DELOOP: Automatic Flow Facts Computation Using Dynamic Symbolic Execution

Hazem Abaza[1]
TU Dortmund, Germany

Zain Alabedin Haj Hammadeh
Institute for Software Technology, German Aerospace Center (DLR), Braunschweig, Germany

Daniel Lüdtke
Institute for Software Technology, German Aerospace Center (DLR), Braunschweig, Germany

Abstract

Constructing a complete control-flow graph (CGF) and computing upper bounds on loops of a computing system are essential to safely estimate the worst-case execution time (WCET) of real-time tasks. WCETs are required for verifying the timing requirements of a real-time computing system. Therefore, we propose an analysis using dynamic symbolic execution (DSE) that detects and computes upper bounds on the loops, and resolves indirect jumps. The proposed analysis constructs and initializes memory models, then it uses a satisfiability modulo theories (SMT) solver to symbolically execute the instructions. The analysis showed higher precision in bounding loops of the Mälardalen benchmarks comparing to SWEET and oRange. We integrated our analysis with the OTAWA toolbox for performing a WCET analysis. Then, we used the proposed analysis for estimating the WCET of functions in a use case inspired by an aerospace project.

2012 ACM Subject Classification Computer systems organization → Real-time system specification; Software and its engineering → Real-time systems software

Keywords and phrases Real-Time, WCET, Symbolic execution

Digital Object Identifier 10.4230/OASIcs.WCET.2022.3

Acknowledgements The authors thank Prof. Dr. Heiko Falk for his valuable feedback. Also, they thank Patrick Kenny for proof-reading the paper.

1 Introduction

Timing analyses aim to verify the timing constraints of a computing system. A timing analysis should start with computing a safe upper bound on the worst-case execution time (WCET) of each task (or sub-task in the case of directed acyclic graph (DAG) tasks) in the computing system. Then, a response-time analysis or a schedulability test should follow considering the scheduling policy and the deadline of each task. Estimates of the WCET of tasks can be obtained by using measurement, static or hybrid methods. The applications may be complex, therefore, the choice of the best method is not straightforward. However, only the static methods can cover all corner cases and can therefore provide safe upper bounds on the WCETs. Also, the development process is iterative, hence, setting up a static analysis would potentially save time and effort after applying changes compared to using measurements.

[1] This author's contribution has been conducted at the German Aerospace Center (DLR) while pursuing his Master's degree

A static WCET analysis has to provide an abstract model of the micro-architecture including, e.g., pipeline and caches, and facts on the program flow. Flow facts include program control-flow and upper bounds on loops. The Implicit Path Enumeration technique (IPET) computes the WCET as an objective function maximization in an integer linear programming (ILP) problem of the abstract interpretation of the micro-architecture and the execution paths of the program [19]. This paper presents an analysis based on dynamic symbolic execution (DSE) to automatically 1) compute upper bounds on loops and; 2) resolve indirect jumps to construct the control flow of the program. Automatic loop bounding and indirect jump resolution are desirable over manual annotation, which is error-prone and sometimes not manageable due to the amount of annotation needed [8].

DSE is a systematic approach to explore program paths and defining predicates [4]. A satisfiability modulo theories (SMT) [7] solver checks the satisfiability of the predicates to identify the next path. DSE has been used widely in computer security for, e.g., vulnerability discovery and reverse-engineering [27]. We use DSE in this work to explore program paths to identify potential jump targets and compute loop bounds. DSE reports results based on the given input values to the program, therefore, it cannot guarantee computing a safe upper bound on the loop bounds for applications implemented as an input-value-based state machine. In such applications, a value analysis should support DSE. However, applications that are implemented following the data-flow programming paradigm can use our DSE-based analysis safely as long as the control flow is input-value independent. In this work, we have special interest in data-flow applications, such as some on-board data processing (OBDP) applications. Hence, a value analysis is beyond the scope of this paper.

