr> & mask) * 2*w) + w]	((r[v] - num_l) >> s) + (r[v] - num_l) ((r[v] - num_l) >> 2*w) + ((r[v] - num_l) >> 2) + (r[v] - num_l)

Fig. 7. Normal forms for n-conditional code after analysis

identified in executable code that runs on SPARC and Pentium. Normal form A (address) is for indexed tables that contain labels as their values. Normal form O (offset) is for indexed tables that contain offsets from the start of the table T to the code of each case. Normal form H (hashing) contains labels or offsets in the indexed table. Form O can also be found in a position independent version as well. Normal form H contains pairs (<value>,<address>) at each entry into the jump table.

In our 4 examples of Figs. 2 to 5, we find the following normal forms, respectively:

- ```
  jcond (r[24] > 5) 0x80489dc
  jmp [0x8048a0c + (r[24] * 4)]
  ```
  ⇒ normal form A
- ```
  jcond (r[9] > 5) 0x10980
  jmp [0x10908 + (r[9] * 4)]
  ```
  ⇒ normal form A
- ```
  jcond ((r[8] - 67) > 53) 0x18804
  jmp [0x1886c + ((((((r[8] - 67) >> 4) << 1) + (r[8] - 67)) & 15) << 3)]
  ```
  ⇒ normal form H
- ```
  jcond (((al < 2 ? 1:0) & 0xff) << 2 | ((al < 4 ? 1:0) & 0xff) << 1 |
         ((al < 8 ? 1:0) & 0xff) > 7) 0x8057dc5
  jmp [0x805f5eb + (((al < 2 ? 1:0) & 0xff) * 4 | ((al < 4 ? 1:0) & 0xff) * 2 |
       ((al < 8 ? 1:0) & 0xff)) * 4]
  ```
  ⇒ normal form A

Examples of form O are given in the appendix.

5 Experimental Results

We tested the technique for recovery of jump table branches on Pentium and SPARC binaries in a Solaris environment. The following integer SPEC95 benchmark programs were used for testing:

- go: artificial intelligence; plays the game of Go
- m88ksim: Motorola 88K chip simulator; runs test program
- gcc: GNU C compiler; builds SPARC code
- compress: compresses and decompresses a file in memory
- li: LISP interpreter
- ijpeg: graphic compression and decompression
- perl: manipulates strings (anagrams) and prime numbers in Perl
- vortex: a database program

All benchmark programs were compiled with the Sun cc compiler version 4.2 on a Solaris 2.6 machine using standard SPEC optimizations (i.e. -O4 on SPARC and -O on Pentium). We also include results for the `awk` script interpreter utility, and the `vi` text editor (on both Solaris 2.5 and 2.6). These programs are part of the Unix OS.

Benchmark	A	O	H	Unknown
awk	0	2	0	0
vi (2.5)	10	1	9	0
vi	0	13	0	1
go	0	5	0	0
m88ksim	0	10	0	2
gcc	0	153	0	1
compress	0	0	0	0
li	0	3	0	0
ijpeg	0	3	0	1
perl	0	32	0	0
vortex	0	21	0	0
total	10	243	9	5

Fig. 8. Number of indexed jumps for SPARC benchmark programs

Benchmark	A	O	H	Unknown
awk	6	0	0	0
vi	12	0	0	2
go	5	0	0	0
m88ksim	17	0	0	0
gcc	207	0	0	5
compress	0	0	0	0
li	3	0	0	0
ijpeg	7	0	0	0
perl	36	0	0	1
vortex	13	0	0	6
total	306	0	0	14

Fig. 9. Number of indexed jumps for Pentium benchmark programs

Fig. 8 and 9 show the number of indexed jumps found in each benchmark program, the classification of such indexed jumps into the 3 normal forms (A, O and H), and any unknown types. In the case of SPARC code, most indexed jump tables are of form O, which means that the indexed table stores offsets from the start of the table to the destination target address. In the case of Pentium code, almost all indexed jump tables are of form A, meaning that the table contains the target addresses for each of the entries in the `case` statement. Unknown entries show the number of jump tables that were not recovered by this technique. These are normally due to highly optimized code that relies on indirect function calls, or on enumerated types which do not do any bounds checking.

The primary motivation for this work was to increase our coverage of decoded code in an executable program. We measured the coverage obtained from our technique using the size in bytes of the text segment(s) of the program, compared to the number of bytes decoded and the number of bytes in jump tables. The figures do not necessarily add up to 100% due to unreachable code during the decoding phase. Also, in the case of SPARC, we duplicate some instructions in order to remove delayed branch instructions; this duplication is counted twice in our model, leading to slightly over 100% coverage in rare cases. Fig. 10 and 11 show the results of our coverage analysis. The results show that when indexed tables are present in the program, up to 90% more of the code can be reached by decoding such tables correctly.

