BEST: a Binary Executable Slicing Tool

Armel Mangean[1], Jean-Luc Béchennec[2], Mikaël Briday[3], and Sébastien Faucou[4]

1 **École Centrale de Nantes, IRCCyN UMR 6597, Nantes, France**
2 **CNRS, IRCCyN UMR 6597, Nantes, France**
3 **Université de Nantes, IRCCyN UMR 6597, Nantes, France**
4 **Université de Nantes, IRCCyN UMR 6597, Nantes, France**

Abstract

We describe the implementation of Best, a tool for slicing binary code. We aim to integrate this tool in a WCET estimation framework based on model checking. In this approach, program slicing is used to abstract the program model in order to reduce the state space of the system. In this article, we also report on the results of an evaluation of the efficiency of the abstraction technique.

1998 ACM Subject Classification C.3 Special-Purpose and Application-Based Systems, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging

Keywords and phrases Program Slicing, Binary Code Analysis, WCET Analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2016.7

1 Introduction

In the recent years, several works have explored techniques to statically estimate the worst-case execution times (WCET) of a program using model checking [10, 6, 4]. The most important issue encountered when using model checking to perform WCET estimation is the exponential size of the state space that must be exhaustively explored during the analysis [20]. To fight this problem, state-of-the-art model checking tools for dense timed systems such as Uppaal [14] use powerful symbolic algorithms and data structures. It has been shown that it allows to deal with small but realistic instances of the WCET problem [10, 6]. It is expected that model checking technology will continue to improve in the coming years, widening the range of instances that can be solved.

A different and complementary direction to deal with the explosion of the state space consists in abstracting the models of the programs [4, 3] or the models of the hardware components [5]. The idea is to remove the information which does not impact the WCET. This work follows this direction, with a focus on the models of programs. In the continuation of prior work [4] we explore the use of program slicing [19] at the level of the binary code to abstract the model of the program.

In this paper we introduce Best, a program slicer for binary code. We describe its architecture and implementation. We explain the interface between Best and Harmless [12], a toolchain built around a Hardware Architecture Description Language (HADL). Thanks to this interface, the core of Best is independent from the target instruction set of the binary code. We also use Best and the Mälardalen benchmarks to show how to compute abstract model of programs and report on the benefits that could be reached with this approach.

The paper is organized as follows. In Section 2 we give an overview of related works. In Section 3 we outline an approach to the estimation of WCET with model checking. In Section 4 we provide a summary of program slicing. In Section 5 we describe the

implementation of Best and its interface with Harmless. In Section 6 we report on an evaluation of the abstraction approach using Best. In Section 7 we conclude the paper.

2 Related works and contribution

In the context of static WCET analysis, program slicing has been explored [17, 15, 4]. Program slicing is mostly used to accelerate the static analysis of flow facts [17, 15]. Our goals are different, as well as the slicing technique. In contrast to our work, program slicing is applied to structured programs at the source code level (or intermediate code level [17]). Our tool works at the binary code level. As a positive side effect, it is independent from both the programming language and the compiler. To our best knowledge, there is no established tool for slicing non-x86 binary code and Best aims at filling this gap.

Our work is the continuation of previous work by Cassez and Béchennec [4]. In this work they propose a prototype tool based on the classical dataflow equations approach [19] that computes slices for ARM-based binary code. Unlike that, our tool is independent from the target instruction set thanks to its interface with the Harmless toolchain. Furthermore, our tool is based on a state-of-the-art graph-based approach [13]. We also provide an evaluation focused on the benefits of the program abstraction technique.

Brandner and Jordan [3] propose a graph pruning technique to increase the precision of static WCET estimation. Branches of the Control Flow Graph (CFG) are pruned based on the criticality of their basic blocks. The criticality is defined as the normalized duration of the longest path passing through the block [2]. According to the authors this approach is akin to "program slicing in the time domain". Based on this pruning approach, a refinement based WCET calculation meta-algorithm is proposed. We do not address a full WCET analysis in this paper. However, their technique could be combined with our approach to improve WCET calculation. Such a combination should allow to further abstract the program in order to deal with state space explosion.