Developing embedded software using the inversion control programming principle improves modularity and maintainability [10]. Therefore, it is not uncommon nowadays to develop embedded software using e.g. C++-based software frameworks. C++-based software frameworks are the main motivation for this work. The German Aerospace Center (DLR) has developed a C++ software framework for developing OBDP applications, called Tasking Framework [17]. We will use it in this paper as a case study. Modularity and maintainability come at the cost of the underlying complexity. Therefore, performing static WCET analysis for such software is challenging. The challenges can be narrowed down to:

- **Control-flow reconstruction due to indirect jumps**
  Indirect jumps result mainly from virtual methods. They ensure that the correct function is called for an object. Calling a virtual method is translated at the binary level to an indirect jump instruction, in which the memory location of the target function is stored in a register. In Listing 1, the function *synchronizeStart()* in the Tasking Framework is defined as a virtual method. Listing 2 shows in Line 3 how the call is translated to an indirect jump in assembly. Such as branching instruction is challenging for the static analysis as it fails to fully construct the control-flow graph (CFG).

Listing 1 Indirect jump inside a simple for-loop where the bound is known at compile time.

```
void Tasking::TaskImpl::synchronizeStart(void){
for (unsigned int i = 0; (i < inputs.size()); i++){
        static_cast<ProtectedInputAccess&>(inputs[i]).synchronizeStart();}}
```

Listing 2 Indirect jump in the assembly code.

```
1  00009cca        ldr r3,[r3,0x7ff000000000]
2  00009ccc        move r0, r2
3  00009cce        blx r3
```

- **Loop Bounding**
  Loops that iterate over lists as shown in Listing 3 are specially challenging source-level loop bounding tools. The information about the list's size and its location in memory is not always available at the source level and requires additional binary level analysis to extract. Even simple *for* loops like the one presented in Listing 1 may be bounded by an object's value, which requires knowledge of the content of the memory location where the object is stored. Moreover, some loops are only available at the binary level. For example, constructing **n** objects from the same class sometimes is translated into loops at the binary level. These loops are hard to detect and bound at the source level.

Listing 3 A loop iterates over a bounded list.

```
//The loop iterates over the associated inputs to notify the task.
void Tasking::Channel::push(void) {
for (InputImpl* i = m_inputs; i != NULL; i = i->channelNextInput){
        i->notifyInput();}}
```

Our analysis uses a low level intermediate representation (LLIR) of the analyzed program as input. It translates each instruction into an SMT formula and symbolically executes them. We build a memory model, stack model, and register model to enhance the DSE such that each SMT formula updates the memory, stack and register models accordingly. With the help of a loop detection algorithm, namely Johnson's Algorithm [20], we bound loops.

We evaluated our analysis on the Mälardalen benchmark and compared the results with other tools, e.g., oRange [5]. The results showed high precision in bounding loops. We used the proposed analysis to provide flow facts to the open-source toolbox OTAWA [2]. Then OTAWA was used to compute the WCET of some Tasking Framework methods for the Cortex M3 architecture.

The rest of the paper is organized as follows: Chapter 2 visits the related work. In Chapter 3, we present our DSE-based analysis to compute loop bounds and resolve indirect jumps. The proposed analysis is evaluated in Chapter 4. Chapter 5 concludes the paper.

2 Related Work

In the scientific literature, SMT has been used to expose the program semantics to improve the tightness of the computed WCETs by eliminating infeasible paths. In [24], Ruiz et al. worked on machine code where they formulated the program states as sets of predicates to expose infeasible paths using SMT solvers. Henry et al. in [18] formulated the problem of computing the WCET as optimization modulo theory, which extends the satisfiability modulo theory. Neither paper addressed the problem of resolving indirect jumps. In [18], the loops must be unrolled before applying the proposed analysis. The analysis of program semantics is admitted to be easier at the source level [23]. However, for C++ software frameworks, performing the analysis at LLIR level is easier than at source level due to the complexity of the C++ language.

Gustafsson et al. presented in [16] an automated analysis to derive loop bounds using *abstract execution*. However, the proposed analysis was not developed to bound loops that iterate over a bounded list like in Listing 3. Therefore, we doubt that the polynomial correlations from the abstract execution can comprehend such loops. Besides that, the analysis was not developed to resolve potential indirect jumps in the CFG.

In many aerospace projects, intensive measurements are applied to estimate the WCET [12] using commercial tools like RapiTime [22]. Applying static analysis is done on critical functions [13]. Using aiT [11] is common to that end. Both approaches need human interaction, e.g., manual annotation. This work aims to automate the flow facts computation and to use the open-source toolbox OTAWA.

