Dynamic Branch Resolution based on Combined Static Analyses

Wei-Tsun Sun, Hugues Cassé

Dynamic Branch Resolution Based on Combined Static Analyses*

Wei-Tsun Sun[1] and Hugues Cassé[2]

1 **IRIT, University of Toulouse, Toulouse, France**
`wsun@irit.fr`
2 **IRIT, University of Toulouse, Toulouse, France**
`casse@irit.fr`

Abstract

Static analysis requires the full knowledge of the overall program structure. The structure of a program can be represented by a Control Flow Graph (CFG) where vertices are basic blocks (BB) and edges represent the control flow between the BB. To construct a full CFG, all the BB as well as all of their possible targets addresses must be found. In this paper, we present a method to resolve *dynamic* branches, that identifies the target addresses of BB created due to the switch-cases and calls on function pointers. We also implemented a slicing method to speed up the overall analysis which makes our approach applicable on large and realistic real-time programs.

Keywords and phrases WCET, static analysis, dynamic branch, assembly, machine language

Digital Object Identifier 10.4230/OASIcs.WCET.2016.8

1 Introduction

The verification of critical real-time system is utterly important to ensure safety and to avoid catastrophic failures. An aspect of this verification is related to check the schedulability of the real-time programs, which requires computing the Worst Case Execution Time (WCET). To be precise, the WCET computation needs to be performed at machine code level to cope with all details of the work of the underlying microprocessor and memory system.

Implicit Path Enumeration Technique (IPET), one of the most effective approach to compute the WCET by static analysis, consists in three phases: (a) the execution path analysis, (b) the block timing analysis, and (c) the WCET estimation as a maximisation of an object function modelled as an Integer Linear Programming system (ILP). Usually the execution paths are represented synthetically by a Control Flow Graph (CFG) where vertices are basic blocks (BB)[1] and edges represent the control flow between the BB. This phase is crucial because, if some paths are missing, the obtained WCET will not be sound.

Yet, working at the machine-code level means that the analyser has to retrieve the control flow from semantically-poor instructions, as the results of translating the rich high-level language structures and of the optimisations performed by the compiler to speed up the program. The CFG may be viewed as a kind of canonical representation of the execution path independent of the high-level language. Yet, some paths are harder to recover. Usually,

* This work is supported by the french research foundation (ANR) as part of the W-SEPT project (ANR-12-INSE-0001).

[1] A BB is a sequence of instructions which are only started at the first instruction and which accepts only the last instruction as a control instruction.

the execution paths are determined by executing control instructions (such as branches) that can be executed conditionally or unconditionally. Different execution paths are the result from control instructions that modify the program counter of the microprocessor to execute one part of the code (e.g. the branch is taken) or another. Most of these instructions are *static*: the target address is obtained as a combination of the instruction program counter and of the literal operand (e.g. a constant) found in the instruction word (therefore known at analysis time). Such control instructions are used to translate selection or loop structures.

In the opposite, the target addresses of some high-level structures are results of complex computations based (a) on the current state of the program or/and (b) on the data stored in memory. For example, the C language supports the concept of function pointer meaning that the control flow depends on the data flow of the program and the program points that set this pointer. Another construct of the C language, the `switch`-case statement, may be translated and optimized as a branch whose possible target addresses are stored in a look-up table: an index is computed from the case value and used to get the corresponding index of the table. We call this type of control instruction, *dynamic*: to be analysed, they require data-flow information that is usually obtained, in turn, from an analysis of the CFG.

The contributions and the organisation of the paper

This paper proposes a new approach to determine the possible target(s) of *dynamic* control-flow instructions based on the combination of different types of analyses, the Circular-Linear Progression (CLP) [5, 11] and the k-set analyses. We are also introducing *LightSlicing*, a policy of program slicing which does not require *address analysis* for both simplicity and better performances. LightSlicing works on the machine code and (1) is relatively cheap in computation time and (2) remains precise enough to slice out the program parts that are not involved in the calculation of the control flow targets. Hence, the reduction of the analysis time for both small benchmarks and large realistic programs. We believe our solution is well-adapted to industrial applications.

The remaining of the article is organized as follows: in Section 2 we look into the problem of *dynamic* branches and expose our combined analyses approach. Section 3 shows our approach for the fast program slicing to speed-up the analysis. The experimentation and the related works are presented, respectively, in Section 4 and 5. Finally, Section 6 concludes the paper and also proposes possible extensions of our approach.

