Recovery of Jump Table Case Statements from Binary Code*

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

Technical Report 444
December 1998

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, as well as any analysis on that procedure.

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 expression substitution. 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 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. The technique was developed as part of our retargetable binary translation framework; however, it is also suitable for other binary-manipulation tools such as binary profilers, instrumentors and decompilers.

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 [21, 22], emulators/simulators [11, 3], code instrumentors [4, 19, 23], disassemblers [12, 2], and decompilers [8, 15].

The fundamental problem in decoding machine instructions is that of distinguishing code from data—both are represented in the same way in von Neumann 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. Hence the 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 [21, 8]. 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

---

*This work is sponsored by the Australian Research Council under grant No. A49702762 and Sun Microsystems.
as most tools deal with a particular set of compilers; for example, TracePoint only processes Windows binaries generated by the Microsoft C++ compiler [23]. 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 [7] and expression substitution into a normal form. Section 3 presents several Pentium and SPARC examples of code that use indexed jumps, 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 extra 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 could be assigned to a variable. This high-level statement has evolved to the well known switch statement in C or 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 Figure 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.

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 70s and 80s, where optimization for space and speed was an important issue. The most common techniques are described here based on [20].

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

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

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 [16, 13]. 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 case 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 [5]:

\[
\begin{align*}
  k & \leftarrow \text{case} \_ \text{selector} - \text{lower} \_ \text{bound} \\
  \text{compare} \ k \ \text{with} \ (\text{upper} \_ \text{bound} - \text{lower} \_ \text{bound}) \\
  \text{if unsigned} \_ \text{greater} \ \text{goto out of range} \\
  \text{assertion: lower} \_ \text{bound} \ <= \text{case} \_ \text{selector} <= \\
  \text{upper} \_ \text{bound}
\end{align*}
\]

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 branch table for a retargetable C compiler is given in [14] in the following specification:

\[
\begin{align*}
  \text{if} \ t1 < v[l] \ \text{goto lolab} \ ; \ l=\text{lower bound} \\
  \text{if} \ t1 > v[u] \ \text{goto hilab} \ ; \ u=\text{upper bound} \\
  \text{goto *table[t1-v[l]]}
\end{align*}
\]

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 indexed 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 assembly code, and where relevant, the address for the assembly instructions or the indexed table.

#### Pentium assembly code

```
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
```

Figure 2. Pentium assembly code for simple switch program, produced by the cc compiler.

#### SPARC assembly code

```
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
```

Figure 3. SPARC assembly code for simple switch program, produced by the cc compiler.

The first two examples in Figures 2 and 3 were generated by the cc compiler on a Solaris Pentium and SPARC machine respectively, from the sample pro-
gram in Figure 1. In Figure 2, register eax is used as the index variable; its value is read from a local variable on the stack ([ebp-8]). 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.

Figure 3 performs the same logical steps as Figure 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 lower bound 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.

Figure 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.

The appendix illustrates two more examples.

---

8057d90: movb 38(%eax),%al
8057d93: testb $0x2,%al
8057d95: setne %edx
8057d98: andl $0xff,%edx
8057d9e: testb $0x4,%al
8057da0: setne %ecx
8057da3: andl $0xff,%ecx
8057da9: testb $0x8,%al
8057dab: setne %eax
8057dae: andl $0xff,%eax
8057db3: shll $0x2,%edx
8057db6: shll %ecx
8057db8: orl %edx,%ecx
8057dba: orl %eax,%ecx
8057dbc: cmpl $0x7,%ecx
8057dbf: jbe 0x280 <8058045>
...
8058045: jmp *134608360(,%ecx,4)
...
805f5e8: f8 7d 05 08 ! Table of addresses
805f5ec: 01 80 05 08 ! of code to handle
     ... ! switch cases

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

Our last example, Figure 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 taken, where an indexed branch is made on the contents of register ecx, offset 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. The appendix illustrates two more examples.
We have developed a technique to recover n-conditional 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 expression substitution to recover pseudo high-level statements, and

3. Check against indexed branch normal forms.
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) \[9\] 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 \[7\]. 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 set by the caller).

For the purposes of determining indexed 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 starting at 0 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. \texttt{eax} on Pentium or \texttt{a0} 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 needed in only a few cases. The heuristic works when the machine code conforms to the operating system’s application binary interface \[1\].