3 DSE-based Flow Fact Computation

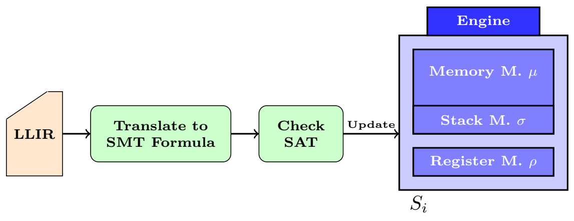

Figure 1 Analysis steps in DELOOP with the engine state.

In this section, we elaborate on our proposed analysis: Dynamic symbolic Execution-based LOOP bounding (DELOOP). The analysis steps are shown in Figure 1. DELOOP takes the executable binary of the given program as input, computes loop bounds and resolves indirect jumps. The analysis carries out the following steps:

1. Lifting the executable binary to *static single-assignment (SSA)* LLIR. We use the commercial tool BINARYNINJA [3] for that purpose. Performing the analysis on LLIR makes the analysis platform-independent.
2. Detecting the loops using Johnson's Algorithm.
3. Translating each SSA instruction in the LLIR into SMT formulas. We use Microsoft Z3 [6] as the SMT solver.
4. Building and initializing memory, stack and register models as arrays of bit vectors. The models will store the state of the memory, stack and registers.
5. Symbolically executing each instruction by checking the satisfiability of the equivalent SMT formula and updating the affected model.

After lifting the executable binary of the given program, the CFG is reconstructed. DELOOP computes an upper bound on the number of executions for each *basic block*. Combined with the loop detection algorithm, DELOOP can report an upper bound on loops. The lifting tool, BINARYNINJA, is a reverse engineering framework used mainly for binary analysis. We used its Python API to parse the assembly code and facilitate all parts of the analysis.

3.1 Loop Detection

We implemented Johnson's Algorithm to detect loops in the given CFG. The algorithm takes the CFG as a directed graph $\mathbf{G}\ (\mathbf{V}, \mathbf{E})$, which consists of a non-empty set of vertices $\mathbf{V}$ and a set of ordered pairs of vertices called edges $\mathbf{E}$. The algorithm can detect the loops, known as *elementary circuits*, within a time bounded by $\mathbf{O((n+e)(c+1))}$ and space by $\mathbf{O(n+e)}$, where $\mathbf{n}$ is the number of vertices, $\mathbf{e}$ the number of edges and $\mathbf{c}$ the elementary circuits in the graph. A single elementary circuit is defined as a closed path where no node appears twice, except that the first and last nodes are the same. Two elementary circuits are distinct if they are not cyclic permutations of each other.

DELOOP groups the basic blocks in a single elementary circuit (i.e., loop). Each detected loop, denoted by λ, is given a loop ID that is equal to theID of the last basic block in the loop. Recursive function calls are not handled with the loop detection algorithm. However, DELOOP can automatically bound the depth of recursion during the DSE phase.

3.2 SMT formulas and engine state

To symbolically execute the program, we compile the SSA LLIR into SMT formulae. The SSA form of the LLIR facilitates the whole translation process as every SSA instruction is directly mapped to one SMT formula using *array* and *bit vector* theories.

Two memory models are built based on the array theory. Data inside the arrays are formulated as bit vectors with a *size* that matches the target architecture; thus, the arrays are defined as arrays of bit vectors. The first memory is used for symbolic execution of the load/store instructions and is initialized with the values of all the program's data variables in the given executable binary. The second memory, the stack, is dedicated for the push/pop instructions. Both memory models grow and are updated dynamically along the DSE of the program.

