Analysing Switch-Case Tables by Partial Evaluation

Niklas Holsti

Tidorum Ltd
Tiirasaarentie 32, FI 00200 Helsinki, Finland
niklas.holsti@tidorum.fi

Abstract

Tracing the flow of control in code generated from switch-case statements is difficult for static program analysis tools when the code contains jumps to dynamically computed target addresses. Analytical methods such as abstract interpretation using integer intervals can work for some forms of switch-case code, for example a jump via a table of addresses indexed 1 .. n, but fail when the target compiler encodes the switch-case structure in a ROM table with a complex format and uses a library routine to interpret the table at run-time.

This paper shows how to extract the flow of control from such switch-case tables by partial evaluation of the table-interpreting routine. The resulting control-flow graph allows accurate analysis of the execution time and the logical conditions for reaching each case in the switch-case statement.

The method is implemented in Tidorum's Bound-T tool for worst-case execution-time analysis. The implementation builds on some basic Bound-T features for modeling program states in the flow-graph and propagating constant values through the graph.

1. Introduction

Static analysis of the worst-case execution time (WCET) of a program usually begins by building the control-flow graph (CFG). On the machine code level, where most WCET tools work, the tool has to find the possible successor instructions of each instruction under analysis. This is easy when the instruction defines its successors statically but hard for control-transfer instructions with dynamic target addresses, for example register-indirect jumps. Such *dynamic transfer of control* (DTC) instructions often result from switch-case statements [1, 6, 8].

The switch-case statement in languages such as C or Ada is a very flexible control structure. The programmer can choose the type of the switch index, for example an 8-bit or a 32-bit number; whether the cases are numbered densely 1 .. *n* or are a sparse subset of a large range; whether each case is reached by a unique index value or by a set or range of values; and whether there is a default case or not. Compilers often generate quite different kinds of code to implement different kinds of switch-case statements.

For small target processors such as the Intel 8051 or Atmel AVR some compilers try to reduce code size by encoding the switch-case statement into a ROM *switch table* and generating a call or jump to a *switch handler* routine that interprets the table at run-time. There may be several types of switch table, for example depending on the index type, each with its own switch handler.

This paper describes a way to find the full control-flow graph for code that uses switch tables and switch handlers. Section 2 defines a particular switch-table structure and the corresponding switch handler for use in examples. Section 3 states the problem and the goals for the solution. Section 4 defines the suggested solution as a form of partial evaluation. Sections 5 and 6 explain how this solution is implemented in the Bound-T WCET tool [2] and section 7 shows an example. Section 8 summarises the analysis method. Sections 9 and 10 report experience from implementation and experiments, respectively. Section 11 discusses related earlier work and section 12 concludes the paper.

When discussing the Bound-T implementation I will use the term “flow-graph” instead of the usual “control-flow graph”. Section 5 explains why.

2. Example of switch table and handler

The switch-table structure in this example was chosen to make the switch handler brief but not trivial. The structure is not taken directly from any compiler that I know of but is similar to real switch tables for 8-bit processors. The structure assumes a type of Atmel AVR processor with a 16-bit program counter and at most 64 KB of program memory.

In this example a switch table is a sequence of *entries*. An entry represents a set of 8-bit switch-index values that lead to the same case in the switch-case statement. An entry consists of four octets: a *mask* octet, a *match* octet, and the low and high octets of the 16-bit *address* for the case to be taken when the bit-wise logical “and” of the switch index and the *mask* octet equals the *match* octet. The order of entries in the table is arbitrary but the last entry always has a zero *mask* and *match*. This represents the default case if there is one, else fall-through to the statement after the switch-case.

Consider this C subprogram *foo*:

```
void foo (unsigned char k)
{
   switch (k) {
   case 4:
      <statements for k = 4>
   case 8: case 9: case 11:
      <statements for k = 8, 9 or 11>
   default:
      <statements for other values of k>
   }
}
```

The switch table for this switch-case statement is shown in Table 1 below. It has four entries for a total size of 16 octets. Note that the second entry matches both $k = 8$ and $k = 9$ because the mask value 254 masks the least significant bit of k.

In this example a switch-case statement is compiled into code that loads the switch index (the parameter *k* in *foo*) into register *r0* and calls the switch handler *SwHandler*. The switch table is placed in the program memory immediately after the call so that the return address points to the first table entry. *SwHandler* searches the table for an entry that matches the switch index, then jumps to the *address* of this entry.

