Scenario-Aware Program Specialization for Timing Predictability

JOSCHA BENZ and OLIVER BRINGMANN, University of Tübingen

The successful application of static program analysis strongly depends on flow facts of a program such as loop bounds, control-flow constraints, and operating modes. This problem heavily affects the design of real-time systems, since static program analyses are a prerequisite to determine the timing behavior of a program. For example, this becomes obvious in worst-case execution time (WCET) analysis, which is often infeasible without user-annotated flow facts. Moreover, many timing simulation approaches use statically derived timings of partial program paths to reduce simulation overhead. Annotating flow facts on binary or source level is either error-prone and tedious, or requires specialized compilers that can transform source-level annotations along with the program during optimization. To overcome these obstacles, so-called *scenarios* can be used. Scenarios are a design-time methodology that describe a set of possible system parameters, such as image resolutions, operating modes, or application-dependent flow facts. The information described by a scenario is unknown in general but known and constant for a specific system. In this article,¹ we present a methodology for scenario-aware program specialization to improve timing predictability. Moreover, we provide an implementation of this methodology for embedded software written in C/C++. We show the effectiveness of our approach by evaluating its impact on WCET analysis using almost all of TACLeBench—achieving an average reduction of WCET of 31%. In addition, we provide a thorough qualitative and evaluation-based comparison to closely related work, as well as two case studies.

CCS Concepts: • **Computer systems organization** → **Embedded software**; *Firmware*; • **Software and its engineering** → **Source code generation**;

Additional Key Words and Phrases: Predictability, timing analysis, worst-case execution time, analyzability

ACM Reference format:

Joscha Benz and Oliver Bringmann. 2021. Scenario-Aware Program Specialization for Timing Predictability. *ACM Trans. Arch. Code Optim.* 18, 4, Article 54 (September 2021), 26 pages.
<https://doi.org/10.1145/3473333>

1 INTRODUCTION

Developing software that is well-suited for static analysis is a non-trivial task. It requires a design that allows static derivation of flow facts like loop bounds and feasible paths. Since modern software is developed generically, flow facts often depend on parameters of the software that are

¹New Paper, Not an Extension of a Conference Paper.

Authors' address: J. Benz and O. Bringmann, Embedded Systems Department, University of Tübingen, Sand 13, Tübingen 72076, Germany; emails: {joscha-joel.benz, oliver.bringmann}@uni-tuebingen.de.

This work is licensed under a Creative Commons Attribution-NoDerivs International 4.0 License.

© 2021 Copyright held by the owner/author(s).
1544-3566/2021/09-ART54 \$15.00
<https://doi.org/10.1145/3473333>

unknown in general but constant and known for a specific system. Embedded software is developed generically, among other things, to be able to target a wide range of execution platforms and systems. Therefore, some parameters of the software, called *system parameters*, are variable to accommodate for different HW platforms, device configurations or use cases. Given a specific system, this information is not variable but fixed and the corresponding parameters are constant and known as well. As flow facts often depend on such parameters, it is possible to improve timing predictability by making these constant in the source code as well.

To enable generically developed software that is automatically specialized for a given system, so-called scenarios can be exploited. *System scenarios*, which are similar but not equal to the scenarios presented in this work, are commonly used in embedded-systems design and development [9, 10, 31]. In this work, a scenario simply specifies a set of system parameters for a specific embedded software. As mentioned previously, system parameters are parameters of the software that are unknown in general but known and constant for a specific system. In the scope of this work, the parameters we are interested in are function or method parameters. A *scenario instance* maps system parameters to associated values. If a parameter has multiple possible values, there exists an instance for each value. System parameters may also be operating mode or state dependent and it is possible to switch between multiple scenario instances at run time. Thus, scenarios can be used to statically capture and handle dynamic behavior of an embedded system. The value(s) of a system parameter can be derived from the system's environment, the platform that runs the software, or the specification of the system. Possible candidates for parameters are message or buffer sizes, for example, in low-level communication-device drivers or image resolutions in image-processing algorithms. Key or block length in encryption algorithms or error detection and correction codes, such as CRC, can also be parameters of a scenario. Scenarios are particularly well-suited for embedded software, since embedded systems usually have clearly defined and restricted operating conditions and requirements. Moreover, embedded hardware often is specialized for a specific class of applications as well, for example, accelerators for highly efficient computations. Hence, using scenario-aware program specialization it is possible to semi-automatically represent this knowledge in the software.

For example, the following listing shows the signature of a function from the *cjpeg_wrbmp* benchmark (TACLeBench).

```
void write_colormap(j_decompress_ptr cinfo, int map_colors, int map_entry_size,
                  int cMap);
```

The parameters of this function that qualify as system parameters are *map_colors*, *map_entry_size*, and *cMap*. Hence, the set $S = \{\text{map\_colors}, \text{map\_entry\_size}, \text{cMap}\}$ is a valid scenario for *cjpeg_wrbmp*. *map_colors* denotes the number of different colors in an image, *map_entry_size* specifies the number of bytes used to encode a color and *cMap* controls whether to use a colormap at all. Given a specific system, it is possible to fix these parameters to one or a few constants. It may also be possible that a single value for each parameter is sufficient for the system to fulfill its function. For example, the set of pairs $I = \{(\text{map\_colors}, 768), (\text{map\_entry\_size}, 4), (\text{cMap}, 1)\}$ is a possible scenario instance. Although practically infeasible, each time *cjpeg_wrbmp* is deployed on a system, it could be changed manually to reflect the information provided by an associated scenario instance on source level. To allow for the development of software which contains such scenario-dependent information, we propose scenario-aware program specialization. The goal of this methodology is to use scenario-provided information to improve predictability and analyzability of software. In addition, we provide an implementation of our approach as part of a clang-based code-transformation framework that supports software written in C and C++.