Besides the models for memory and stack, we have a third model for representing the registers and flags. This model is also updated dynamically. Together, the memory model μ, the stack model σ and the register model ρ represent the *engine state* S. SSA instructions are translated to formulas in a form that implies the mathematical effect of the SSA instruction on the engine state. For example, the SSA instruction $R_2 = R_3 + 1$ is translated as shown in Equation 1 where bit vector variables are defined for R_2, R_3 and the immediate value.

$$R_2 = R_3 + 1 \implies BitVec(R_2, size) = BitVec(R_3, size) + BitVec(1, size) \tag{1}$$

Memory instructions are also interpreted in the same way. For example, the SSA instruction shown in Equation 2 is computed as *select(mem,*$0x8080$*)* where *mem* is the memory model and $0x8080$ is the load address. The translator performs the previous steps for all kinds of LLIR operations.

$$R_2 = [data\_0x8080] \implies BitVec(R_2, size) = select(mem, 0x8080) \tag{2}$$

3.3 Dynamic symbolic execution

DSE is used in a number of industrial tools to explore the CFG of a sequential program $\mathbf{P}$ for identifying test inputs that can lead the execution to new paths [7]. A path Π in the program $\mathbf{P}$ is said to be feasible if there is a non-empty set of inputs I such that $\forall i \in I$ the execution of $\mathbf{P}$ follows the path $\mathbf{\Pi}$. If $I = \emptyset$, then the path is not feasible.

Inspired by that concept, we try to explore loop bounds. For a program $\mathbf{P}$ starting at an initial path $\mathbf{\Pi_{in}}$ with a set of initial inputs I_{in}, we aim to deduce the set of outputs at the end of the path $\mathbf{\Pi_{in}}$: I_{out}. Our approach uses I_{out} as the new I_{in} to reach the next path. Following this concept, we dynamically execute all the feasible paths in the given CFG.

DELOOP checks the satisfiability of every SMT formula and updates the engine state S with the effect of execution. The SMT formulas are categorized into four main types: memory-related, stack-related, register-related and director formulas. Director formulas represent the branching instructions and are responsible for setting the execution path for the solver. Memory-related formulas update the memory model μ in the engine state. Similarly, stack and registers-related formulas update the stack σ and register ρ models respectively.

The concept of states transformed our execution from a static to a dynamic symbolic execution. For example, during the translation of $R_2 = R_3 + 1$, the translator first checks whether there are previous variables in the engine state for R_3 and R_2. In the case of already existing variables, the value of R_3 is fetched from ρ and increased by one and then assigned to R_2. If R_3 has a previous value of 100, then the translation process is done as follows:

$$R_2 = R_3 + 1 \implies BitVec(R_2, size) = BitVec(100, size) + BitVec(1, size) \tag{3}$$

The same is true for the memory instruction in Equation 2. If the address $0x8080$ has a value, let it be $0xa080$, then R_2 will be updated as follows: $R_2 = [data\_0x8080] \implies 0xa080$.

3.3.1 Bounding loops

The execution starts from the program entry point and continues to the CFG's exit function, or to the synthetically inserted exit point, which can be defined by the person who performs the analysis to stop the analysis at a designated point. DELOOP symbolically executes each SSA instruction and updates the engine state. Also, for each basic block B_i, DELOOP stores the number of executions EX_i of B_i. After finishing executing, the loops that are detected by Johnson's Algorithm, are visited and the bound is computed as the maximum number of executions for each basic block in loop λ. Let $\bar{\beta}$ be a function that returns an upper bound for a given loop λ:

$$\bar{\beta}(\lambda) = \max_{\forall B_i \in \lambda} \{EX_i\} \tag{4}$$

In the case of nested loops, Equation 4 returns the total number of executions of the inner loop, which is a non-necessary over-approximation. Therefore, before reporting the loop bounds we check if there are nested loops and update the loop bounds of inner loops as follows: $\bar{\beta}(\lambda_{inner}) = \bar{\beta}(\lambda_{inner})/\bar{\beta}(\lambda_{outer})$

3.3.2 Indirect jumps

Symbolic execution builds correlations between basic blocks for the program under analysis. It generates equations depending on an input variable to describe the jump target and the execution sequence of the program. These correlations can be used to resolve indirect jumps and anticipate the next basic block to be executed. However, the static symbolic execution generates multiple equations, based on the input and CFG path, that may satisfy the jump target resolution. These equations can be represented as first-degree-polynomial equations in the form of $a + x * C$ where a is the base of the jump table and $x * C$ is an offset. In each SMT formulated equation, C will depend on the input and the CFG path. The dynamic symbolic execution narrows the search space for these equations as it defines the execution path based on the given inputs for every solution iteration. In our generated engine model, the value of the indirect jump register is being updated based on the SAT formulations from state i till the indirect jump call instruction. That implicitly resolves the generated SAT inter-basic block formulations.