SwHandler can be written in AVR assembly language [3] as shown in Listing 1 below. For later reference the left margin shows the assumed word address of the instruction (in hex). Semicolons start comments that extend to end of line.

Listing 2 below shows the AVR code for function *foo* including the code for the switch-case statement and the hex form of the switch table. Listing 2 assumes that *k* is passed to *foo* in register *r16* and some arbitrary amounts of code in the case branches.

Listing 1. Example switch handler

```
SwHandler:
          ; Switch-case handler. Entered by call with the
          ; switch index in r0 and the switch table in
          ; program memory after the call instruction. Exits
          ; to the chosen case. Changes r1, r2, and Z.

0100 pop   r30        ; low octet of table address
0101 pop   r31        ; high octet of table address
          ; Z = r31:r30 = word address of the switch table.
0102 add   r30,r30    ; Multiply Z by two to make
0103 adc   r31,r31    ; it an octet address for lpm.

loop:          ; Z points at the next switch table entry.
0104 lpm   r1,Z+      ; r1 := entry.mask
0105 lpm   r2,Z+      ; r2 := entry.match
0106 and   r1,r0      ; r1 := index and mask
0107 cp    r1,r2      ; compare to entry.match
0108 breq  found      ; branch if entry matches index
0109 adiw  Z,2        ; no match, point at next entry
010A rjmp  loop       ; try next entry

found:         ; Entry matches. Z points at entry.address.
010B lpm   r1,Z+      ; r1  := address low octet
010C lpm   r31,Z      ; r31 := address high octet
010D mov   r30,r1     ; Z   := whole address
010E ijmp             ; DTC jump to address in Z.
```

Table 1. Example switch table

mask	*match*	*address points to:*
255	4	the code for the case $k = 4$
254	8	the code for the case $k = 8$, 9 or 11
255	11	the code for the case $k = 8$, 9 or 11
0	0	the code for the default case

Listing 2. Example switch-case statement code

```
foo:
0200 mov   r0,r16   ; r0 := k
0201 call  SwHandler
          ; The switch table consists of the following
          ; 16 octets, shown in hex:
0203 FF 04 0B 02    ; k = 4, address = 020B
0205 FE 08 1C 02    ; k = 8 or 9, address = 021C
0207 FF 0B 1C 02    ; k = 11, address = 021C
0209 00 00 24 02    ; default, address = 0224
020B < code for the case k = 4 >
021C < code for the case k = 8, 9 or 11 >
0224 < code for the default case >
0229 ret            ; return from foo.
```

3. Problem and goals

The problem is to find the full control-flow graph for machine code that uses switch tables and switch handlers, for example with the structure described in section 2 but of course not limited to that example. The machine code is given as a memory image that is a mixture of code and data, not clearly demarcated. The solution should:

- find all cases of all switch-case statements,
- *not* mix up different switch-case statements to create false paths in the flow-graph,
- produce the sequence of instructions that leads to each case, so that later steps in the analysis can find an accurate WCET for each case,
- connect each case with the corresponding values of the switch index, again for use in later analysis steps (for example to find bounds for a loop that is nested in a case and depends on the switch index),
- apply uniformly to several kinds of switch tables and handlers and be robust to changes in their structure as the compilers evolve.

The solution should also be easy to implement in the generic, processor-independent parts of a WCET tool, in my case Bound-T [2], with minimal changes to the processor-specific parts, for example the parts of Bound-T that decode AVR instructions.

Bound-T can analyse many aspects of a subprogram in a (calling-) context-dependent way but the flow-graph of a subprogram must be independent of context. Analysing a switch handler (for example *SwHandler*) as an ordinary, independent subprogram cannot give a context-dependent resolution of the DTC (the *ijmp* in *SwHandler*). Instead, a switch handler must be analysed as an integral part of the subprogram that contains the switch-case statement (for example *foo*). This is similar to in-line expansion of the call to the switch handler.

The target addresses for the DTC result from executing the switch-handler instructions that access the switch table. The analysis must thus simulate or execute these instructions. Furthermore, the analysis must unroll the table-scanning loop in the switch handler. Each iteration of the loop leads to a different case; unrolling the loop separates the paths to the different cases for separate analysis.