The remainder of this article is structured as follows: Section 2 discusses related work and Section 3 introduces scenario-aware program specialization for timing predictability. Section 4 gives a detailed explanation of our transformation framework that implements our methodology for C/C++. Section 5 evaluates the effect of our approach on timing predictability w.r.t. **worst-case execution time (WCET)** using most of the TACLeBench benchmark suite [7], two case studies and a comparison with related work. Finally, Section 6 concludes this work and discusses possible future work.

2 RELATED WORK

There exists much work on improving (timing) predictability of software with focus on numerous aspects, while many approaches aim at optimizing for WCET.

Lokuciejewski et al. introduced a methodology that uses function specialization, or procedure cloning, to reduce pessimism in WCET analysis [21]. To that end, only functions that are part of the **worst-case execution path (WCEP)** are considered. As this can lead to a large increase in code-size, functions are selected for specialization based on certain WCET-aware conditions. For example, a function is specialized if one of the specialized parameters can be used to tighten or determine a previously unknown loop bound inside the cloned function. The authors used their own compiler, the **WCET-aware C compiler (WCC)** [8], to implement this approach. This compiler provides further WCET-oriented optimizations some of which were applied in addition to function cloning.

The approach proposed by Becker et al. [3], utilizes **feedback-directed optimization (FDO)** to use standard compilers for WCET-oriented compilation. To that end, a WCET analysis tool is used to generate a basic block execution profile that is fed to the underlying FDO framework. More specifically, basic blocks are grouped, such that all blocks in a group have a similar execution frequency. Hence, this approach aims at optimizing the binaries produced by a compiler for WCET analysis, rather than increasing the timing predictability of the software itself.

Although, not specifically targeted at optimizing for WCET, the work by Yi et al. [32] can be seen as a generalization of the approach proposed by Becker et al. More specifically, the authors present a scripting language that allows the description of parameterized optimizations or code transformations in general, which are performed by the interpreter of said language. These descriptions could be generated, for example, by an optimizing compiler. By separating parameterized transformation from a compiler, it is possible to use feedback from analyses to search for optimal values of these parameters, for example, WCET analysis. Hence, it would be possible to combine our work with the work of Yi et al. by generating descriptions of the necessary source-to-source transformation rather than performing those transformations directly. However, the scenario-aware analyses necessary to be able to generate those transformations cannot be performed by the work of Yi et al. but rather by the methodology presented in this work.

A machine learning based approach to reduce WCET by optimizing the inlining of functions performed by a compiler was proposed by Lokuciejewski et al. [22]. More precisely, a random forest classifier is used to determine if a function should be inlined. Features are selected with minimization of WCET in mind and consist of, among others, the size and the WCET of the function considered for inlining. Training and validation datasets were created by compiling benchmarks with and without inlining a function call. In case that inlining a call decreases WCET, the corresponding feature vector is associated with the class of to be inlined calls.

Puschner et al. [27] proposed to transform a program into a single execution path to increase timing predictability. To that end, a so-called if-conversion is used, which transforms a branch into a single execution path by using predicated instructions. Thus, this approach is both compiler- and target-architecture dependent, as it only works if predicated instructions are fully supported.

Kafshdooz et al. [18] proposed an approach which generalizes the one developed by Puschner et al. More specifically, a WCET analysis tool is used to determine the WCEP. Then, all functions on this path are considered for transformation, while two basic blocks are merged into a single path only if their WCET is similar.

Barany et al. [2] aimed at simplifying WCET analysis, by annotating flow facts on source level instead of binary level. To that end, the authors developed a source-to-source transformation framework. The latter is used to do optimizations, such as loop unrolling, on source level instead of binary level or **intermediate (IR)** level and to transform the annotated flow facts accordingly.

A framework for transforming flow facts according to optimizations applied to software was developed by Kirner [19]. Flow facts are annotated at source level, but, in contrast to [2], the compiler is allowed to optimize. More specifically, a GCC-based compiler was modified to change flow facts according to optimizations applied to the software.

Park [26] presented an approach that utilizes user-annotated run-time information to reduce the number of feasible paths during analysis. More specifically, a developer is required to annotate flow facts to source-level statements, which are then transformed into sets of program paths that represent the information provided by the user. Then, the resulting, possibly reduced, set of feasible program paths is used to perform static timing analysis. The software under analysis is not changed to reflect the annotated information. Thus, other analyses cannot benefit from refined feasible paths information. In addition, statement-level annotations are rather low level and therefore hard to introduce or maintain during development or evolution of complex embedded software.

Nirkhe et al. [24] used partial evaluation to allow the development of fully predictable software for hard real time systems. To this end, the authors developed a programming language that targets such systems. This language allows a developer to differentiate between compile- and run-time values. The former are values which are known at compile time and restricted to be changed in ways that can be statically analyzed at compile time. Based on this information, partial evaluation is used to create a specialized version of the software that is guaranteed to terminate. Since the halting problem is undecidable, this approach is not applicable for many systems, especially those that are not hard real time. Moreover, the authors stated that partial evaluation often results in code explosion. The latter can be reduced by declaring a value as compile-time only if it is required for determining program termination. This poses a very complex problem for the developer, which further reduces the applicability of the proposed language.