3 WCET estimation using model-checking

WCET estimation can be reduced to a reachability problem in a network of timed automata [4]. The Uppaal tool that supports timed automata extended with bounded integer variables is used to build the models, and to solve the reachability problem.

A model of the hardware is built where each architectural feature (pipeline(s), cache(s), bus(es), memory, ...) is modeled by one or more timed automata. These automata are synchronized through *channels* to model the actual hardware behavior. For instance the automaton modeling the fetch stage of a pipeline is synchronized with the automaton modeling the instruction cache which is synchronized with the automaton modeling the bus and so on. The timings of the hardware are modeled by guards and clocks on some edges of the automata. A simple model of a memory controller could be the timed automaton of Figure 1. Notice that this model accounts only for the timing.

A model of the program is automatically built from the binary code. In this model, each location corresponds to an instruction. An edge leaving a location corresponds to the execution of the instruction. For conditional branches, two edges leave the location according to the behavior of the branch (taken or not taken). Each edge is synchronized with the automaton that models the instruction fetch so that it may only be fired if the hardware fetches a new instruction. Memory locations are updated according to the semantics of the instruction and to its advance in the pipeline. The model of the program has an initial state,

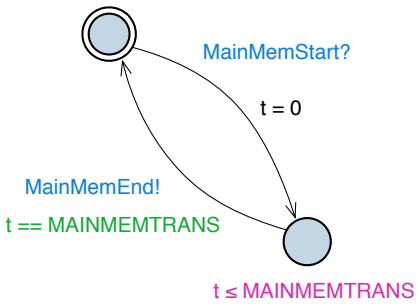


Figure 1 Simple modeling of a memory using UPPAAL. In the initial state (on the top left) the memory waits for an access (MainMemStart synchronization channel). When the access is requested, clock t resets and the automaton remains in the bottom right state until t reaches the MAINMEMTRANS value. Then the memory returns to the initial state and notifies the end of the memory access (MainMemEnd synchronization channel).

I, that corresponds to the entry point of the program and a final state, F, that corresponds to the point at which the WCET has to be computed.

At last, a global clock x is used to measure the time. It is initialized at 0.

The WCET is then the largest value, $max(x)$, of x when F is reached. $max(x)$ can be computed with a model-checker and the following reachability property $R(T)$: "Is F reachable with $x \geq T$?". If $R(T)$ is true and $R(T+1)$ is false then T is the WCET of the program.

This approach is modular since the hardware and software models are built separately and the hardware model does not depend on the software to check. No assumption is made about the structure of the binary code generated by the compiler and the model of the program is built automatically without need for annotations

Modeling the values stored in memory

Data stored in memory and registers – called a *location* in the remaining of the paper – and used by the program can be either included in or abstracted away from the model. Each location included in the model is associated with a bounded variable. When the program accesses a location, the timing is computed by the models of the hardware. If the location is included in the model, the associated variable is also read / written. If the location is abstracted away, the data to be written is discarded and any read access returns the special value $\perp$.

On the one hand, every location included in the model adds a dimension to the state of the system and thus contributes to the growth of the state space. On the other hand, $\perp$ values can lead the model checker to explore paths that are not in the systems when they impact a conditional branch instruction. Thus the problem is to automatically compute the minimal set of locations that impact on the control flow of the program and that should be included in the model. In this paper, we focus on this problem.

4 Program Slicing

4.1 Notations

Let $\mathcal{I}$ be a finite set of instructions. Let $\mathcal{L}$ be a totally ordered finite set of labels. A program P is a finite subset of $\mathcal{L} \times \mathcal{I}$ such as $\forall (l,i) \in P, \quad (l,i') \in P \leftrightarrow i = i'$. We denote $\mathcal{V}$ the set

of variables of P. If we consider the program in Figure 2a, $\mathcal{I}$ is the subset of instructions of the 32 bits PowerPC instruction set used by the program, $\mathcal{L}$ is the set of memory addresses aligned on 4 bytes boundaries in the range $[3000, 3034]$ and $\mathcal{V}$ is the set of memory locations explicitly or implicitly used (i.e. $\{r1, r3, r8, r9, r10, lr, ctr\}$).