2 Dynamic Branch Resolution

This section presents the combined analysis used to resolve the targets of *dynamic* branch.

2.1 Path Analysis

The path analysis aims to provide a representation of the execution paths of the program. In this paper, we focus on the CFG representation, a graph $G = (V, E, \epsilon)$ where V is the set of BB, the set $E = V \times V$ is the set of edges, and the vertex $\epsilon \in V$ is the entry of the graph. G is built from the binary representation of the program by following the instruction flow from known entry points that may be the starting point of the program or function entries provided in the symbol table of the binary file. Usually, it is not feasible to sequentially decode the code segments because (a) they may also contains data and (b) some instruction sets have variable-size instruction words.

Algorithm 1 CFG Building.

1: $V \leftarrow \{\epsilon\}; E \leftarrow \emptyset; wl \leftarrow \{(\epsilon, i_0)\}$
2: **while** $wl \neq \emptyset$ **do**
3: $(v, i) \leftarrow pop(wl)$
4: $B \leftarrow [i]$
5: **while** $\neg$ *is_control*(i) **do** ▷ if the current instruction is not a control instruction
6: $i \leftarrow next(i); B \leftarrow B$ @ $[i]$ ▷ append the instruction to the BB
7: **end while**
8: **if** $B \notin V$ **then**
9: $V \leftarrow V \cup \{B\}$ ▷ collect the current BB
10: **end if**
11: $E \leftarrow E \cup \{(v, B)\}; wl \leftarrow wl \cup \{(B, target(i))\}$ ▷ creating the edges between BBs
12: **if** *is_conditional*(i) **then** ▷ for conditional instructions, such as BEQ
13: $wl \leftarrow wl \cup \{(B, next(i)\}$ ▷ the next instruction will be used to start a new BB
14: **end if**
15: **end while**

Algorithm 1 gives an overview of how to build a CFG. In brief, the sets V and E are built incrementally from the synthetic initial BB, ϵ and from the initial instruction i_0. A BB is obtained from continuously collecting the current instruction i until reaching a control instruction. Depending on the nature of this instruction (conditional or not), the next instruction and/or the target instruction of the branch are added to the working list wl. When wl is empty, all paths have been traversed and the CFG is complete.

Functions *is_control*, *is_conditional*, and *next* are instruction-set dependent but can be easily derived from the instruction words, which also applies for the *target*s of *static branches*. However, the *target* may also be resulted from a computation involving the state of the program: such an instruction is called a *dynamic branch*. Having dynamic branches leads to an incomplete CFG. To compute its possible targets, an analysis of the possible program state is required.

2.2 The Flow of the Dynamic Branching Analysis

Let's take the example of Fig 1 that implements a `switch`-case statement. The actual computation of the branch address is performed by instructions at addresses 0x2e260 to 0x2e268. r_3 is loaded from a byte in memory; if it is not greater than 3 (comparison at 0x2e264), it is used to compute the address $pc + r3 \times 4$ that points to an entry of the subsequent table that contains the actual targets of the branch. Hence, (a) the calculation of the possible targets of *dynamic* control instruction 0x2e268 is feasible and (b) we need to use a value analysis to evaluate its components. Another outcome of this example is that we need to apply value analyses to partial CFG and we need also to repeat the analysis until we get the whole CFG: on the first path, the CFG until the *dynamic* control instruction is obtained and values for *pc* and $r3$ are estimated and enable the calculation of *dynamic* control instruction targets; in the second path, the CFG is extended and, possibly, new *dynamic* control instruction are discovered and so on.

In our approach, the flow to identify the targets of dynamic branches consists of the following steps: (1) First we perform a CLP analysis [5, 11] to obtain the possible values of the registers and the memory addresses. The range of the represented values is over-approximated, i.e. this includes both of all possible values (values which may present during

```
    Addrs   Content    Assembly (ARM)
 1  2e250   e59f2050   ldr    r2, [pc, #80] ; load for r2
 2  2e254   e51b3008   ldr    r3, [fp, #-8] ; load for r3
 3  2e258   e0823003   add    r2, r3, r2 ; r2 used by some cases
 4  2e25c   e59f3264   ldr    r3, [pc, #612]
 5  2e260   e5d33000   ldrb   r3, [r3]
 6  2e264   e3530003   cmp    r3, #3
 7  2e268   979ff103   ldrls  pc, [pc, r3, lsl #2]
 8  2e26c   ea000075   b      2e448 ; address of the default case
 9  2e270   0002e280   ; target address for the 1st case
10  2e274   0002e2dc   ; target address for the 2nd case
11  2e278   0002e364   ; target address for the 3rd case
12  2e27c   0002e3c4   ; target address for the 4th case
13  2e280   e59f3254   ldr    r3, [pc, #596] ; the first case
```