Gheorghita et al. [10] proposed an approach that uses scenarios as well. These are similar but not equal to those scenarios used in this work. In their work, scenarios must cover all possible inputs and the entire application is cloned for each scenario. In addition, scenarios, and what we call instances in this work, are extracted automatically using static analysis. Hence, any information provided by a scenario must already be present in the source code of the application. Then, a modified dead-code elimination is applied to each clone to remove the parts of an application that cannot be executed in the associated scenario. Finally, the WCET is calculated on source level based on a timing schema, which is required as noted by the authors. More precisely, a WCET for each clone of the source code is determined and the WCET of the entire application is defined as the largest of those WCET estimates. The most significant difference to our approach is the definition of a scenario, which is quite restrictive in their case, as the set of extracted scenarios must cover all possible input data. In our work, scenarios may cover only a subset of the input data and it is possible to only partially clone an application. Aside from that, the information provided by a scenario instance is usually not already contained in the source code. Moreover, in addition to cloning parts of the source code, we apply scenario-aware specialization to further improve the effect of cloning on timing predictability.

Fig. 1. Workflow of scenario-aware specialization of embedded software.

3 SCENARIO-AWARE SPECIALIZATION FOR TIMING PREDICTABILITY

In this section, we present our methodology for scenario-aware program specialization. First, we describe the workflow associated with our approach, which is shown in Figure 1. This workflow does not require any changes to generically developed embedded software to qualify as input to our methodology. The first step takes software as input and performs scenario extraction. What follows is a fully automatic process that is the core of our methodology for scenario-aware program specialization. First, so-called *specialization points* (SPs) are created, which basically represent call sites in the program that could be specialized for a given scenario. Next, the created SPs are optimized for the resulting software to meet non-functional requirements or other optimization goals. For example, in addition to improved timing predictability, the binary-size of the specialized software could be constrained. Then, actual specialization is performed based on the optimized set of SPs. Thus, the embedded software provided as input is changed automatically such that the information provided by the supplied scenario instance(s) is reflected in the software. Finally, the specialized software can be deployed on the target system without further changes or adaption. However, it may be necessary to compile the specialized software prior to deployment.

The following sections provide a qualitative comparison of scenarios and flow-fact annotations and go into detail on relevant steps of our workflow. Since our approach is programming-language independent, we provide an implementation for C and C++ in Section 4.

3.1 Scenarios vs. Flow-Fact Annotations

First, note that the main goal of our approach is to reduce the number of annotations necessary for a successful WCET analysis—making annotations optional, if possible. Hence, flow-fact annotations are not regarded as a competing approach but rather as a last resort, in case timing predictability is not sufficient. However, in the following, we provide a thorough discussion and comparison of flow-fact annotations with scenarios.

Flow-fact annotations are the state of the art and most widely used way to provide flow facts to static analyses on different levels of abstraction. This is justified, since annotations are both very flexible and powerful. More precisely, almost any kind of information can be annotated and it is even possible to parameterize analysis algorithms for selected parts of an application. It is also possible to express relations between variables or mark memory regions as read only. Moreover, annotations allow the specification of flow facts on a very fine-grained level, such that it is possible to annotate each statement individually. Yet, the flexibility and expressiveness of annotations come with a cost. First, to be able to fully exploit annotations, an in-depth understanding of both the software under analysis and the analysis itself is necessary. In addition, the more detailed annotations are, the more likely it is that they are affected or invalidated by software evolution. For example, annotating some information at statement level may be affected by renaming a variable or performing reordering calculations inside a function. Finally, annotations are highly tool dependent. Hence, analyzing an application with different analyses, for example, safety analysis on source level and WCET analysis on binary level, generally requires two, possibly overlapping, sets of annotations.

<pre> 1: func EXTRACT(g: CallGraph, max_dist=2) 2: root ← g.root 3: scenarioParams ← {} 4: scenarioScore ← 0 5: for dist ∈ [1, max_dist] do 6: funcs ← CalleesWithDistance(root, dist) 7: params ← {} 8: score ← 0 9: for f ∈ funcs do 10: f_params, f_score ← SCENARIOPARAMS(f) 11: params ← params ∪ f_params 12: score ← score + f_score 13: end for 14: if scenarioScore ≤ score then 15: scenarioScore ← score 16: scenarioParams ← params 17: end if 18: end for 19: GenerateCode(scenarioParams) 20: end func </pre>	<pre> 1: func SCENARIOPARAMS(f) 2: params ← UsableParams(f) 3: subset ← {} 4: subsetScore ← 0 5: for s ∈ P(params) do 6: score ← 7: InfluencedControlFlowStatements(s) 8: if subsetScore ≤ score then 9: subsetScore ← score 10: subset ← s 11: end if 12: end for 13: return subset, subsetScore 14: end func </pre>
(a) Automated Scenario Extraction.	(b) Extract Scenario Parameters.

Fig. 2. Algorithms for automated scenario-parameter extraction.

Scenarios, on the other hand, are much more abstract and only rely on information inherent to the system under analysis. Of course, scenarios are no match for flow-fact annotations regarding expressiveness or granularity, and they are not supposed to be. Instead, since scenarios provide information at function level, they are less likely to be affected by changes to the software. This is especially true for professionally engineered software, as changes to function signatures may break compatibility with possibly large parts of a program or even other programs and are therefore avoided. Thus, scenarios are easier to reuse. For example, a library developer can provide hand-optimized scenarios for a library which can be used easily by users of that library. In addition, scenarios are analysis independent and after scenario-aware specialization, any analysis may benefit from the information provided by a scenario instance. Hence, any analysis or tool performing an analysis can benefit from the increased predictability.

In conclusion, scenarios are a non-competing approach to annotations that provide an easier to use, more resilient, and reusable way to improve the predictability of software.