A basic block is a sequence of instructions of P with one entry point, its first instruction, and one exit point, its last instruction. A basic block is maximal if it is not contained in any other basic block. Let $G_P = \langle V_P, E_P, u_{G_P}, v_{G_P} \rangle$ where V_P is the finite set of maximal basic blocks of P and $E_P \subset V_P \times V_P$ is such that there is an edge between $v_1 \in V_P$ and $v_2 \in V_P$ if and only if the first instruction of v_2 can be executed immediately after the last instruction of v_1 in P. $u_{G_P} \in V_P$ and $v_{G_P} \in V_P$ are respectively the entry block and the exit block of P. Then G_P is the CFG of P.

4.2 General overview

Program slicing has been introduced by Weiser [19]. Weiser defines a program slice as an executable program that is obtained from the original program by deleting zero or more statements, computing the same values for a given subset of variables of the program. He claims that a slice corresponds to the mental abstractions that people make when they are debugging a program. The original formulation of program slicing proposed by Weiser is based on iterative solutions of data-flow equations. Ottenstein and Ottenstein [16] were the first to redefine slicing as a reachability problem in a dependence graph representation of a program. They use a Program Dependence Graph (PDG) [8] for static slicing of single-procedure structured programs. Efforts have been made to extend this approach to unstructured programs [1, 13] and multiple-procedure programs [11, 13]. More details on the topic can be found on the survey by Tip [18].

We consider in this section a toy example to highlight the slicing method. It is a simple program that computes iteratively the first 30 values of the Fibonacci sequence ($F_n = F_{n-1} + F_{n-2}$, with $F_0 = 1$ and $F_1 = 1$). The code targets the PowerPC instruction set. The program works as follow:

- The `_start` label (Figure 2a, line 1) is the program entry point. It gets minimal startup code that initializes the stack pointer $r1$ and calls the `main` at 3010 (Figure 2a, line 7). If the `main()` function returns, it enters in an infinite loop (Figure 2a, line 5) ;
- Figure 2a, lines 8 to 11 initialize the sequence. The loop is controlled by the dedicated *ctr* counter register ;
- Figure 2a, lines 13 to 16 are the instructions in the loop. $r9$ and $r10$ stores respectively the current and the last value and are used to compute the next value (in $r3$).

A slice is computed with regards to a slice criterion $\mathcal{C} = \langle l, v \rangle$ with $l \in \mathcal{L}$ a label and $v \subseteq \mathcal{V}$ a set of variables. So, if we consider the program in Figure 2a and the slicing criterion $\langle 3030, \{ctr\} \rangle$, i.e. the value of register *ctr* when the instruction pointer contains the address 3030, we obtain the slice shown in Figure 2b. Indeed, the instruction `bdnz 3024` at address 3030 (Figure 2b, line 16) implicitly modifies the register *ctr*, *ctr* is set by `mtctr r8` at 3018 (Figure 2b, line 10) and $r8$ is set by `li r8,29` at 3010 (Figure 2b, line 8).

To compute a slice in binary code, we need to handle arbitrary control flows (as opposed to control flow of structured programs) and inter-procedurality. In our use case, we must also exclude the techniques that change the order of the instructions. Given all these constraints, we have to use slicing techniques based on graph manipulations [13].

This approach is based on the computation of several graphs. The first one is the CFG of the program. Figure 3a gives the CFG of `fibcall-O2.elf`. Then the Data Dependence

```
1  00003000 <_start>:
2       3000:   li     r1,1            ;r1  <- 1
3       3004:   ori    r1,r1,49296     ;ri
   <- r1 | 49296
4       3008:   bl     3010            ;call main
5  0000300c <loop>:
6       300c:   b      300c            ;branch
7  00003010 <main>:
8       3010:   li     r8,29           ;r8  <- 29
9       3014:   li     r10,1           ;r10 <- 1
10      3018:   mtctr  r8              ;ctr <- r8
11      301c:   li     r9,1            ;r9  <- 1
12      3020:   b      3028            ;branch
13      3024:   mr     r9,r3           ;r9  <- r3
14      3028:   add    r3,r9,r10       ;r3
   <- r9+r10
15      302c:   mr     r10,r9          ;r10 <- r9
16      3030:   bdnz   3024            ;ctr--,
17                                     ;branch if ctr!=0
18      3034:   blr                    ;return
```