4. The solution by partial evaluation

Partial evaluation is the execution of a program with some inputs bound to concrete values but other inputs not so bound (free input variables) [4]. The result is therefore not a concrete output value but a *residual program* that still depends on the unbound inputs. The residual program is a *specialization* of the original program: it is specialized to the domain where the bound inputs have the given values.

The proposed analysis of switch tables and switch handlers uses partial evaluation of subprograms as follows. A switch handler is a subprogram with two inputs: the switch index and the switch table. At analysis time, in a given invocation of a switch handler for a given switch-case statement the switch index is usually unbound (has an unknown, dynamic value) but the switch table is bound to a static constant: the table generated for this switch-case statement.

If we partially evaluate the switch handler under this binding, the residual subprogram depends only on the switch index and not on the switch table. The partial evaluation resolves the DTC instructions into control transfers with static target addresses, copied or computed from the switch table.

For the switch handler shown in section 2 partial evaluation with a known switch table means that we know the value loaded by the execution of any *lpm* instruction. Thus the target address of each possible execution of the *ijmp* DTC instruction is known even if the value of the switch index (*r0*) is unknown.

Within the Bound-T tool the partial evaluation is implemented in a way that fits the Bound-T architecture, not as a general-purpose partial evaluator such as the *mix* evaluator described in [4]. In Bound-T the original, unevaluated subprogram (the switch handler) is represented implicitly by its entry address and the instructions in the target program that can be reached from the entry address. The residual subprogram (the switch handler specialized to a given switch table) is represented as a part of the flow-graph of the subprogram that contains the switch-case statement. This part is a subgraph rooted at the node that invokes the switch handler. The nodes of the subgraph represent (executions of) instructions in the switch handler; the leaves of the subgraph represent the DTC leading to each case.

In the terminology of [4] the *source language* of this partial evaluator is machine-code program-memory images and the *target language* is Bound-T flow-graphs. (As a part of Bound-T the *implementation language* is Ada, but this is not important.)

The next two sections explain how partial evaluation is implemented in Bound-T and why it is a natural extension of the way in which Bound-T builds flow-graphs from machine code. This says more about Bound-T than about the partial evaluation method for switch-case analysis. Eager readers may skip to section 7 for an example of the analysis.

5. Building flow-graphs in Bound-T

This section describes the structure of flow-graphs in Bound-T and the iterative algorithm for building flow-graphs from machine code. The next section extends the algorithm to include partial evaluation.

First a definition of terms. The internal representation of a subprogram in Bound-T is a *flow-graph* (FG). A flow-graph differs from a control-flow graph (CFG) because (as defined in this paper) a CFG node represents a given machine instruction in *any* program state while an FG node represents a given instruction in some *subset* of program states. Thus, a given instruction is always represented in at most one CFG node, but can be represented in several FG nodes when this instruction is modeled separately for different program states. Bound-T adopted the flow-graph concept to model complex control mechanisms such as nested zero-overhead loops in DSPs.

The abstraction of the program state that is used for flow-graphs is called the *flow-state*. The program counter (PC) is always a concrete part of the flow-state; a flow-state implies a PC value. Each node in a flow-graph is *tagged* with a flow-state. No other node in this flow-graph is tagged with this flow-state.

Each flow-graph node has several attributes to model the instruction in this node. For this paper the main attribute is the *computational effect*: a set of assignments of expressions to variables (registers or memory locations). For example, the effect of the AVR instruction $lpm\ r1, Z+$ is modeled by the assignments $r1 := pm[Z]$, $Z := Z+1$ where $pm[Z]$ stands for the value of the program memory octet at address Z. (Ignore the fact that the 16-bit Z pointer is composed of the two 8-bit registers *r30* and *r31*. This complication is nasty but not relevant here.)

Each edge in the flow-graph is provided with a Boolean expression that is a necessary but perhaps not sufficient *condition* for taking this edge. For example, the condition for the branch-taken edge after the AVR instruction *breq* is that the "zero" flag be set, here written as $zf = 1$. The condition is evaluated after the effect of the source node.

Bound-T starts the analysis of a subprogram by building the flow-graph of the subprogram. This is an iterative algorithm very like the algorithm in [6]. For each new flow-state the algorithm first adds a blank node to the flow-graph and then proceeds to fill in the blank nodes with their attributes. The final flow-graph contains all flow-states and instructions in the subprogram that can be reached from the entry address. The algorithm follows.