Figure 1 Example of the switch code in ARM's assembly.

the program execution) and spurious values (which are included due to the analysis because of the performed abstraction). (2) A k-set analysis is then applied to gather the concrete values of the registers and the memory addresses. (3) The dynamic branch resolution is carried out to find the targets of the control instructions. If new targets of a control instruction are found, the CFG of the program will be updated with newly added code segments and the analysis will restart from the step (1). The analysis terminates once reached a fix-point such that no more new targets are added.

2.3 CLP analysis and its drawbacks

The CLP analysis makes use of abstract interpretation [6] with the trade-off of (1) having the better performance, especially when performing analysis on loops, and (2) the accuracies of the analysis, for example the strategy for performing widening.

A range of integers can be represented in the format of CLP, and we call the represented range a CLP value. To differentiate, we use *sub-values* to call the integers within a CLP value. Each CLP value is a triple (b, δ, m) representing set $\{b + \delta i \mid 0 \leq i \leq m\}$: b is the starting integer, δ the amount of difference between integers and m the number of integers within a CLP representation. For example, to represent a set of integers 2, 4, 10 in CLP, we will have base = 2, delta = 2, and multiple = 5, such that the CLP value will cover the set 2, 4, 6, 8, 10. Therefore, one may consider that in the domain of CLP, the set of sub-values is presented in an *over-sampling* manner, i.e. in order to include all the possible (in this case 2, 4, and 10) values, some spurious values (6 and 8 here) are included.

The over-sampling behaviour of CLP is for the sake of soundness but this can also bring unwanted behaviours in the analysis. We use Figure 1 to demonstrate this. Figure 1 contains the ARM instructions typically generated from a switch case and perform as the follows: (1) storing the result of some calculation to the register r_2 (lines 1 to 3), which will be used later; (2) lines 4 and 5 provides the switch-index number used to calculate the target address of the switch-cases; (3) the values of the switch-index, infers as the number of the possible targets, are limited by line 6 so that line 7 will only execute if the switch-index falls in the desired range; and (4) if none of the case is chosen, the default case falls through (line 8). By looking at the lines 6 and 7, we know that the index (stored in r_3) falls between the range 0 to 3. The target addresses to load is calculated as $pc + r_3 \times 4$, which are stored between the

address 0x2e270 to 0x2e27c (lines 9 to 12). The CLP representation of these target addresses is thereby of base = 0x2e280, delta = 0x4, and multiple = 0x51. This indicates that there could be 82 (number of the multiple plus one) potential targets which is a huge difference from the actual amount of the possible targets (which is 4).

2.4 k-set analysis

To overcome the drawback of CLP abstraction, we use a k-set analysis [4]. In contrast with the CLP, the values in the k-set analysis are in the form of sets which size is bound to k values. If we get a set bigger than k, it is approximated to $\top$ (any possible value): this property avoids too long or endless analysis looping to reach a fix-point. The k-set analysis is only slightly better than a constant propagation because it usually does not cope well with most of variable behaviour (often linear): the analysis time would become excessively large. Yet, the branch target addresses are not linear and to avoid the over-sampling problem, they need to be stored as an explicit set and k-set is a good candidate to represent them.