(a) Dump of `fibcall-O2.elf`.

```
1  00003000 <_start>:
2       3000:   --     --
3       3004:   --     --
4       3008:   --     --
5  0000300c <loop>:
6       300c:   --     --
7  00003010 <main>:
8       3010:   li
   r8,29
9       3014:   --     --
10      3018:   mtctr  r8
11      301c:   --     --
12      3020:   --     --
13      3024:   --     --
14      3028:   --     --
15      302c:   --     --
16      3030:   bdnz
   3024
17
18      3034:   --
```

(b) Slice for $\mathcal{C} = \langle 3030, \{ctr\}\rangle$.

Figure 2 Dump and slice of a binary executable.

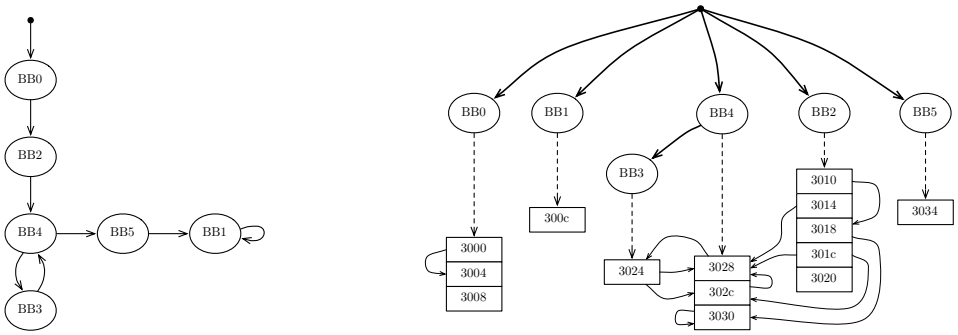
Graph (DDG) and the Control Dependence Graph (CDG) are computed from the CFG. The DDG captures data dependencies between instructions. Its nodes are the instructions of P. There exists an edge between two nodes of the DDG when the source node does a reaching definition of a memory location used by the target node. The CDG captures control dependencies between basic blocks. Its nodes are the maximal basic blocks of P. There exists an edge between two nodes of the CDG when the source node determines whether the target node is executed or not.

After the DDG and the CDG, the next graph is the Program Dependence Graph (PDG) [8]. It is built by merging the DDG and the CDG. Node sets of the DDG and the CDG being disjoint (nodes are instructions in the DDG and maximal basic blocks in the CDG), the PDG gets its consistency from special edges that represent the belonging of a set of instructions to a basic block. In summary, the PDG captures the belonging of set of instructions to basic blocks, data dependencies at instruction level and control dependencies at basic block level. Figure 3b gives the PDG of `fibcall-O2.elf`.

If P does not contain procedure calls, or if these calls are "inlined" when the CFG is built, it is possible to compute slices on the PDG. The slice corresponding to a given criterion is obtained by performing a backward reachability analysis. The slice is initialized with the slice criterion. When an instruction in the slice is the target of a data dependence edge, the source instruction is added to the slice. When an instruction in the slice belongs to a basic block which is the target of a control dependence edge, the last instruction of the source basic block is added to the slice. This procedure is iterated until a fixpoint is reached.

In Figure 3b, dashed, bold and solid edges represent respectively the belonging of a set of instructions to a block, a control dependency between two basic blocks, and a data dependency between two instructions. Considering once again the program in Figure 2a and the slicing criterion $\langle 3030, \{ctr\}\rangle$, we obtain the slice shown in Figure 2b. Indeed, the backward reachability analysis shows that the instruction at address 3030 has a data dependency with the instruction at 3018 which has also a data dependency with the instruction at 3010 and the basic block $BB2$ has no control dependency apart from the entry point.