Building the flow-graph of a subprogram in Bound-T
Initialization. The flow-graph is initialized to consist of one blank node tagged with the flow-state that represents the entry address of the subprogram.
Iteration. The algorithm repeatedly executes the *Fill node* step until there are no blank nodes in the flow-graph.
Fill node. Pick a blank node *N* from the flow-graph. The flow-state of the node identifies (through its PC value) the instruction executed in this state. Fetch this instruction from the memory image of the target program and fill in the attributes of node *N* from this instruction. Determine all successor flow-states for node *N*. The successor of a normal call instruction is the return point in the caller. A normal return instruction has no successors. For each successor state *s* of *N*, if there is not already a flow-graph node *S* tagged with *s* then add such a new blank node *S* to the flow-graph. Make a new edge from *N* to *S*.

The analysis of normal subprogram calls in Bound-T is not relevant to this paper because a call to a switch handler will be analysed as if the call were *in-lined*. A switch-handler call is analysed as a kind of jump instruction by a simple variation of the above algorithm: the successor of a call to a switch handler is taken to be the first instruction in the switch handler, instead of the return point in the callee.

For the example in section 2 the return point contains the switch table, not AVR instructions, so control never reaches the return point.

6. Partial evaluation in Bound-T

This section explains how the flow-state concept was extended to implement partial evaluation during flow-graph building in Bound-T.

First note that the flow-graph building algorithm can already be viewed as partial evaluation. The entry address (initial PC value) is one input for the target program; building the flow-graph amounts to partial evaluation of the target program with respect to this input, keeping all other inputs unbound. The partial evaluation of an instruction amounts to finding the effect of the instruction on the PC, in other words finding the successor instructions.

We can extend this partial evaluation simply by *adding more concrete state components to the flow-state*. Of course, this forces us to *compute the effect of each instruction on these new flow-state components* to find the successor flow-states of the instruction.

To implement this in Bound-T the flow-state type is extended with a *data-state* component that is either null or a pointer to a *data-state object*. A data-state object models the values of program variables just before executing the node tagged with this data-state.

For this paper a data-state object is a *partial mapping of variables to values*. In other words a data-state binds some variables to known values but leaves all other variables unbound. For example, the data-state on entry to *SwHandler* from the call in *foo* could bind the variable holding the return address (the top stack word) to the value 0203 (hex) and leave all other variables unbound.

The flow-graph building algorithm is extended to handle data-states as follows. When partial evaluation is not in progress the data-state is null and the algorithm works as before. Otherwise the algorithm uses the data-state to partially evaluate the computational effects and edge conditions and uses the residual effects and conditions to update and propagate the data-state over nodes and edges.

Handling data-states while building the flow-graph
When filling a node for a given flow-state. If the given flow-state has a non-null data-state, partially evaluate the computational effect of the node on this data-state and store the *residual* effect in the node. Also create the *post-state* of the node as the given data-state updated by the residual effect: assignment of a constant binds the target variable, other assignments unbind it. The post-state models the program state *after* executing the instruction in this node. If the given flow-state has a null data-state make the post-state null too.
When adding an edge from a source node to a successor flow-state. If the post-state of the source node has a non-null data-state, partially evaluate the given edge condition on this data-state and if the result is *false* discard the edge as infeasible. Otherwise store the *residual* condition in the edge. If the successor flow-state has a specified data-state (whether null or not) use it as such (this happens when starting or stopping partial evaluation). Otherwise use the post-state of the source node but constrained by the residual edge condition: if the condition implies a known value for a variable then update the successor data-state with this binding.
When filling a DTC node. If the node has a non-null data-state then try to compute the target address from the data-state. If this succeeds (*ie*. if the DTC target depends only on variables bound to constants in the data-state) then add the corresponding (static) edge; also, if this DTC represents an exit from a switch handler then put a null data-state in the target of the new edge, to stop partial evaluation on this path.

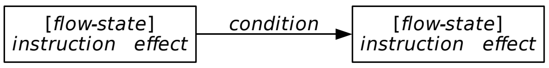
The extensions to the algorithm use existing Bound-T services for propagating constant values in flow-graphs and computational effects. New code was needed mainly for the container of data-state objects.