Let $\widehat{S} \subseteq 2^{\mathbb{N}}$ to be the set of k-set values that abstracts concrete value as set over $\mathbb{N}$. The abstraction, $\alpha : \mathbb{N} \to \widehat{S}$ is quite simple: $\forall n \in \mathbb{N}, \alpha(n) = \{\ n\ \}$. It is easy to extend a function $f : \mathbb{N} \to \mathbb{N}$ to $\widehat{f} : \widehat{S} \to \widehat{S}$ by just applying the concrete function f to each element of the input set:

$$\forall\ a \in \widehat{S},\ \widehat{f}(a) = \{\ f(e) \mid e\ \in\ a\} \tag{1}$$

Likely, an abstraction of functions with an arity bigger than 1 may be built by a Cartesian product. Yet, if the size of the resulting set is bigger than k, the resulting value is approximated to $\top$ (the abstraction of any possible value). In the static analysis, in order to reduce the number of paths to explore, we often use a join operator $\sqcup$ to combine together values when several paths of the CFG join. Its definition for k-set is given in Eq. 2: the set union $\cup$ is mainly used while the resulting set size is lower or equal to k. Otherwise, the result is $\top$.

$$\forall\ a,\ b \in\ S,\ a\ \sqcup\ b = \begin{cases} \top & if\ \ |a\ \cup\ b| > k \\ a\ \cup\ b & else \end{cases} \tag{2}$$

$$\forall\ a,\ b \in\ S,\ a\ \nabla\ b = \begin{cases} a & if\ a = b \\ \top & else \end{cases} \tag{3}$$

Finally, to speed up the convergence of paths containing loops, a widening operator $\nabla : \widehat{S} \times \widehat{S} \to \widehat{S}$ is useful. As the usual k-set implementation exhibits very poor performances, because of the number of generated values in a loop context, we use a stringent implementation of ∇ in Eq. 3: if both operands are the same the result is this value, else the $\top$ value is returned. This definition works well with purpose of our k-set analysis: the code addresses are rarely, maybe never, the result of a computation and even less the result of a loop computation. They are either read from the memory or computed from the *pc* register. Therefore, we want to get rid as soon as possible of values which are not instruction addresses.

However, this widening operator quickly leads to a lot of values approximated to $\top$. Usually, this is not an issue as we are not interested by most of computed data computations except when this value is the address handled by a store instruction: in this case, a $\top$ address would touch the whole memory. This means that all information collected by the analysis about the memory is scratched and lost. In turn, this would negatively impact the remaining

of the analysis. This is why the k-set analysis is useless alone: it is combined with another more precise value analysis (like CLP) such that, when an important value is required (address to load from, address to store to) and approximated by $\top$ in k-set, the matching CLP value is used instead and this usually leads to a much more precise analysed value.

Applying this method to the example of Fig. 1, the instruction at 0x2e260 may produce, for register r_3, the $\top$ value for k-sets and $(0, 1, 255)$ for CLP. If the condition `ls` of the comparison at 0x2e264 holds, r_3 becomes $\{0, 1, 2, 3\}$ for k-sets thanks to the CLP value $(0, 1, 3)$. From this, the address accessed at instruction 0x2e268 is $0x2e270 + \{0, 1, 2, 3\} \times 4$ and results in $\{0x2e270, 0x2e274, 0x2e278, 0x2e26c\}$. These are the exact set of addresses stored in the table used to translate the `switch`-case obtained by loading the words at the addresses provided by the k-set value.

3 Dynamic Branch and Real-Time Application

The approach presented in the previous section is effective but expensive to apply to a complete program. Therefore, we expose here a slicing method to speed up these analyses.

3.1 Analysis of the Whole Program

The functionality of real-time systems are often divided into tasks. The execution of the tasks are scheduled statically in the event loop or dynamically with the help of a real-time operating system. A task can be stand-alone, i.e. performing its functionality without depending on the outcome of the other tasks. On the other hand, a task might require the results of the others, through task communications [12]. The communication between the tasks relies on mechanisms such as globally shared variables and pointers, where a task writes to a variable and it is read by other tasks.

When analysing solely a task which reads from a globally shared variable, there will be no assumption made to the value of such variable, i.e. the writes to the variable are outside of the analysed task. Also, as stated in [7], to have a safe analysis (where all the possibilities are considered) of the function pointers, it is required to perform the analysis on the program as a whole. Indeed, a function pointer called in one task may be used by another task.

3.2 LightSlicing – a Smart and Effective Slicing Approach

To have safe results, tasks communicating with each other shall be analysed together. It is obvious to see that the complexity of the analysis grows as the number of tasks grows. Even though the amount of instructions to analyse grows, it can also be seen that some codes/instructions do not have influence over the results of the analysis. In this case, the technique *program slicing* [14] can be applied to remove the uninteresting codes. For example, in Figure 1, the lines 1 to 3 (as well as lines 8 to 13) do not affect the outcome of the branching, and they can be sliced away.