Slicing the PDG is suboptimal for programs with procedure calls [13]. To overcome this limitation, inter-procedural slicing techniques use a fourth graph, the System Dependence Graph (SDG) [11]. To build the SDG, in a first step, the PDG of each procedure must be built. In a second step, these PDGs are connected with call, parameter-in and parameter-out edges to account for procedure calls and parameters passing. The slicing algorithm on the SDG is based on two backwards analyses similar to the one used for the PDG. The first



(a) CFG of `fibcall-O2.elf`. **(b)** Simplified PDG of `fibcall-O2.elf`.

Figure 3 Dump and slice of a binary executable.

backward analysis does not follow parameters-out edges. It only adds to the slice instructions up to the entry point. The second backward analysis does not follow call and parameter-in edges. It adds to the slice all instructions down to the procedures output parameters. As a result, unwanted dependencies to output parameters from called procedures are not added to the slice.

4.3 Abstraction of programs for WCET estimation.

Program slicing has many use cases in software engineering. In this paper we want to compute the set of memory locations that impact the WCET of a program. To determine this set of locations we have to determine a suitable slicing criterion. This criterion is the set of pairs $\langle l, v \rangle$ such that l is the label of a conditional branch instruction and v is the set of memory locations read by this instruction. If we consider the program in Figure 2a, it has only one conditional branch instruction: `bdnz 3024` at address 3030. The branch is taken if the count register *ctr* is not zero. So, to compute the locations that should be part of the state of the model we have to compute the slice for the criteria $\{\langle 3030, \{ctr\}\rangle\}$. The set of variables used either explicitly or implicitly by the initial program is $\{r1, r3, r8, r9, r10, lr, ctr\}$. The subset of variables used in the slice is $\{r8, ctr\}$ (see Figure 2b). Only these two registers have to be included in the state of the model.

Let us underline that computing this slice gives us extra informations. For each register in the slice, we also know which instructions impact its value at a given execution point. In the general case, not all the instructions using a register in the slice are in the slice. Such instructions must be processed as instructions using registers not in the slice. Their output must not be written to the state. This allows to further reduce the number of states to explore.

5 Implementation

Architecture

Our tool, BEST for Binary Executable Slicing Tool, computes slices on binary executable files. Its architecture is illustrated in Figure 4.

The decoding and interpretation of the binary files relies on a library generated by the HARMLESS toolchain [12]. HARMLESS is an Hardware Architecture Description Language that is used to model a whole processor. In this study, we are only interested in the model

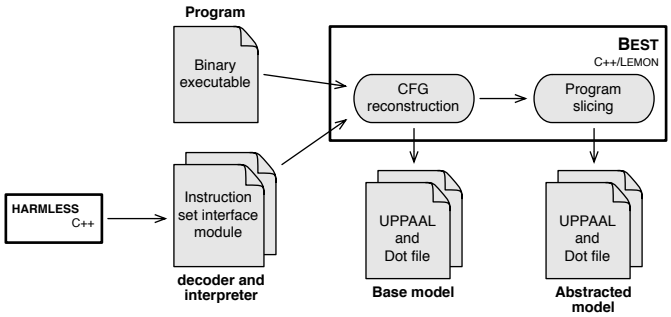


Figure 4 Structure of the tool.

of the instruction set. The HARMLESS compiler is primarily designed to generate either functional or cycle accurate simulators. We re-targeted it to extract static information of the instruction set. The library generated from HARMLESS can read a binary file (`.elf` format in our case) and give information about each instruction such as:

- the instruction mnemonic;
- the memory locations that are read by the instruction;
- the memory locations that are written by the instruction;
- is the instruction a branch instruction? is it a conditional branch? what is its target (if it is statically defined)?

In this study we have used only the PowerPC instruction set, but BEST is not architecture-dependent, thanks to this library.

Using this library, BEST does a CFG reconstruction from a PowerPC binary executable file. Then it applies program slicing to compute the set of memory locations that should be in the model. The main output is an abstract model of the program that can be used to solve the WCET estimation problem with UPPAAL. For validation and visualization purposes, the different models built along the computation can be exported as graphs or as timed automata in the UPPAAL format [14].