During the execution in our execution model, the indirect jump target is correlated to the CFG and the input through the forward propagation of the data. The result correlation is an SMT formulation of bit vectors and memory arrays. To resolve the formulation into meaningful targets, a reversed data-flow analysis with defined stop conditions needs to be run. However, this solution will lead to multiple resolutions for the formulation with no SAT guarantees. The dynamic symbolic solution solves this problem through the forward update of the engine states.

$$\begin{split} call(R3) \implies BitVec(R_3, size) = BitVec(select(mem, 0x8080), size) + \\ BitVec(select(mem, BitVec(R_1, size)), size) \end{split} \tag{5}$$

The update of the state after each execution implicitly preserves forward propagation of the memory arrays and bit vector values that will correctly resolve the jump target. For example, an indirect jump call formulation as in Equation 5 can be resolved to the jump target address by substituting the propagated values of the memory address and R_1 at the engine state executing the indirect call instruction.

Table 1 Benchmark results where L: loops; E: exact bounding.

Program	*#L*	*E*	*Program*	*#L*	*E*	*Program*	*#L*	*E*
adpcm	27	27	bs	1	1	cnt	4	4
cover	3	3	crc	6	6	duff	2	2
edn	12	12	expint	3	3	fac	1	1
fdct	2	2	fft1	30	30	fibcal	1	1
fir	2	2	inssort	2	2	jcomplex	2	2
ludcmp	11	11	matmult	7	7	ndes	12	12
ns	4	4	nsichneu	1	0	prime	2	2
qsort-exam	6	6	qurt	3	3	select	4	4
ud	11	11						

Table 2 Loop-bounding tools comparison where BLT: bounded loop total.

Tool	BLT	% BLT	E	% E
DELOOP	158	99%	158	99%
oRange [5]	134	84%	117	73.5%
SWEET [9]	100	63%	81	51%

4 Evaluation

4.1 Mälardalen WCET benchmarks

The Mälardalen WCET benchmarks [15] are open-source test programs for WCET analysis. Although the Mälardalen WCET benchmarks are ANSI-C code, they can be used to verify our tool and compare its results against the state-of-the art tools. For validating our tool, we use Tasking Framework in the next section.

We used 25 programs from the Mälardalen WCET benchmark suite to test our tool. The results are presented in Table 1. E represents the number of loops which could be exactly bounded. For all programs except one, DELOOP can exactly bound the loops. For the very large function **nischneu**, the lifter, BINARYNINJA, failed to restore the CFG of the main function. It might not be surprising to exactly bound all the detected loops because we symbolically execute the program using the SMT formulas. In Table 2, we compare our results with oRange [5] and SWEET [9]. For oRange and SWEET, we recall the results from the cited papers. BLT and %BLT represent the number of bounded loops and percentage out of 159 loops respectively.

4.2 A use case developed using Tasking Framework

Tasking Framework [17] is an open-source [14] software development library. Also, it is a multithreading event-driven execution platform for embedded software. It provides abstract classes with virtual methods to realize an application by a directed graph of connected *tasks* and *channels*, where each computation block of a software component is realized by the class *task*, and the data exchanged between tasks is an object of the class *channel*. Periodic tasks are connected to a source of events as shown in Figure 3. Tasks can start executing as soon as their input data is available, thus, some of them can work concurrently. A task forwards the data to the next task by pushing it to the associated channel, which represents an interface between two tasks, and activating the next task. This data-driven activation mechanism is implemented in Tasking Framework with different activation semantics, e.g., and, or semantics.

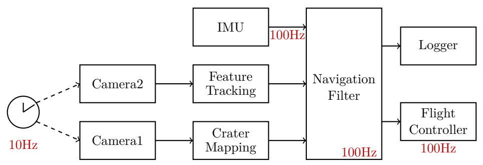

Figure 2 Use case inspired from the optical navigation sub-system in the ATON project [25].