The problem now is that slicing a program is a costly operation in analysis time requiring data flow analyses and several graph constructions and are even more time-consuming operations applied to machine language. To be effective, the slicing time must not exceeds the gain in analysis time of the *dynamic* control instructions. It already exists an approach to perform fast slicing on machine code [10]. However, while the slicing is performed, the working elements (registers and memory locations of interest) continue to grow even if their content is no more relevant, which leads to keep unnecessary parts of the program. It is not efficient enough to significantly reduce the program CFG size.

Algorithm 2 LightSlicing algorithm.

1: $K \leftarrow I$
2: $wl \leftarrow \{(v,\ USE(i))\ /\ i \in I \wedge i \in v \wedge v \in V\}$ ▷ consider BB containing instructions of I
3: **while** $wl \neq \emptyset$ **do**
4: $(v, we) \leftarrow pop(wl)$
5: $WE(v) \leftarrow we$
6: **for all** $i \in reverse(v)$ **do** ▷ examine instructions of BB in reverse order
7: **if** $DEF(i) \cap we \neq \emptyset$ **then** ▷ if the instruction define a useful register
8: $K \leftarrow K \cup \{i\}$ ▷ the instruction is kept
9: $we \leftarrow we \setminus DEF(i) \cup USE(i)$ ▷ its used registers are now interesting
10: **end if** ▷ yet its defined registers are no more interesting
11: **if** $i = FIRST\_INST(v)$ **then** ▷ when reaching the first instruction
12: $wl \leftarrow wl \cup \{(w, we \cup WE(w)) \mid w \in pred(v) \wedge we \setminus WE(w) \neq \emptyset\}$
13: **end if** ▷ the predecessors are added to wl if the fixed-point is not reached
14: **end for**
15: **end while**

Hence, we introduce our slicing approach on binary code: the *LightSlicing*. It adapts the conventional *DEF and USE* approach used to build DU- or UD-chains described in [8]. The *DEF* and the *USE* give up a set of elements (i.e. registers and/or memory addresses) that an instruction writes a value to and reads a value from, respectively. Conventional program slicing based on this will start with a set of elements of interest, which we call *working elements, or we*. During the process, the instructions *inst* that write to any of these elements, donated by using the *DEF(inst)*, will be kept. Then, the redefined element(s) are removed from the working elements; while the required elements, i.e. the elements to read which are obtained by using the *USE(inst)*, are added to the working element. In general, identifying the registers in the *DEF* and *USE* is straight-forward. In contrast, obtaining the memory addresses can be complicated. For example, a memory address to read (or write) may be stored in a register which value is decided at run-time. Therefore, the help of *address analysis* is required. For a large program, to have a coherent result of the address analysis, the whole program must be taken into account, which may lead to an expensive computation time. To achieve better performance, LightSlicing does not require the address analysis: the whole memory is considered as a single register: we loose in precision but hope to gain a lot in speed. LightSlicing is applied just before each iteration of the dynamic branching resolution to avoid unnecessary computations and hence the speed-up. The details of LightSlicing is shown in Algorithm 2: the result is K, the set of instructions to keep while the initial set of interesting instructions is I.

4 Experiments and Findings

We carried out the experiments over two sets of benchmarks: the Mälardalen benchmarks [9], and the realistic industrial Engine Management System from Continental Corporation (which consists of 7 tasks, 184 KLOC). The experiments were carried out on Intel i7-4810MQ 2.8 GHz with 32 GB of RAMs. Because we are mainly interested in the detection of the unknown target addresses, due to the `switch`-cases and calls on function pointers, we only experimented with *cover*, *duff*, and *lcdnum* from the Mälardalen benchmarks. The results of the analysis times and speed-ups (due to applying LightSlicing) are shown in the Figure 2. Since the examples from the Malardalen benchmarks are much smaller than the industrial

case, we multiply the analysis time with 1000 to make them visible in Figure 2. Because we are interested in the impacts due to the application of the slicing, we performed the analyses for three different cases: (a) the analysis without program slicing, (b) applying program slicing with address analysis, and (c) the analysis with *LightSlicing*. We are able to resolve all the dynamic branches of the Mälardalen benchmarks, but 92% for the industrial example. It is mainly due to that the slicing of the industrial application makes irreducible loops that are not completely handled by OTAWA framework, hence we can not perform the analysis on the whole program but on its individual tasks.