Most of the extensions for data-state handling are implemented in the processor-independent parts of Bound-T. The processor-specific modules only have to start and stop the partial evaluation at suitable points in the analysis. For this paper I assume that the processor-specific modules detect when a call or jump instruction enters a switch handler; at that point these modules start partial evaluation by putting the initial data-state for the switch handler in the target of the edge that enters the switch handler. Likewise, I assume that processor-specific modules detect when a DTC is an exit from a switch handler. Section 8 discusses these assumptions.

7. Example

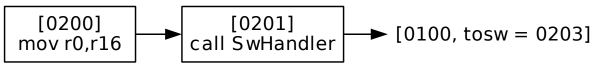
This section shows how the partial evaluation works for the *foo* function and the *SwHandler* from section 2 by a series of snapshots of the growing flow-graph.

Nodes and edges in the flow-graph are drawn as follows:



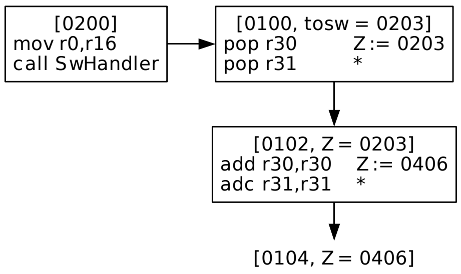
The flow-state is shown in brackets [] at the top of the box that depicts a node. The flow-state starts with the instruction address (PC) in hex, followed by the data-state bindings if any. For brevity only relevant bindings are shown. The AVR instruction is shown below the flow-state, followed by the relevant parts of its residual computational effect. An edge with no condition is unconditional (always taken). Blank nodes are shown as a bare "[*flow-state*]" with no box.

The first figure below shows the flow-graph of *foo* after the first two instructions are inserted (with null data-states) and just after detecting that the second instruction (the call) enters a switch handler. The AVR-specific modules of Bound-T have accordingly defined the successor of the call to be the first instruction in *SwHandler* (PC = 0100 hex) with a data-state that binds the return address (top of stack word, *tosw*) to 0203 hex. There is one blank node with this flow-state [0100, *tosw* = 0203].



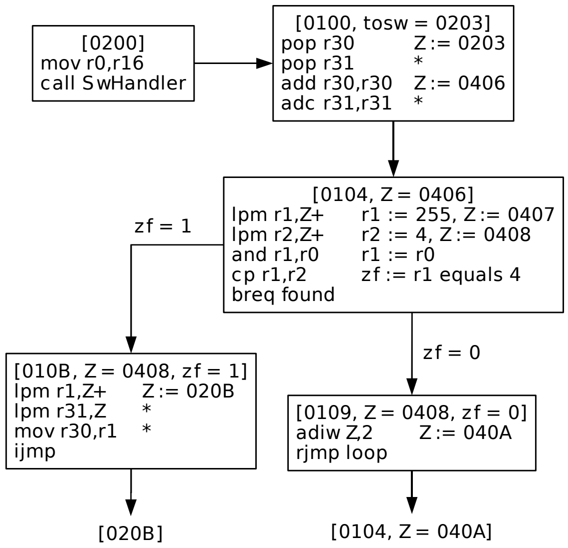
As the flow-graph grows I will compress the figures by showing several successive instructions in one box.

The next figure shows the flow-graph when the first four instructions from *SwHandler* have been inserted. Note how the partial evaluation of the *pop* instructions transformed the *tosw* binding into a binding for the *Z* pointer and how the evaluation of the *add* and *adc* instructions doubled the value bound to *Z*. (The asterisks indicate a computational effect that was combined with a preceding instruction to build a 16-bit operation from two or more 8-bit operations.) The single blank node shows that the next instruction to be added is the first instruction in the loop in *SwHandler*, at address 0104, with a data-state binding *Z* to 0406 hex, the octet address of the first entry in the switch table.