4.2. Expression Substitution

Once a slice has been computed, we perform expression substitution by means of forward substitution. This is a common technique used in reverse engineering when recovering higher-level statements from more elementary ones, such as assembly code \[6, 10\] and COBOL code \[17\].

As per \[10\], a definition of a register \( r \) at instruction \( i \) in terms of a set of \( a_k \) registers, \( r = f_i (\{a_k\}, i) \), can be forward substituted at the use of that register on another instruction \( j \), \( s = f_j (\{r, \ldots\}, j) \), if the definition at \( i \) is the unique definition of \( r \) along all paths in the program, and no register \( a_k \) has been re-defined along that path. The resulting instruction at \( j \) would then look as follows:

\[
s = f_j (\{f_i (\{a_k\}, i), \ldots\}, j)
\]

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 relationship for register \( r \). More formally,

\[
|\text{du}(r, j)| = 1 \land \\
\text{ud}(r, j) = \{ i \} \land \\
j \in du(r, i) \land \\
\forall a_k \bullet a_k \text{-clear}_{i \rightarrow j}
\]

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 relationship.

In our example of Figure 2, the application of forward substitution to the slice found in Section 4.1 gives the following pseudo high-level statements:

\[
(3) \ jcond ([ebp-8] > 7) 0x80489dc
(4) \ jmp [0x8048a0c + ([ebp-8] - 2) * 4]
\]
4.3. Normal Form Comparison

Our previous example can be rewritten in the following way:

\[
\text{jcond } (\text{var} > \text{num}_u) X \\
\text{jmp } [T + (\text{var} - \text{num}_l) \times w]
\]

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 this case. In this example, 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 an indexed jump table. We call the previous normal form type A. Figure 6 shows the 3 different normal forms that we have 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 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.

Normal form H contains pairs (<value>,<address>) at each entry into the indexed table.

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

\begin{itemize}
  \item jcond (r[24] > 5) 0x80489dc \\
  jmp [0x8048a0c + (r[24] * 4)] \\
  \Rightarrow \text{normal form A}
  \item jcond (r[9] > 5) 0x10980 \\
  jmp [0x10908 + (r[9] * 4)] \\
  \Rightarrow \text{normal form A}
  \item jcond ((r[8] - 67) > 53) 0x18804 \\
  jmp [0x1886c + \\
  ((((((r[8] - 67) >> 4) << 1) \\
  + (r[8] - 67)) & 15) << 3)] \\
  \Rightarrow \text{normal form H}
\end{itemize}

Examples of form O are given in the appendix.

5. Experimental Results

We tested the technique for recovery of indexed branch code on Pentium and SPARC in a Solaris environment. The following SPEC95 benchmark programs were used for testing:

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

All benchmark programs were compiled with the Sun cc compiler version 4.2 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.

Figures 7 and 8 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.

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 indexed 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. Figures 9 and 10 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.

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. Indexed 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 [18].

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 [8].

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 indexed 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 [19].

IDA Pro, a Pentium disassembler, makes use of compiler patterns to determine which compiler was used to compile the original source program [2]. IDA Pro’s recovery of indexed 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 expression substitution to understand and recover the code of n-conditional branches implemented by indexed 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 retargetable binary translation project at The University of Queensland. For more information refer to:

A. Appendix

Figures 11 and 12 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 \texttt{.+8}, which produces the side effect of setting the \texttt{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], %10 ! Read idx variable
sub %10, 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
```

```
43ed4: 0x0021c ! Table of offsets from
43ed8: 0x000af4 ! table to case code
43edc: 0x000f8 ! e.g. 0x43ed4 + 0x00f8
        ! = 0x43fbc
43ee0: 0x0008d0 ...
```

Figure 11. Form O example for SPARC assembly code.

```
ldsb [%l6], %0 ! Get switch var
clr %i3 ! (Not relevant)
sub %00, 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
call .+8 ! Set \%o7 = pc
sethi %hi(0x0), %g1 ! Set \%g1 =
or %g1, 0x1c!, %g1 ! %0001c
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
```

Figure 12. Form O example for SPARC assembly code using position independent code.

References


[14] C. Fraser and D. Hanson. A Retargetable C Compiler: Design and Implementation. The Ben-