3.2 Scenario Extraction

As described in Section 1, scenarios capture software and hardware parameters that are generally unknown, but available for a specific embedded system. To make use of scenarios to specialize software, manual labor is required to extract such scenarios for a given system. In the following, we present our automated approach to scenario extraction, which provides the user with a set of possible system parameters, so-called *scenario-parameter candidates*. Hence, the user is responsible for creating scenario instances and determining which of the automatically identified parameter candidates are known at design time. Since scenario instances are always created manually, the main goal of automated scenario-parameter extraction is usability. Thus, a developer should be

Fig. 3. Scenario extraction for the jpeg_wrbmp benchmark.

```

int num_colors = cinfo->
    actual_number_of_colors ; ...
for ( i = 0; i < num_colors; i++ )
    ...
if ( cMap != 0 ) ...
if ( map_entry_size == 4 ) ...
for ( ; i < map_colors; i++ )

```

Fig. 4. Parameters referenced in write_colormap.

able to determine if the extracted candidates are known at design time. Moreover, the number of extracted candidates should be as low as possible to keep scenario-instance creation feasible.

Figure 2(a) shows our algorithm for automatic scenario-parameter candidate extraction. It operates on the **call graph (CG)**, while callees with a maximum distance of *max_dist* are considered during extraction. The default for *max_dist* is 2. This restriction is necessary to prevent an explosion of the number of extracted candidates. For example, Figure 3(a) shows part of the CG of *jpeg_wrbmp*. Since the functions *finish_output_bmp* and *write_colormap* have a distance of one to *main*, we consider those during scenario extraction. Figure 3(b) shows the signatures of those functions.

The set of parameter candidates for a specific distance *dist*, with $1 \leq dist \leq max\_dist$, is created as follows: $params_{dist} = \bigcup_{f \in \text{CalleesWithDistance}(\text{root}, dist)} f_{params}$. Figure 2(b) shows the algorithm for SCENARIOPARAMS which calculates f_{params} . The latter denotes those parameters of function *f* that are suitable for scenario-aware specialization. First, *UsableParams* is called to select parameters of a function that are usable for specialization in general and are likely to improve predictability. Parameters are filtered based on the following criteria: type and access. In case of basic arithmetic types, no pointer or reference types are considered. For compound types, such as C structs and C++ classes, pointers and reference types are allowed. Regarding access, only those parameters that are referenced in certain control-flow statements are considered. These currently are for and while loops and if and switch statements.

For example, the function *write_colormap* (cf. Figure 3(b)) has four parameters, three of which are integer parameters. The first parameter, *cinfo*, is a pointer to a C struct, which consists of more than 60 data members. Figure 4 shows all parameters that are referenced in control-flow statements, as well as the corresponding statement itself. Note that dataflow inside a function is considered as well during this step. Therefore, *cinfo->actual_number_of_colors* is identified to be a usable parameter as well, although it is not directly referenced in a control-flow statement. In this example, filtering for usable parameters leads to a reduction of parameters to be considered from more than 60 to just 4.

Next, the subsets of the set of function parameters returned by *UsableParams* are iterated from smallest to largest set. Hence, filtering for usable parameters is necessary to make iteration of

```

1 struct Scenario {
2   const unsigned int output_height;
3   const int actual_number_of_colors;
4   const int map_colors;
5   const int map_entry_size;
6   const int cMap;
7 };

```

(a) Extracted Scenario for *cjpeg_wrbmp*.

```

1 struct Scenario Instance = {
2   .output_height = 30,
3   .actual_number_of_colors = 256,
4   .map_colors = 768,
5   .map_entry_size = 4,
6   .cMap = 1
7 };

```

(b) Scenario Instance for *cjpeg_wrbmp*.Fig. 5. Results of scenario extraction for *cjpeg_wrbmp*.

all subsets feasible. To select the most promising subset a simple score is calculated for each, as shown in Figure 2(b) (lines 6–7). *InfluencedControlFlowStatements* returns the number of control-flow statements influenced by the parameters of a subset. A control-flow statement is influenced, if its loop bound or conditional expression can be statically derived if the values of all parameters of a subset are constant and known. Finally, the subset with the largest score is returned for each considered function. The score of a set of scenario-parameter candidates is the sum of scores of all contained parameter subsets. Again, the set with the largest score is returned to the user.

Once automatic scenario-parameter candidate extraction finishes, the user creates the final scenario by identifying candidates that are known at design time. To that end, knowledge about the system at hand and its environment is considered. In addition, this knowledge is used to derive one or more scenario instances. If there exists a scenario parameter which has different values at run time and these values are known at design time, there are two options. First, multiple scenario instances are created and there has to be exactly one scenario instance for each possible value of the parameter. Note that Section 3.5.1 discusses the details of multi scenario-instance specialization. Alternatively, the corresponding scenario parameter can be left out of the scenario to prevent creation of multiple scenario instances.

In case of *cjpeg_wrbmp*, all selected scenario-parameter candidates are part of the final scenario as shown in Figure 5(a). Moreover, it is possible to create a single scenario instance, which is shown in Figure 5(b). In accordance with the *cjpeg_wrbmp* benchmark, we assume an application that outputs BMP images with the same size and same number of colors. Hence, the parameters *output_height* and *actual_number_of_colors* are known and constant. Additionally, all output BMP files are colormapped, thus *cMap* is known at design time as well. We also know that the number of bytes per color-map entry is four, which provides a value for *map_entry_size*.

3.3 Specialization-Point Creation

Since, parameters of functions or methods are selected during scenario extraction and a scenario instance provides a *parameter-value map*, functions can be specialized using procedure cloning [6].