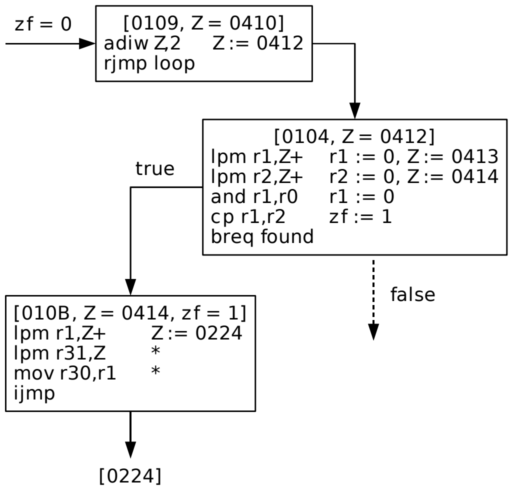


The next figure shows the flow-graph when it contains all the loop instructions and the first possible exit from the loop (for $k = 4$). On the left the loop exits when $zf = 1$. The *ijmp* DTC is resolved to a static jump because the data-state binds *Z* to 020B hex. This identifies the first case of the switch. Partial evaluation in this branch stops because the successor flow-state [020B] has a null data-state.

On the right, when $zf = 0$, the loop is about to repeat (*rjmp loop*). The successor flow-state contains the address of the loop-head (0104) which is already represented by a filled node, but it has a different data-state: *Z* is bound to 040A, not 0406 as in the existing node. The algorithm therefore creates new nodes for the second iteration of the loop. In fact the loop will be fully unrolled because the data-state binds *Z* to a different value in each iteration of the loop.



The third loop iteration is unrolled in the same way. The figure below shows the flow-graph parts for the fourth iteration which accesses the last switch-table entry (the default case) at octet address 0412 (word address 0209). The loop-repeating edge with the original condition $zf = 0$ becomes infeasible because the data-state binds zf to 1, making the residual condition *false*. This ends the unrolling and also the partial evaluation.



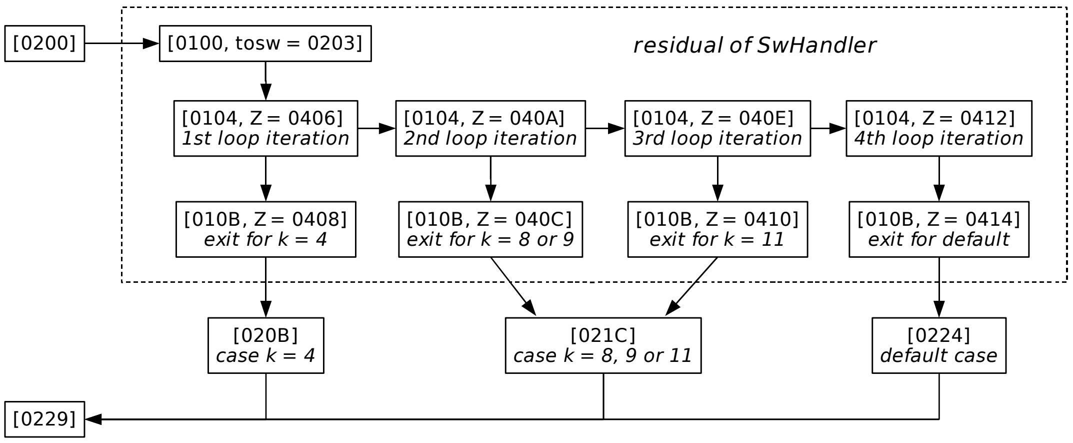
The last and largest figure, below, is an overview of the final flow-graph of *foo*. The residual form of *SwHandler* within this flow-graph is a tree of comparisons and conditional branches that explicitly models the sequence of instructions leading to each case. The edge conditions (not shown in the figure) define the corresponding index values.

8. Summary

To summarise, this method for finding the flow of control encoded in switch tables comes in three parts:

1. A flow-graph structure that can model the same instruction separately in different states (the flow-state in Bound-T).
2. A state abstraction and transfer functions for data values (the data-state and computational effects in Bound-T).
3. Means to detect entry to and exit from a switch handler in order to start/stop partial evaluation.

The first two points enable partial evaluation of machine code into parts of flow-graphs. The third point applies partial evaluation to reveal the flow of control in switch tables.