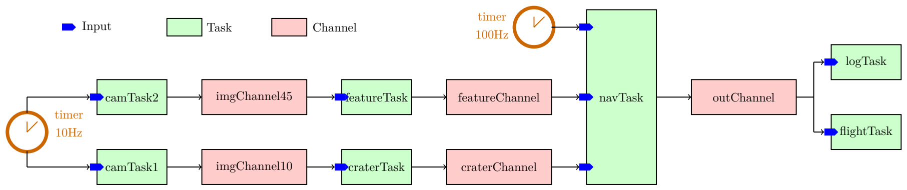

Figure 3 The use case in Figure 2 as realized by the Tasking Framework.

Tasking Framework has been used for many real-world aerospace applications such as Autonomous Terrain-based Optical Navigation (ATON)[25] and Scalable On-Board Computing for Space Avionics (ScOSA)[21]. ScOSA is an ongoing project in 2022.

We evaluated our analysis on a use case inspired from the optical navigation sub-system in the ATON project [25], and implemented using the Tasking Framework. In this sub-system, two camera drivers, *camTask1* and *camTask2*, run periodically and transfer the images to 1) a crater navigation component *craterTask* and 2) a feature tracking component *featureTask* respectively. The output of these components feeds the navigation filter *navTask* to estimate the position. The output is logged by *logTask* and forwarded to the flight controller *flightTask*.

4.2.1 Results

- **SWEET:** Its input is an IR based on the ARTIST2 Language for Flow Analysis (ALF). To apply SWEET, we built the binary code, then lifted it to LLVM using RetDec [1], which is a retargetable machine code decompiler based on LLVM. We translate the LLVM IR to ALF using the translator introduced in [26]. SWEET failed to build its abstract execution model.
- **oRange:** We generated the binary code and lifted it back to C code using RetDec. oRange reports *NOCOMP* for all loops in the use case.
- **DELOOP:** We integrated DELOOP with OTAWA as shown in Figure 4 to compute the WCET.

The results are presented here:

- **Loops:** Unlike the loops in the benchmark, Tasking Framework does not contain any simple loop like the one in Listing 4. The loops in Tasking Framework are either bounded by an object's attribute, see Listing 1, or iterates over a list, see Listing 3. However, the code of the user-developed tasks may contain different types of loops.

Listing 4 Simple ANSI-C loop.

```
for (int i=0; i<20 ; i++){}
```

DELOOP provides more than one bound for loops, one bound per instance. For example, each channel in our case study will run its own copy of the *push()* function; thus, the loop in Listing 1 will be executed by different tasks in the case study. DELOOP will compute an upper bound for each copy of the loop. The loop is bounded by the number of associated inputs and is thus bounded by *two* for the *navTask* while it is bounded by *one* for all other tasks.

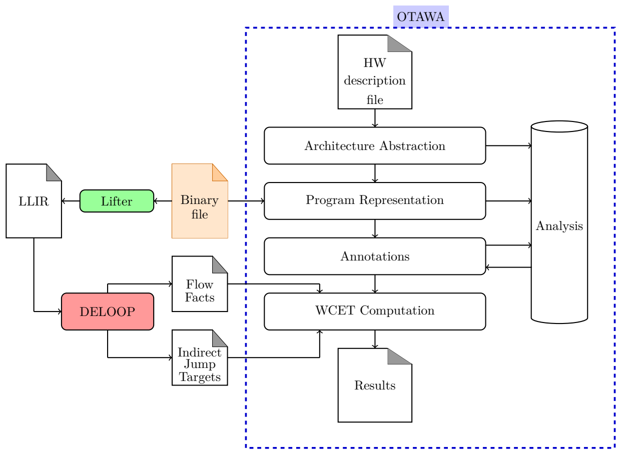

Figure 4 DELOOP integrated with OTAWA.

Also, DELOOP detected an implicit loop, which does not appear in the source code, as shown in Listing 5. *navTask* has *three* input objects, thus, the bound of this loop is *three*.