To allow for very fine-grained specialization, each call site is considered individually, while specialization of a call site leads to the callee being specialized as well. Hence, it is possible to specialize a subset of call sites to a function. Non-specialized call sites simply call the original, unspecialized function. To simplify handling of call sites selected for specialization, we use *SPs*. An *SP* is a tuple $SP = (CS, M)$, where *CS* is the set of associated call sites and *M* is the set of parameter-value pairs (p, v) that can be used for specialization. More specifically, all call sites $c \in CS$ can be specialized with all pairs $(p, v) \in M$, and parameter *p* is mapped to value *v* during specialization. Therefore, *SPs* associate call sites, which may have different callees, with a set of parameter-value pairs to be used for specialization. Note that there exists exactly one *SP* for a


```

1 param_value_map = {
2   actual_number_of_colors: 256,
3   map_colors: 768,
4   map_entry_size: 4,
5   cMap: 1
6 }
7
8 call_sites = {
9   "main.write_colormap", // first
    call site
10  "main.write_colormap" // second
    call site
11 }

```

Fig. 6. Example SP of `cjpeg_wrbmp`.

<pre> write_colormap(&jpeg_dec_1, 768, 4, 1); ... write_colormap(&jpeg_dec_2, 768, 4, 1); </pre>	$\Rightarrow$	<pre> write_colormap(&jpeg_dec_1); ... write_colormap(&jpeg_dec_2); </pre>

Fig. 7. Call sites to `write_colormap`.

specific parameter–value map. Hence, all call sites that can be specialized with the same set of parameter–value pairs are associated with a single SP.

SPs are created by traversing the CG. All call sites are selected for specialization and SPs are created accordingly. Note that call sites in functions already selected for specialization are also considered as part of this step. Any optimization based on SPs, such as the aforementioned possibility to selectively specialize call sites, is performed during SP optimization.

Figure 6 shows one of the two SPs created for the `cjpeg_wrbmp` benchmark. This SP is created during specialization of call sites from `main` to `write_colormap`. Since `write_colormap` is called twice from `main`, the list of associated call sites (cf. lines 8–11) contains two entries. The parameter–value map is shown in lines 1–6. It maps all affected parameters to their values according to the scenario instance shown in Figure 5(b). Note that the parameter `output_height` of that scenario instance is not part of the created SP, since it is not used in `write_colormap`.

3.4 Specialization-Point Optimization

Using SPs as an abstraction is very flexible and allows a wide range of optimizations. In the following, we present all optimizations that are implemented as part of our framework and applied automatically.

3.4.1 Specialization-Point Refinement. SP refinement is based on the possibility of specializing call-context sensitively. Refining a specialization point $SP = (CS, M)$ is done in two steps. First, a subset of call sites $CS1 \subseteq CS$ that share a set $M1$ of parameter–value pairs is gathered. The latter must meet one important condition, that is $\forall(p, v) : (p, v) \in M1 \Rightarrow \nexists v' : (p, v') \in M$. Note that value v for parameter p has to be statically derivable. Next, a new specialization point $SP1 = (CS1, M')$ is created, where $M' = M \cup M1$. The original SP is updated as follows: $SP = (CS \setminus CS1, M)$.

For example, Figure 7 shows the two call sites from `main` to `write_colormap` of `cjpeg_wrbmp`. Note that three of the four parameters passed to all calls to `write_colormap` are constant values. Using SP refinement it is possible to define a smaller scenario for `cjpeg_wrbmp` than the one shown in Figure 5(a). The resulting scenario consists of the parameters `output_height` and `actual_number_of_colors`.

Fig. 8. Input and result of SP refinement.

```

1 param_value_map = {
2   map_colors: 768
3 }
4
5 call_sites = {
6   "write_colormap.
   funcA"
7 }

```

Fig. 9. SP after propagation.

Figure 8(a) shows the SP created for the call sites shown in Figure 7 and the size-reduced scenario. Figure 8(b) shows the SP after refinement, which is identical to the one created based on the full scenario (cf. Figure 6). The call sites after specialization using the refined SP are shown on the right of Figure 7. In this example, refinement does not create a new SP, since all call sites of the existing SP can be refined with the same set of parameter–value pairs.

3.4.2 Specialization-Point Propagation. Another optimization is *SP propagation*, which is call-context sensitive as well. In contrast to refinement, existing SPs remain unchanged and are used to create new ones. More precisely, let f_{SP} be a function that was specialized for specialization point $SP = (CS, M)$. Now assume, that f_{SP} contains a call site $cs = g(p_0, \dots, p_n)$, $n \in \mathbb{N}$ to function g . Further, assume that g cannot be specialized for any of the user-provided scenario instances. Instead, assume that one or more parameters passed from f_{SP} to g can be statically derived as a result of specializing f with SP . In that case, SP propagation can be used to allow specialization of cs for the statically derivable parameters. Hence, a new specialization point $SP2 = (\{cs\}, M2)$ is created, with $M2 = \{(p_i, v_i) : v_i \text{ statically derivable}\}$. Therefore, SP propagation can be loosely described as specialization-aware context-sensitive constant propagation.

For example, assume that the function `write_colormap` from `cjpeg_wrbmp` calls another function `funcA` with parameter `map_colors`: `funcA(map_colors)`. Although this function was not considered during scenario extraction as performed in Section 3.2, using SP propagation it is possible to specialize it, which yields a new SP as shown in Figure 9.

As mentioned in Section 3.2, we propose to extract scenario parameters based on functions close to CG root. Combined with SP refinement and propagation it is possible to extend the coverage of scenario-aware program specialization throughout the CG. This allows automatically extracted scenarios to perform similar to manually extracted scenarios, as shown in Section 5.2. Moreover, input required from a user can be kept small and rather simple.