BEST is distributed in open-source[1]. To the best of our knowledge, there is no established program slicing tool for non-x86 binary code, especially in open-source. BEST aims to fill this gap. It is implemented in C++. Apart from HARMLESS it relies on the graph manipulation library LEMON [7].

Limitations and Future work

The current version of BEST has different limitations that we want to break in the near future. First, the computation of the abstraction is limited to the register file. The other levels of memory (stack words, all other parts of the volatile memory and non-volatile memory) are automatically excluded from the model. It will be straightforward to take into account the other levels by extending the technique used for the register file. The first step will be the analysis of the stack frame. Being able to track data dependencies between memory and registers through stack loads and stores will produce a more accurate model of the binary executable, and so more accurate WCET estimations.

The second limitation is the limited support for programs with multiple procedures. The slice is currently computed on the PDG. It is not much of hard work to build the SDG and

[1] Available at `https://github.com/TrampolineRTOS/BEST`.

Table 1 Ratio of registers (resp. instructions) in slice compared to the unsliced program.

Compiler	Optim.	Registers in slice				Instructions in slice			
		Avg.	Min	Max.	Std. dev.	Avg.	Min	Max.	Std. dev.
Gcc	`-O0`	61.8%	43.7%	78.9%	8.8%	21.6%	1.7%	43.2%	8.9%
	`-O1`	64.6%	19.1%	87.5%	13.6%	36.7%	3.24%	67.9%	10.2%
	`-O2`	64.3%	13.3%	92.9%	20.3%	37.8%	2.9%	72.5%	15.7%
	`-O3`	62.8%	9%	96.4%	21.7%	34.7%	0.4%	72.2%	18%
Cosmic	`-no`	40.5%	8.6%	86.7%	16.6%	34%	1.9%	60.2%	15.2%
	default	37%	2.8%	66.7%	15.8%	37%	2.8%	66.7%	15.8%

adapt the slicing algorithm because Best has been designed on structures and algorithms intended to produce inter-procedural slices. The main benefit of inter-procedural slicing resides on a more accurate slicing of procedure parameters i.e. even smaller slices.

6 Experimental results

We have conducted experiments to measure the reduction of the set of memory locations that must be included in the model. Given the current restriction of Best, we have focused on the registers. To do so, Best outputs the following information for each program:

- the number of registers used either explicitly or implicitly and the number of instructions in the original program ;
- the number of registers used either explicitly or implicitly and the number of instructions in the sliced program (using the slicing criterion defined in Section 4).

We used the Mälardalen WCET benchmarks [9] to generate the programs. We had to exclude certain programs to account for the current limitation of our tool: program containing floating point arithmetic or switch-case statements and recursive programs.

We used the library generated by the Harmless compiler from a description of a PowerPC e200z4 core based on the 32 bits PowerPC instruction set. This architecture includes 32 general purpose registers ($r0$, $r1$, ..., $r31$) and 5 dedicated registers (cr, xer, lr, ctr, pc). We used two different compilers: Gcc 5.3.1 and Cosmic C 4.3.7. For a given compiler, the generated binary may be very different according to the optimizations. For instance without optimization Gcc generates code where local variables are loaded from and stored to the stack frame each time they are used, whereas in higher optimization levels local variables are allocated in registers. Thus we created different program versions for each optimization level offered by each compiler (4 levels for Gcc and 2 levels for Cosmic C).

All in all, we created 6 versions of each of the 16 Mälardalen benchmarks fitting our constraints and we ran Best on these 96 programs. Due to space limitations the detailed results are provided online[2]. The results are summarized in Table 1 and 2. Table 1 gives the ratio of registers and instructions in the slice compared to the original program. Table 2 gives the number of registers in the slice. It is not meaningful to compare our results with pior work [4] because we consider a different instruction set and different compilers (or at least compiler version for Gcc). We do not comment either on the execution time of Best that were below one second in every case.

[2] Available at `https://github.com/TrampolineRTOS/BEST`.

Table 2 Average number of registers in slice.