Listing 5 A constructor template translated into a loop in assembly code.

```
template<size_t n>
InputArrayProvider<n>::InputArrayProvider(void):
InputArray(inputMemory, n) {}
```

- **Indirect jumps:** The indirect jumps in Tasking Framework are mainly due to virtual methods. Virtual methods are there to support, for instance, three scheduling policies. After compilation, each indirect jump has only one target. Therefore, resolving the indirect jumps using DSE is safe. All the indirect jumps in our case study were resolved.
- **WCET Computation:** As mentioned earlier in this paper, we use OTAWA as a static analyzer and DELOOP as a flow facts generator as shown in Figure 4. This setup expands the capabilities of OTAWA in estimating WCET for C++ code. After given OTAWA a hardware description file for *armv-7m*, the WCET estimation starts with reconstructing the CFG. Then, the results of the loop analysis performed by DELOOP are passed to OTAWA for the WCET analysis. The analysis is performed for a bare-metal implementation.

  In OBDP applications based on a data-flow programming paradigm, ideally, each task pushes to the associated channel to activate the next task. This data-driven activation mechanism is implemented in Tasking Framework via the *push()* method. *push()* starts a chain of method calls, which ends with *queue()* that queues the next connected task in the ready queue. The chain contains two loops and one indirect jump. Bounding the WCET of *push()*, i.e., the chain of function calls, helps in estimating the overhead imposed by Tasking Framework. The implementation of *push()*[2] contains two loops: Loop1 is the outer loop that iterates over the tasks associated with the considered channel; Loop2 is executed for each iteration on Loop1 and it iterates over the inputs of each associated task with the considered channel. The WCET of *push()* executed by the task *camTask1* is 2435 cycles. Note that the channel *imgChannel10* is associated with only one input object, i.e. task *craterTask*. The same result is valid for the *push()* executed by the task

[2] `https://github.com/DLR-SC/tasking-framework/commit/349ce3ddd98cd1fe69daf08318e1b8cbf9c01e9b`

camTask2 because it has the same flow facts. The WCET of *push()* executed by the task *featureTask* and *craterTask* is 3635 cycles. Finally, the WCET of *push()* executed by the task *navTask* is 4800 cycles. Table 3 summarizes the results. As the results show, *push()* has different WCET values for different tasks, but it is bounded and fixed for each task.

Table 3 Results of the WCET analysis for the push function in the use case in Figure 3.

Task	Loop1	Loop2	WCET (cycles)
camTask1	1	1	2435
camTask2	1	1	2435
craterTask	1	3	3635
featureTask	1	3	3635
navTask	2	1	4800

- **Performance:** The analysis was executed on a workstation with Linux, i7-9750H processor and 16Gbyte RAM. The use case has a binary size = 664 kbyte. The analysis used 25% of the CPU capacity and 640 Mbyte of memory. The analysis took about 81 seconds to compute the flow facts.

5 Conclusions

The complexity of modern architectures, software development practices and compilers often leads to executable code which is difficult to match to its source code. Additionally, manual computation of flow facts and manual annotation are error-prone especially for software developed using object-oriented practices, in which one loop can be executed many times by different objects for different number of iterations. This provides motivation to compute the flow facts at the binary level.

In this work, we proposed an analysis to bounding loops and resolving indirect jumps using DSE. The proposed analysis lifts the executable binary to SSA LLIR, then each SSA instruction is translated into an SMT formula. Using the Z3 SMT solver, the satisfiability is checked and memory, stack and register custom models are updated accordingly. We showed that the proposed analysis can safely compute upper bounds on loops in the Mälardalen benchmarks. Also, we used the proposed analysis together with OTAWA to compute the WCETs for a use case developed using the Tasking Framework.

Although successful in computing loop bounds and resolving indirect jumps, the proposed analysis has two main limitations: 1) the need for value analysis for some applications to guarantee that the computed bounds are safe; 2) using a memory model, which might be very complex for large applications and therefore increase the analysis time. We will investigate in the future development the scalability of DELOOP to larger applications in our ScOSA project. Also, we are interested in verifying whether DELOOP yields any improvement in terms of WCET estimation by conducting more case studies for which oRange and SWEET can compute the flow facts.

References

1 Avast. RetDec. `https://github.com/avast/retdec`. [accessed May 03, 2022].

2 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

3 BINARYNINJA. Binary Ninja. `https://binary.ninja/`. [accessed May 03, 2022].

4 Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656, 2016. `doi:10.1109/SANER.2016.43`.