<pre> 1: for all functions f in program do 2: if not hasMultipleClones(f) then 3: continue 4: end if 5: callSites $\leftarrow$ getCallSitesTo(f) 6: for $cs \in$ callSites do 7: if not specialized(cs) or 8: partOfMultiSP(cs) then 9: continue 10: end if 11: if not CHECKCALLSITE(cs) then 12: RemoveSpecialization(cs) 13: end if 14: end for 15: end for </pre>	<pre> 1: func CHECKCALLSITE(cs: CallSite) 2: SP = GetSP(CS) 3: Callee = GetCallee(SP) 4: for $L \in$ GetLoops(Callee) do 5: if not BoundForSP(L, SP) then 6: return False 7: end if 8: end for 9: return True 10: end func </pre>
(a) Overall Algorithm.	(b) CheckCallSite.

Fig. 10. Selective multi-clone specialization.

3.4.3 Selective Multi-Clone Specialization. Automatically applying optimizations like SP refinement or propagation may have a negative effect on predictability. It is possible that different call sites to a single function are specialized for different SPs as a result of refinement or propagation. For example, refining two call sites to a function *funcB* for different constant parameter values requires two different clones. In addition, *funcB* may contain a statically unboundable loop. In that case, each clone of *funcB* would increase the number of unbounded loops and therefore decrease predictability. Since the goal of the presented optimizations is to increase predictability and simplify scenario creation, it is necessary to prevent any negative effects. Thus, we use a heuristic called *selective multi-clone specialization* to prevent the creation of multiple clones that may reduce predictability. Multi scenario-instance specialization is not considered, since it is user controlled and targeted at handling dynamic behavior at run time.

Figure 10(a) shows the algorithm that implements our heuristic. The workflow is as follows. For each function f with multiple clones, the set of call sites to f is collected. The actual heuristic is implemented in CHECKCALLSITE, which is shown in Figure 10(b). This function determines if a call site should be removed from the associated SP and therefore not be specialized. More specifically, it makes sure that each loop part of the corresponding callee can be statically bound. To that end, it calls *BoundForSP* that uses the additional value information provided by specializing for SP to derive safe loop bounds.

3.5 Specialization

As previously mentioned, we propose to specialize functions using procedure cloning, which is a transformation applied by compilers to increase optimization potential. However, it is not immediately clear at which level to perform it. In general, specialization can be applied on source level, IR level, or binary level. Performing procedure cloning for predictability on IR level or binary level requires implementing it as part of a compiler. Such a restriction to a single compiler implies the limitation to a more or less fixed set of possible target architectures. Although the resulting binary can be fully controlled from inside a compiler, this requires additional restrictions on the optimizations a compiler is allowed to perform. Moreover, these restrictions can be placed

upon general purpose compilers as well by controlling optimizations via corresponding flags, although more coarse-grained [3]. On the other hand, performing specialization on source level is compiler independent and therefore much more flexible. For example, there is no restriction on possible target platforms, such as ASICs, when using scenario-aware specialization to improve timing predictability. Beyond that, any analysis working on source-level or below benefits from increased source-level predictability. In contrast, increased binary-level predictability, for example, would not affect linting tools or other source-level analyses. In addition, applying specialization on source level can be combined with generating source-level annotations for analysis tools. The latter can significantly improve the results yielded by such a tool [2, 19]. Hence, we propose to apply scenario-aware program specialization at source level.

3.5.1 Multi Scenario-Instance Specialization. It is possible that a system requires multiple scenario instances that are switched at run time. For example, an embedded system that captures and processes images may work with different camera resolutions. In that case, any algorithm that depends on the size of the captured image can be specialized for multiple scenario instances—one for each resolution. More specifically, each call site to such an algorithm can be specialized for two or more scenario instances. Since scenario instances cause creation of SPs, each call site is associated with multiple SPs. Specializing a call site for multiple SPs yields multiple call sites instead of one. Hence, a mechanism is required that allows selection of the appropriate call site at run time, which is based on those scenario parameters that vary between different instances.

To ensure functional correctness, a call site associated with multiple SPs either has to be specialized for all SPs or none. Otherwise, it is possible that a scenario instance is active at run time, for which specialization was not performed. To handle run-time behavior, we propose *scenario guards* to be generated around each call site that is part of multiple SPs. Scenario guards can be implemented using conditional statements available in any modern programming language. More specifically, for a specialization point $SP = (CS, M)$ and a call site $cs \in CS$, a guard checks that $a_p == v$ holds for all parameter–value pairs (p, v) and the corresponding call argument a_p . That way, it is guaranteed at run time that the correct version of a function is used for each active scenario.

Note that we assume that scenario instances are switched implicitly at run time. For example, an embedded system that captures and processes images may switch between scenario instances by resizing a captured image or changing camera resolution at run time. Therefore, it is not necessary to manually adopt the transformed program to perform scenario-instance switches, since the currently active instance is automatically detected using scenario guards. Our current model of multi scenario-instance specialization does not support asynchronous scenario-instance changes that may lead to a change during execution of a specialized function.

4 SCENARIO-AWARE SPECIALIZATION FOR C/C++

As mentioned in Section 1, we implemented our methodology in a framework for automated scenario-aware program specialization of C and C++ software. Furthermore, we propose to apply specialization on source level. Hence, a framework for source level cross translation unit analysis and transformation was developed based on the *LibTooling* library provided by the clang compiler framework [30]. On top of this tool, we implemented our approach as described in the previous section and the source-to-source transformations discussed in the following.