[0200]
[0100, tosw = 0203]
residual of SwHandler
[0104, Z = 0406] 1st loop iteration
[0104, Z = 040A] 2nd loop iteration
[0104, Z = 040E] 3rd loop iteration
[0104, Z = 0412] 4th loop iteration
[010B, Z = 0408] exit for k = 4
[010B, Z = 040C] exit for k = 8 or 9
[010B, Z = 0410] exit for k = 11
[010B, Z = 0414] exit for default
[020B] case k = 4
[021C] case k = 8, 9 or 11
[0224] default case
[0229]

This partial-evaluation method largely achieves the goals listed in section 3. The two main problems left are processor-specific. The first problem is to model the computational effects of all instructions so exactly that the partial evaluation of the switch handler resolves the DTCs. For example, no version of Bound-T now models the “half carry” flag for BCD arithmetic. If a switch handler uses this flag in a DTC the partial evaluation will not resolve the DTC.

The second problem is to detect when a switch handler is entered or exited. Bound-T now uses the compiler-specific identifiers of the switch handlers and works only if these identifiers are present in the symbol-table of the program. An alternative could be to use data-flow analysis or slicing as in [5-9] to detect that a given DTC is “table driven”.

Other applications of partial evaluation in WCET analysis can be imagined. For example, the *printf* function in C is notoriously difficult for WCET analysis because it is extremely data-dependent; *printf* is really an interpreter driven by the contents of the format string. Now, the great majority of *printf* calls have a constant string as the format parameter, for example:

```
printf (”%d and %f\n”, ivar, fvar);
```

Partial evaluation of such a *printf* call with respect to the constant format string should transform the interpretive loop over the format string into sequential code in the residual flow-graph. It should also transform most format-dependent conditional branches into unconditional flow of control. In the above example, the residual flow-graph should contain one *%d* (decimal integer) formatting action, followed by formatting of the constant string `” and ”`, followed by one *%f* (decimal floating-point) formatting action, followed by a new-line action. Thus, partial evaluation should resolve the format-dependent aspect of the WCET for *printf*.

In general, partial evaluation could help the context-specific analysis of any subprogram that has control-flow that strongly depends on a parameter that is often constant, or that can be resolved to a constant by other analysis.

9. Implementation experience

So far Tidorum has implemented this method of switch-case analysis in Bound-T for two target processors: Atmel AVR and Intel 8051. The compilers currently supported are the IAR and Keil compilers for the 8051 and the IAR compiler for the AVR. The method works as expected but the implementation of course showed that some extensions were necessary. This section briefly describes the extensions.

Firstly, on the AVR processor some switch handlers use single-bit load and store instructions that work with the dedicated "T" bit in the status register. As foreseen in section 8 it was necessary to extend Bound-T/AVR with models for this bit and these instructions, in order to get good residual branch conditions and to terminate the partial evaluation.

Secondly, some switch handlers call their own subroutines, for example to load the next entry from the switch table into registers, compare it to the switch index, and execute a DTC when they match. During partial evaluation all calls to these handler subroutines also have to be in-lined in the flow-graph of the subprogram that contains the switch-case statement.

Thirdly, some switch handlers implement DTC by pushing the target address on the processor stack and executing a return instruction as if the target address were a return address. Bound-T was extended to use data-flow analysis of the stack contents to separate such DTC "returns" from ordinary returns.

10. Experiments

Testing the method on several AVR and 8051 programs showed that the residual flow-graphs were less complex than could be feared. The switch handlers contain many conditional branches within loops and so loop unrolling could create quite large residual flow-graphs. However, the partial evaluation resolves many branch conditions to constants, leaving unconditional branches. This reduces the number of basic blocks in the residual flow-graphs (note that Bound-T allows unconditional branches within basic blocks) but leaves an unusually large number of instructions per basic block.

The IAR compiler for the AVR has an option that controls the kind of code generated for switch-case statements. This option can force the compiler to use in-line comparison code, or a switch table with a call to a shared switch handler, or a switch table with an in-lined form of the switch handler. Comparing the first two forms for some switch-case statements with 8-bit index values showed that the WCET for a switch handler call is usually much larger (by a factor of 10 or so) than the WCET for in-line comparison code. This is explained by the very general design of the IAR switch tables and switch handlers. The ratio would probably be less for multi-octet index-types where the in-line comparison code must be larger.

11. Related work

Earlier work on switch-case control flow [5-8] uses program slicing and constant propagation to find dense tables of addresses or jumps, indexed by the switch index. This is related to but not the same as partial evaluation. Only the "hashing form" in [8] involves run-time search in a table.