Figure 2 shows the distributions of time taken by each analysis. For the analysis without slicing, we use the real execution time (in micro-seconds) for the vertical axis. To compare the performance for the analyses that take advantage of the slicing, we use the speed-up as the vertical axis. The speed-up is calculated by Equations 4. When the value of the speed-up is less than one, this means that the analysis runs slower than the non-slicing approach: we use red lines in the figure to represent this boundary.

It shows that the analysing times decrease drastically, with the average of 7.30 times, and the maximum of 33.37 times of speed-up, when applying LightSlicing. From (a) we can see that the CLP analysis takes the majority of the analysis times because it is more complex than the k-set analysis. Since the address analysis used in (b) is implemented within the CLP analysis, which leads the CLP analysis to take more proportion in the analysis. Because the size of the CFGs are reduced, the time spent on k-set analysis is also reduced. We can also observe that the slicing does not impact overall performance heavily. In (c), both of the CLP and k-set analyses are performed upon the reduced CFGs, as the result from applying LightSlicing and hence the reduction of the analysis time. LightSlicing takes more proportion in the analysis because it works on the full CFGs, however it reduces the analysis by large amounts for all cases.

It also shows that having slicing with address analysis may have negative impact on the performance for smaller examples, whose speed-ups are less than one (in the grey-out area). This is because the overhead from the address analysis can not be compensated by the time saved due to slicing. We avoid the address analysis in LightSlicing and obtained the improvements up to 33.37 times faster. LightSlicing works particularly well on larger codes which justifies its use in realistic and real-time applications.

$$Speedup = \frac{T_{non\text{-}sliced}}{T_{sliced}} \tag{4}$$

5 Related Works

Building CFGs from binaries is a recurrent issue for static analysis of binaries, for making smart debugger or for reverse-engineering programs [15]. Theiling, in [13], proposes a multi-instruction set generic framework to extract CFGs from binaries. He identifies the issues in the determination of *dynamic* branches but no solution is provided.

Bardin et al. in [4] use variable-precision k-sets to compute the targets of *dynamic* branch instructions. The k determination is variable for each handled value and adjusted according to the need of precision, focused in this case, on the set of possible targets of a branch. The experimentation on an industrial application (21 kloc of C) exhibits relatively long computation times (in tens of minutes). Moreover, the authors do not address the problem of memory loss due to imprecise k-set values.

In [3, 1, 2], Balakrishnan and Reps present a complete method to perform data flow analysis, resolve *dynamic* branches and extend the CFG in an incremental way. The approach

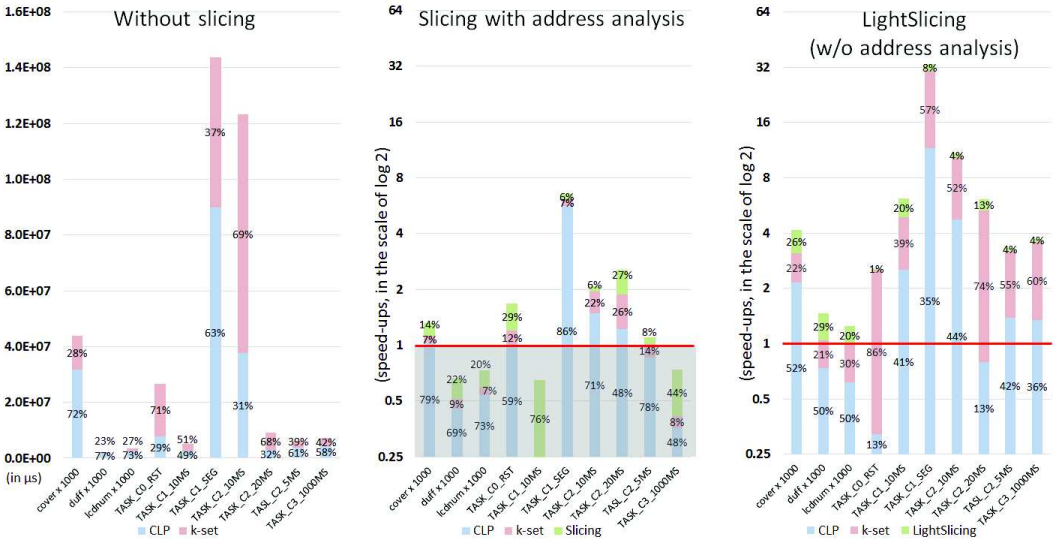

Figure 2 Execution comparisons of the dynamic branching analyses.

is quite integrated and therefore relatively costly in analysis time. Yet, they use, as abstraction of values, a form close to CLP and hence suffers from the over-sampling problem.