The `li` and `ijpeg` programs show a small coverage of their code sections. This is due to indirect calls on registers which are not yet analysed in our

Program	w/o analysis	with analysis	difference
awk	22%	64%	42%
vi (2.5)	24%	93%	69%
vi	30%	95%	65%
go	91%	100%	9%
m88ksim	37%	69%	32%
gcc	58%	89%	31%
compress	91%	91%	0%
li	33%	36%	3%
ijpeg	20%	22%	2%
perl	10%	99%	89%
vortex	70%	79%	9%

Fig. 10. Coverage of code for SPARC benchmarks

Program	w/o analysis	with analysis	difference
awk	22%	65%	43%
vi	28%	88%	60%
go	89%	99%	10%
m88ksim	36%	73%	37%
gcc	52%	86%	34%
compress	84%	84%	0%
li	24%	26%	2%
ijpeg	18%	20%	2%
perl	9%	99%	90%
vortex	68%	75%	7%

Fig. 11. Coverage of code for Pentium benchmarks

framework to determine their target addresses. In the case of `ijpeg`, a large percentage of the procedures are reached only via indirect calls, hence they are never decoded. In the context of our binary translation framework, we rely on an interpreter to process such code at runtime.

6 Previous Work

Not much work has been published in the literature on recovery of indexed jump targets. These techniques tend to be ad hoc and tailored to a specific platform or compiler, and tend to rely on pattern matching.

The qpt binary profiler is a tool to profile and trace code on MIPS and SPARC platforms. Profiling and tracing is done by instrumenting the executable code. Jump tables are detected by relying on the way in which the compiler generated code for the jump, mainly by expecting the table to be in the data segment in the case of MIPS or in the code segment, immediately after the indirect jump, on the SPARC. The end of the table is found by examining the instructions prior to the indirect jump and determining the table's size; alternatively, the text space is scanned until an invalid address is met [16].

The dcc decompiler is an experimental tool for decompiling 80286 DOS executables into C code. The method used in this tool was that of pattern matching against known patterns generated by several compilers on a DOS machine [5].

EEL is an executable editing library for RISC machines. Slicing is used to determine the instructions that affect the computation of the indirect jump and determine the jump table. No precise method is given. Measurements on the success of this technique on SPARC using the SPEC92 benchmarks reveal that 100% recovery of indexed jumps is achieved for code compiled by the gcc and the Sun Fortran compilers, and 89% for the SunPro compilers. The recovery ratio was measured by counting the number of indirect jumps expected and recovered [17].

IDA Pro, a disassembler for numerous machines, makes use of undocumented techniques to determine which compiler was used to compile the original source program [15]. IDA Pro's recovery of jump tables is good but their technique has not been documented in the literature.

Our techniques compare favourably with those of other tools. They have been tested extensively with code generated from different compilers on both CISC and RISC machines, indicating the generality and machine independence of the technique.

7 Conclusions

We have presented a technique based on slicing and copy propagation to understand and recover the code of n-conditional branches implemented by jump tables. The technique is suitable for recovery of code in machine-code manipulation tools such as binary translators, code instrumentors and decompilers.

Our technique has been tested on Pentium and SPARC code in a Solaris environment against the SPEC95 integer benchmark programs. Over 500 n-conditional branches were correctly detected in these programs, making this technique suitable for path coverage during the decoding of machine instructions. For the `perl` benchmark, over 90% of extra code was decoded due to this recovery technique.

This work is part of the resourceable and retargetable binary translation project UQBT. For more information refer to `http://www.csee.uq.edu.au/csm/uqbt.html`

References

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

[2] R.L. Bernstein. Producing good code for the case statement. *Software–Practice and Experience*, 15(10):1021–1024, 1985.

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

[4] C. Cifuentes and A. Fraboulet. Intraprocedural static 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.

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

[6] C. Cifuentes and S. Sendall. Specifying the semantics of machine instructions. In *Proceedings of the International Workshop on Program Comprehension*, pages 126–133, Ischia, Italy, 24–26 June 1998. IEEE CS Press.

[7] C Cifuentes, D Simon, and A Fraboulet. Assembly to high-level language translation. In *Proceedings of the International Conference on Software Maintenance*, pages 228–237, Washington DC, USA, 18–20 November 1998. IEEE CS Press.

[8] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for

execution profiling. In *Proceedings ACM SIGMETRICS Conference on Measurement and modeling of Computer Systems*, 1994.

[9] V Communications. *Sourcer - Commenting Disassembler, 8088 to 80486 Instruction Set Support.* V Communications, Inc, 4320 Stevens Creek Blvd., Suite 275, San Jose, CA 95129, 1991.

[10] C.W. Fraser and D.R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[11] C.W. Fraser and D.R. Hanson. *A Retargetable C Compiler: Design and Implementation.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1995.

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

[13] J.L. Hennessy and N. Mendelsohn. Compilation of the Pascal case statement. *Software–Practice and Experience*, 12:879–882, 1982.