Bound-T analyses jumps through dense address tables with a combination of instruction-pattern matching, to find these jumps, and data-flow analysis (based on Presburger Arithmetic) to find the bounds of the address table. The instruction patterns in Bound-T are currently target-specific and inflexible; slicing methods and data-flow patterns as in [5-8] would be an improvement.

De Sutter *et al*. in [5] start from a "super-conservative" control-flow graph in which each DTC is modeled as an edge to a special, unique "hell node" that represents an unkown program point. The graph also has edges from the hell node to every node that could be the target of a DTC. Constant propagation then prunes away some of these hell edges. Kästner and Wilhelm present a similar top-down method [7]; however they identify address tables by their assembly-language form (lists of label identifiers), not by their usage.

The drawback of such top-down flow-graph pruning methods [5, 7] is that they first need some other method to *find* all the instructions in the program. That is impractical in the analysis of binaries for processors with instructions of different sizes, because the common executable-file formats such as ELF do not mark instruction boundaries in the memory images. Some cross-compilers even deliberately overlay instructions so that, for example, the code can jump to the second octet of a 3-octet instruction to use the last two octets as a 2-octet instruction. The bottom-up flow-graph extension methods in Bound-T and [6] work also for such processors and compilers.

Another problem with some top-down methods is that they assume that a subprogram consists of *contiguous* code. This is is false for several cross-compilers that implement optimizations such as shared subprogram epilogues where the code for a subprogram can jump into the middle of some other subprogram far away in the address space.

Tröger and Cifuentes [9] use slicing to find calls of virtual functions in the object code of C++ programs, another form of DTC. They model the computational effect of instructions in a "High-Level Register Transfer Language", HRTL, similar to the formulation in Bound-T. The static part of their analysis does not try to find the actual addresses of the callees. Instead, an HRTL interpreter dynamically executes the program and records the computed target addresses of the virtual function calls. However, the interpreter records only the executed paths, not all possible paths as in partial evaluation, so the virtual function tables may not be fully explored.

I know of no other analysis tool that creates residual subprograms in flow-graph form. However, this is much like context-specific optimization while compiling an in-lined subprogram.

The "abstract execution" method in SWEET [10] creates graphs of data-states (execution histories) but does not expand the CFG. The current SWEET input language, NIC, does not use switch tables.

12. Conclusion

I see this switch-case analysis as a small example of cooperation between the two main approaches to program analysis, the first being concrete state *enumeration* by executing the program and the second being state *comprehension* by abstracting the program. Partial evaluation represents the first approach. Bound-T uses the second approach to find loop bounds. I believe that WCET analysis would benefit from more use of state enumeration. The problem, of course, is choosing which state components to enumerate. Enumerating the states in switch handlers was an easy instance of this problem.

References

[1] G. Bernat and N. Holsti. Compiler Support for WCET Analysis: a Wish List. In *Proc. of the 3rd International Workshop on WCET Analysis* (WCET 2003), Porto, July 2003.

[2] Tidorum Ltd. Bound-T Execution Time Analyzer. *http://www.bound-t.com*.

[3] Atmel Corporation. 8-bit AVR Instruction Set. Rev. 0856D-AVR-08/02.

[4] N.D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, Vol. 28, No. 3, September 1998, pp. 480-503.

[5] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert and B. Demoen. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, June 2000, pp. 1013-1019.

[6] H. Theiling. Extracting safe and precise control flow from binaries. In *Proc. of the 7th International Conference on Real-Time Computing Systems and Applications*, Dec. 2000, pp. 23-30.

[7] D. Kästner and S. Wilhelm. Generic Control Flow Reconstruction from Assembly Code. *Proc. LCTES'02 – SCOPES'02*, June 2002, pp. 46-55.

[8] C. Cifuentes and M. Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *Proc. of the 7th International Workshop on Program Comprehension*, May 1999, pp. 192-199.

[9] J. Tröger and C. Cifuentes. Analysis of Virtual Method Invocation for Binary Translation. In *Proc. Ninth Working Conference on Reverse Engineering* (WCRE'02), 2002, pp. 65-74.

[10] J. Gustafsson, A. Ermedahl, C. Sandberg and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *Proc. of the 27th IEEE International Real-Time Systems Symposium* (RTSS'06), December 2006, pp. 57-66.