5 Marianne de Michiel, Armelle Bonenfant, Hugues Casse, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 161–166, 2008. `doi:10.1109/RTCSA.2008.53`.

6 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

7 Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011. `doi:10.1145/1995376.1995394`.

8 Andreas Ermedahl and Jakob Engblom. Execution time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer*, 4:437–455, 2007.

9 Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Christine Rochange, editor, *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2007.1194`.

10 Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, October 1997. `doi:10.1145/262793.262798`.

11 Christian Ferdinand and Reinhold Heckmann. aiT: Worst-case execution time prediction by static program analysis. In Renè Jacquart, editor, *Building the Information Society*, pages 377–383, Boston, MA, 2004. Springer US.

12 Jorge Garrido, Daniel Brosnan, Juan A. de la Puente, Alejandro Alonso, and Juan Zamorano. Analysis of WCET in an experimental satellite software development. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 81–90, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2012.81`.

13 Jorge Garrido, Juan Zamorano, and Juan A. de la Puente. Static analysis of WCET in a satellite software subsystem. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 87–96, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2013.87`.

14 German Aerospace Center (DLR). Tasking Framework. `https://github.com/DLR-SC/tasking-framework`. [accessed May 03, 2022].

15 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. `doi:10.4230/OASIcs.WCET.2010.136`.

16 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 57–66, 2006. `doi:10.1109/RTSS.2006.12`.

17 Zain Alabedin Haj Hammadeh, Tobias Franz, Olaf Maibaum, Andreas Gerndt, and Daniel Lüdtke. Event-driven multithreading execution platform for real-time on-board software systems. In *Proceedings of the 15th Annual Workshop on Operating Systems Platforms for Embedded Real-time Applications*, pages 29–34, 2019.

18 Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maïza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. *SIGPLAN Not.*, 49(5):43–52, June 2014. doi:10.1145/2666357.2597817.

19 Hajer Herbegue, Hugues Cassé, Mamoun Filali, and Christine Rochange. Hardware architecture specification and constraint-based WCET computation. In *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 259–268. IEEE, 2013.

20 Donald B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

21 Andreas Lund, Zain Alabedin Haj Hammadeh, Patrick Kenny, Vishav Vishav, Andrii Kovalov, Hannes Watolla, Andreas Gerndt, and Daniel Lüdtke. ScOSA system software: the reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture. *CEAS Space Journal*, May 2021. doi:10.1007/s12567-021-00371-7.

22 RAPITASytems. RapiTime. https://www.rapitasystems.com/products/rapitime. [accessed May 03, 2022].

23 Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Erwan Jahier, Nicolas Halbwachs, Fabienne Carrier, Mihail Asavoae, and Rémy Boutonnet. Improving WCET evaluation using linear relation analysis. *Leibniz Transactions on Embedded Systems*, 6(1):02:1–02:28, February 2019. doi:10.4230/LITES-v006-i001-a002.

24 Jordy Ruiz and Hugues Cassé. Using SMT solving for the lookup of infeasible paths in binary programs. In Francisco J. Cazorla, editor, *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 95–104, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2015.95.

25 Stephan Theil, N Ammann, Franz Andert, Tobias Franz, Hans Krüger, Hannah Lehner, Martin Lingenauber, Daniel Lüdtke, Bolko Maass, Carsten Paproth, et al. ATON (autonomous terrain-based optical navigation) for exploration missions: recent flight test results. *CEAS Space Journal*, 10(3):325–341, 2018.

26 Rick Veens. Adding support for static WCET analysis to LLVM, 2018. Master's thesis. URL: https://research.tue.nl/en/studentTheses/adding-support-for-static-wcet-analysis-to-llvm.

27 Alexey Vishnyakov, Andrey Fedotov, Daniil Kuts, Alexander Novikov, Darya Parygina, Eli Kobrin, Vlada Logunova, Pavel Belecky, and Shamil Kurmangaleev. Sydr: Cutting edge dynamic symbolic execution. In *2020 Ivannikov Ispras Open Conference (ISPRAS)*, pages 46–54, 2020. doi:10.1109/ISPRAS51486.2020.00014.