4.1 Function Specialization

Once SPs are optimized, scenario-aware procedure cloning is done in two phases: SP to function parameter mapping followed by standard procedure cloning. Currently, parameter mapping is

<pre> 1 void main() { 2 finish_output_bmp(&jpeg_dec_1); 3 write_colormap(&jpeg_dec_1, 4 768, 4, 1); 5 ... 6 } </pre>	⇒	<pre> 1 void main() { 2 finish_output_bmp(&jpeg_dec_1); 3 write_colormap(&jpeg_dec_1); 4 ... 5 } 6 } </pre>

(a) Specialization of Call Sites in Function main.

<pre> 1 void write_colormap(2 j_decompress_ptr cinfo, 3 int map_colors, 4 int map_entry_size, 5 int cMap) { 6 ... 7 } </pre>	⇒	<pre> 1 void write_colormap(2 j_decompress_ptr cinfo) { 3 const int cinfo_actual_number_of_colors = 256; 4 const int map_colors = 768; 5 const int map_entry_size = 4; 6 const int cMap = 1; 7 ... 8 const int num_colors = cinfo_actual_number_of_colors; 9 ... 10 } </pre>

(b) Specialization of write_colormap.

Fig. 11. Before and after specialization of cjpeg_wrbmp.

done by name. Therefore, a function parameter p_f is specialized using the value of the parameter-value pair (p, v) , if $\text{name}(p) == \text{name}(p_f)$. In addition, parameters have to have the same type. If the data type of a function parameter is composite or a pointer or reference to it, each data member of that composite type is considered during parameter mapping. Hence, it is possible to specialize only a subset of the data members of a struct or class. Mapping by parameter-name equality is currently done for simplicity and can be replaced by more sophisticated approaches. For example, using a user-defined mapping between scenario and function parameter.

A function is cloned for a specialization point $SP = (CS, M)$ by removing non-composite-type parameters p , with $(p, v) \in M$ from the function definition and all (re-)declarations. Moreover, a local constant is defined at the start of the function body with $\text{const type}(p) \text{ name}(p) = v$ for all pairs $(p, v) \in M$. These definitions appear in the same order and have the same type as the corresponding parameters in the original function signature. For a member of composite data type, all accesses to that member inside the cloned function will be replaced with accesses to the corresponding local constant. Note that we assume that any member accesses related to a parameter p are of the form $p \rightarrow \text{member}$.

For example, Figure 11(b) shows how `write_colormap` of the benchmark `cjpeg_wrbmp` is specialized. Parameters with arithmetic type are removed and a local constant with the same name and type is created. In addition, this variable is initialized with the value provided by the associated SP. A member of a struct or class is specialized similarly, while the name of the local constant is prefixed with the name of the corresponding parameter. Hence, the member `actual_number_of_colors` of parameter `cinfo` is transformed to a local constant with the name `cinfo_actual_number_of_colors`. As shown in Figure 11(a), all call sites to `write_colormap` are transformed by removing all specialized, non-composite type parameters.

4.2 Propagating Loop Bounds to Binary Level

Having statically derivable loop bounds on source level does not guarantee that this information is still available on binary level. This is, among other reasons, due to compiler optimizations. For example, modern compilers like GNU GCC perform induction variable merging and elimination. Such optimizations may lead to multiple nested loops sharing a single induction variable.

We present a source-to-source transformation that aims at preserving loop bounds from source to binary level. More specifically, the goal of this transformation is to increase the number of automatically detected binary-level loop bounds without the need for annotations. This transformation is based on what we call a *count variable* of a loop. The latter is an induction variable of a loop that has to be of integral type. In addition, it must be possible to statically derive a lower and an upper iteration bound for this loop. A loop may either increase or decrease its count variable and any change to a count variable must be performed using any of the basic arithmetic operators. If a loop does not meet those requirements, the transformation presented in this section is not applied. Given a loop with lower bound l and upper bound u , we change each write access to a count variable with one of the following transformations:

```

1 countVar = newVal  $\Rightarrow$  (countVar = (l <= newVal && u >= newVal) ? newVal : 0/0, newVal)
2 countVar = newVal  $\Rightarrow$  (countVar = (l <= newVal && u >= newVal) ? newVal : 0/0, oldVal)

```

It is possible to transform any write access to an integer variable such that the corresponding change is applied using a single assignment expression. Since assignments are expressions in C and C++, the transformations must produce valid expressions with correct values. Hence, the comma-operator is used, which is an expression, and its subexpression is used as value of the entire expression. More specifically, the postfix and prefix operators ++ and -- can be transformed to an assignment similar to $countVar = countVar + 1$ and $countVar = countVar - 1$, respectively. For postfix operators the transformation in line 2 would be used while prefix operators can be handled using the transformation in line 1. Compound operators like += or -= can be transformed analogously.

Using this transformation, an assignment or change to an induction variable is only valid if it is bounded by the statically derived upper and lower bound. Otherwise, a division by zero would occur which usually leads to a hardware exception being triggered. Any other kind of control-flow terminating statement is suitable as well, for example, an assertion. Terminating the control flow is important as it allows a static analysis to detect the value of the corresponding variable for all valid execution paths. Such a termination is not going to happen at run time, since the bounds used in this transformation are derived using safe static analysis and therefore guaranteed to be valid.

It is possible that this transformation prevents some compiler optimizations or reduces the efficiency of those. Although this is a tradeoff to be paid when aiming for binary-level timing predictability, this transformation needs to be applied with care and may require feedback from an analysis tool. In our framework, we currently use feedback from aiT [11] to determine if applying this transformation increases timing predictability as intended.