[14] H. Huang, W. Tsai, S. Bhattacharya, X. Chen, and Y. Wang. Business rule extraction techniques for COBOL programs. *Journal of Software Maintenance: Research and Practice*, 10(1):3–35, January 1998.

[15] Ida PRO disassembler, Data Rescue. `http://www.datarescue.com/ida.htm`, 1997.

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

[17] J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *SIGPLAN Conference on Programming Languages, Design and Implementation*, pages 291–300, June 1995.

[18] A. Sale. The implementation of case statements in Pascal. *Software–Practice and Experience*, 11:929–942, 1981.

[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] T. Thompson. An Alpha in PC clothing. *Byte*, pages 195–196, February 1996.

[21] TracePoint. HiProf hierarchical profiler. `http://www.tracepoint.com/noframes/hiprof/products/hiprof`, 1997.

[22] *Unix System V: Application Binary Interface.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[23] The wine project. `http://www.winehq.com`, 1996.

[24] N. Wirth and C.A.R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9(6):413–432, 1966.

[25] C. Wrandle. Notes on the `case` statement. *Software–Practice and Experience*, 4:289–298, 1974.

A APPENDIX

Figs. A.1 and A.2 illustrate two examples of form O from SPARC code. The former contains an indexed table of offsets from the table to the code that handles each individual switch case. The latter also contains an indexed table of offsets from the table to the code, however, the way the address of the table is calculated is position independent code (via the call to `.+8`, which produces the side effect of setting the `o7` register with the current program counter).

```
10a58:  0x0009c               ! Indexed table
10a5c:  0x000dc               ! of offsets
10a60:  0x000fc
10a64:  0x0011c
...

  sethi %hi(0x10800), %l1   ! Set table address
  add   %l1, 0x258, %l1     ! into %l1
  ...
  ld    [%fp - 4], %l0      ! Read idx variable
  sub   %l0, 2, %o0         ! Subtract min val
  cmp   %o0, 5              ! Cmp with range-1
  bgu   0x10b14             ! Exit if out of range
  sll   %o0, 2, %o0         ! Multiply by 4
  ld    [%o0 + %l1], %o0    ! Fetch from table
  jmp   %o0 + %l1           ! Jump to table+offset
  nop                       ! Delay slot instr
```

Fig. A.1. Form O example for SPARC assembly code.

```
  ldsb  [%l6], %o0          ! Get switch var
  clr   %i3                 ! (Not relevant)
  sub   %o0, 2, %o0         ! Subtract min value
  cmp   %o0, 54             ! Cmp with range-1
  bgu   0x44acc             ! Exit if out of range
  sll   %o0, 2, %o0         ! Multiply by 4
43eb8:
  call  .+8                 ! Set %o7 = pc
  sethi %hi(0x0), %g1       ! Set %g1 = 0x0001c
  or    %g1, 0x1c, %g1      !
  add   %o0, %g1, %o0       ! %o0 = 0x43eb8 + 0x1c
                            !     = 0x43ed4
  ld    [%o7 + %o0], %o0    ! Fetch from table
  jmp   %o7 + %o0           ! Jump to table+offset
  nop                       ! Delay slot instr

43ed4:  0x0021c             ! Table of offsets from
43ed8:  0x00af4             ! call instr to case code
43edc:  0x000f8             ! e.g. 0x43eb8 + 0x00f8
                            !      = 0x43fb0
43ee0:  0x008d0
...
```

Fig. A.2. Form O example for SPARC assembly code (vi 2.5) using position independent code. Offsets are relative to the address of the call instruction.

```
4233c:
 call     .+8                  ! Set %o7 = pc
 sethi    %hi(0xfffffc00), %o2 ! %o2 = -1024
 ldsb     [%i5], %o0           ! %o0 = <expr>
 add      %o2, 736, %o2        ! %o2 = -288
 add      %o2, %o7, %o2        ! %o2 = pc-288
 sub      %o0, 2, %o0          ! Subtract min value
 mov      %o1, %o7             ! (Not relevant)
 mov      %o2, %o1             ! %o1 = <expr>
 cmp      %o0, 54              ! Cmp with range-1
 add      %l1, 1, %l1          ! (Not relevant)
 bgu      0x42f58              ! Exit if out of range
 sll      %o0, 2, %o0          ! Multiply <expr> by 4
 ld       [%o0 + %o1], %o0     ! Fetch from table
 jmp      %o0 + %o1            ! Jump to table+off
 nop                           ! Delay slot instr

(288 bytes earlier than the call instruction):
 4221c:   0x298                ! Table of offsets from
 42220:   0xd3c                ! table to case code
 42228:   0x15c                ! e.g. 0x4221c + 0x15c
                               !      = 0x42378
 4222C:   0x8fc
 ...
```

Fig. A.3. A different form O example for SPARC assembly code, also using position independent code. This code is generated from the same source code as the example in Fig. A.2, but with a different version of the compiler. Offsets are relative to the start of the table.