In a recent article [7], Holsti et al. experiments and compares several common value analyses (including CLP) to resolve the target of *dynamic* control instructions. They identify several shortcomings in these classic value analyses (particularly the over-sampling problem of CLP) and the requirement to analyse the whole application for function pointers resolution inside tasks of a real-time system. None of the experimented analyses overtakes the others but their limitations are highlighted.

6 Conclusion

In this paper, we have presented an approach to resolve *dynamic* control instructions. The approach is based on a usual value analysis (CLP in this case but this could be another value analysis) used to help the k-set analysis. The k-set analysis enables us to precisely preserve the possible target addresses of branch instructions. In the case of function pointers, it is shown in [7] that a whole analysis of the application is needed. As this analysis may be time-consuming, we propose a fast slicing method which works on the machine codes and speed up the subsequent value analyses.

The experiments conducted on a subset of Mälardalen benchmarks and on a real industrial application shows good but not perfect results. The main cause of unresolved *dynamic* branches is the precision of the auxiliary value analysis (CLP): in future works, we plan either to improve our CLP analysis, or to replace it with a better value analysis, or to combine together several value analyses providing different aspects of the program values.

Then, although we have achieved very good analysis time, particularly on the real industrial application, it remains some room for improvement: at each step of the analysis, CLP and k-set analyses are wholly re-computed while only a small part of the CFG is changed. A good effect of Abstract Interpretation based analyses is that the detailed behaviors of each instruction/BB are simplified and abstract states are used to present the effects constituted by each part of the CFG. The changes in the abstract states propagate throughout the CFG

and new paths of propagation could be formed according to the evolution of the abstract states. If the propagation of new paths does not contribute a lot of time to the overall analysis, we expect substantial speed-ups from the incremental calculation of the CLP and the k-set analyses for the calculation of *dynamic* branches targets.

References

1 G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. *Compiler Construction: 14th International Conference, CC 2005*, chapter CodeSurfer/x86 – A Platform for Analyzing x86 Executables, pages 250–254. Springer Berlin, 2005.

2 G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 2732–2733. Springer Berlin, 2004.

3 G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. *Verified Software: Theories, Tools, Experiments*, chapter WYSINWYX: What You See Is Not What You eXecute, pages 202–213. Springer Berlin Heidelberg, 2008.

4 S. Bardin, P. Herrmann, and F. Védrine. *Refinement-based CFG reconstruction from unstructured programs*, pages 54–69. Springer, 2011.

5 Hugues Cassé, Florian Birée, and Pascal Sainrat. Multi-architecture value analysis for machine code. In *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, pages 42–52, 2013. doi:10.4230/OASIcs.WCET.2013.42.

6 P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT*, pages 238–252. ACM Press, 1977.

7 Niklas Holsti, Jan Gustafsson, Linus Källberg, and Björn Lisper. Analysing switch-case code with abstract execution. In *15th International Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, pages 85–94, 2015. doi:10.4230/OASIcs.WCET.2015.85.

8 S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

9 WCET Project Mälardalen University. Benchmarks. URL: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

10 C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster WCET Flow Analysis by Program Slicing. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED*, pages 103–112. ACM, 2006.

11 R. Sen and Y. N. Srikant. Executable Analysis with Circular Linear Progressions. Technical Report IISc-CSA-TR-2007-3, Computer Science and Automation Indian Institute of Science, February 2007.

12 A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

13 H. Theiling. Extracting safe and precise control flow from binaries. In *Proc. of 7th Conference on Real-Time Computing System and Applications*, 2000.

14 M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.

15 W. Yin, L. Jiang, Q. Yin, L. Zhou, and J. Li. A control flow graph reconstruction method from binaries based on XML. In *Computer Science-Technology and Applications, 2009. IFCSTA'09. International Forum on*, volume 2, pages 226–229, Dec 2009.