Gcc				Cosmic	
`-O0`	`-O1`	`-O2`	`-O3`	`-no`	*default*
8.8	13	11.8	12.1	11.9	12

These results confirm that slicing is an effective abstraction technique for our use case. It allows a significant reduction of the number of variables that should be included in the model (reduction of the dimension of the state space) as well as the number of instructions the output of which should be taken into account (reduction of the number of states to explore). As expected, the best results are obtained for programs with very simple control flow, namely `fdct.c` and `jfdctint.c`, whereas the worst results are obtained for programs with nested control statements and procedure calls , namely `ndes.c` and `adpcm.c`. However, let us underline that the structure of the source code is not always the dominant factor. For some programs, it appears that the compiler (version and/or optimization) has more impact on the capacity of the program slicer to abstract the binary. Example of such programs are `expint.c` and `fir.c`.

7 Conclusion

This article describes the working principles of a tool that computes abstract models of binary executables to be processed by a WCET estimation toolchain based on model checking. Our tool uses program slicing to compute the set of memory locations of the program that have an impact on the WCET of the program. The content of these memory locations is tracked in the abstract model of the program whereas the content of the other ones is abstracted away. A first prototype has been implemented and evaluated. The evaluation has been performed on the Mälardalen benchmarks using two compilers for the PowerPC architecture with varying optimization levels. On average, 41% of the registers can be abstracted. This is a promising result.

References

1 Hiralal Agrawal. On slicing programs with jump statements. *ACM Sigplan Notices*, 29(6):302–312, 1994.

2 Florian Brandner, Stefan Hepp, and Alexander Jordan. Static profiling of the worst-case in real-time programs. In *International Conference on Real-Time and Network Systems (RTNS)*, 2012. doi:10.1145/2392987.2393000.

3 Florian Brandner and Alexander Jordan. Refinement of worst-case execution time bounds by graph pruning. *Computer Languages, Systems & Structures*, 40(3-4):155–170, 2014. doi:10.1016/j.cl.2014.09.001.

4 Franck Cassez and Jean-Luc Béchennec. Timing Analysis of Binary Programs with UPPAAL. In *International Conference on Application of Concurrency to System Design (ACSD)*, 2013.

5 Franck Cassez and Pablo González de Aledo Marugán. Timed automata for modeling caches and pipelines. In *Workshop on Models for Formal Analysis of Real Systems (MARS)*, 2015. doi:10.4204/EPTCS.196.4.

6 Andreas Engelbredt Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: modular execution time analysis using model checking. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET*

2010, July 6, 2010, Brussels, Belgium, pages 113–123, 2010. doi:10.4230/OASIcs.WCET.2010.113.

7 Balázs Dezső, Alpár Jüttner, and Péter Kovács. LEMON – an Open Source C++ Graph Template Library. *ENTCS*, 264(5):23–45, 2011.

8 Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

9 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 136–146, 2010. doi:10.4230/OASIcs.WCET.2010.136.

10 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 101–112, 2010. doi:10.4230/OASIcs.WCET.2010.101.

11 Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

12 Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Guillaume Savaton, and Yvon Trinquet. Harmless, a hardware architecture description language dedicated to real-time embedded system simulation. *JSA*, 58(8):318–337, 2012.

13 Akos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. Interprocedural Static Slicing of Binary Executables. In *International Workshop on Source Code Analysis and Manipulation*, 2003.

14 Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *STTT*, 1(1-2):134–152, 1997.

15 Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *International Symposium on Code Generation and Optimization (CGO)*, pages 136–146, 2009.

16 Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. *ACM Sigplan Notices*, 19(5):177–184, 1984.

17 Christer Sandberg, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Faster WCET flow analysis by program slicing. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 103–112, 2006. doi:10.1145/1134650.1134666.

18 Frank Tip. A Survey of Program Slicing Techniques. *Journal of programming languages*, 3(3), 1995.

19 Mark Weiser. Program Slicing. In *International Conference on Software Engineering (ICSE)*, 1981.

20 Reinhard Wilhelm. Why AI + ILP is Good for WCET, but MC is not, nor ILP alone. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2004.