5 EVALUATION

To evaluate our approach, we used almost all benchmarks from the TACLeBench benchmark suite [7]. For comparison with related work, we used the entire Mälardalen benchmark suite [14], a subset of the MiBench [16] benchmarks and the DSPStone [33] fixed-point benchmarks. We did not use the two *parallel* benchmarks from TACLeBench, since aiT does not support analysis of concurrently executed software [12]. In case of MiBench, we only used those benchmarks that were necessary for a complete comparison with related work. For compilation of the benchmarks, we used the ARM GCC toolchain version 7.3.1 and aiT for ARM version 19.04 for WCET analysis. All analyses executed with aiT used an ARM Cortex-M4F as target processor. We used the TACLeBench collection with commit id a34b46c2 and retrieved the Mälardalen, MiBench, and DSP-Stone benchmark suites from the official homepages [13], respectively, [15] and [17].

In the following sections, we describe each performed evaluation along with the associated experimental setup and methodology. In addition, we present and discuss the corresponding results in each section. Note that all WCET estimates reported in the following sections refer to the WCET as reported by aiT, if not explicitly stated otherwise. Since the number of benchmarks used in all the following evaluations is quite large, we did not annotate exact flow facts on binary level. Instead, we manually analyzed each benchmark to derive an upper bound for all loop bounds of a benchmark. Then aiT was instructed to use this upper bound as a fallback in case it was unable to automatically determine a loop bound.

5.1 Comparison to Related Work

In addition to the qualitative benefits of our work compared to the methodologies utilized by our related works, as discussed in Sections 3.1, 3.5, and 2, we want to evaluate if our approach performs at least as good as the related work w.r.t. WCET predictability. For the comparison, we selected related work with the most extensive evaluation using larger subsets of established benchmark suites, namely Mälardalen, MiBench, and DSPStone. More specifically, we chose the work of Barany et al. [2] and Kafshdooz et al. [18]. These cover a wide range of methodologies that are used in many of the other related works discussed in Section 2. The approach of Barany et al. combines source-level analyses and transformations with the transformation and generation of tool-specific flow-fact annotations. More specifically, loop unrolling, fusion, blocking, or interchange are performed on source level instead of binary level and annotations are transformed accordingly. Thus, it can be beneficial to combine their approach with ours, since increased source-level predictability can improve the results of the analyses and transformations performed by their work. In contrast, Kafshdooz et al. merge binary-level basic blocks that are part of different branches of a conditional jump. To determine if basic blocks should be merged, a WCET analysis is performed. Hence, combining our approach with theirs can be valuable, since applying our approach can lead to much more precise WCET analysis results.

5.1.1 Experimental Methodology. To compare our approach with the work of Barany et al., we used the complete Mälardalen benchmark suite and the fixed-point version of the DSPStone benchmark. For comparison with Kafshdooz et al., we used the same subsets of Mälardalen and MiBench as the related work. In both cases, the relative WCET, as reported by the related work,² was used for comparison. We chose optimization level *O1* for comparison with Barany et al. and optimization level *O3* for comparison with Kafshdooz et al.

First, we used aiT to calculate a reference WCET. Then, we used manually extracted scenarios to transform each benchmark without any SP optimizations. We chose this approach as manually extracted scenarios without SP optimizations give an experienced user the most flexibility and control. Next, aiT was used to determine a WCET estimate for the transformed benchmarks. Finally, a relative WCET was calculated as discussed in the following section.

5.1.2 Results and Discussion. All figures in this section share the same semantics and we use Figure 12 as an example to discuss it. On the y-axis, the relative WCET ($WCET_{rel}$) of each benchmark is shown, with $WCET_{rel} = \frac{WCET_{trans}}{WCET_{orig}} \cdot 100$. $WCET_{trans}$ denotes the WCET of the benchmark transformed with our framework. $WCET_{orig}$ is the WCET of the original, unchanged benchmark. Due to space restrictions, we only show results for those benchmarks where $WCET_{trans} \neq WCET_{orig}$. As mentioned previously, both $WCET_{orig}$ and $WCET_{trans}$ denote the WCET estimate returned by aiT. The gray horizontal line in Figure 12 marks the reference WCET

²In case of Barany et al., we were provided with the original data by the authors. Unfortunately, we had to get the data of the work from Kafshdooz et al. from their publication, since we were unable to reach any of the authors.

Fig. 12. Comparison with Kafshdooz et al.: benchmarks compiled with O3.

(a) DSPStone benchmarks compiled with O1.

(b) Mälardalen benchmarks compiled with O1.

Fig. 13. Comparison with Barany et al.

of the original benchmarks. The dark gray bars show the relative WCET of the benchmarks after applying our methodology, while the light gray bars represent the results of the corresponding related work. In case of Figure 12, these are the results from Kafshdooz et al.

As already mentioned, Figure 12 shows the results for comparison with the work of Kafshdooz et al. First, we achieve an average reduction of WCET of about 17%, which is slightly lower than that achieved by the related work (about 20%). This is caused by the fact that there was no opportunity for scenario extraction for most of these benchmarks. Unfortunately, many benchmarks have this property, since system parameters are often hard-coded constants instead of variables. On the other hand, we extracted a scenario for *compress*, *crc*, *ludcmp*, *minver*, and *susan*. For these benchmarks, the average reduction of WCET is about 45% and in almost all cases, except for *minver*, we achieve significantly better reductions than the related work. The large reduction in case of *susan* is caused by aiT being able to precisely bound all loops that are on the WCEP. This is due to both specialization and the transformation presented in Section 4.2. In addition, the compiler can eliminate dead code more efficiently which also improves the WCET estimate returned by aiT. Finally, note that it is possible to combine our approach with the one proposed by Kafshdooz et al. to further enhance predictability.

Figure 13(a) shows the results of this work and the work of Barany et al. for the fixed-point version of the DSPStone benchmarks. In contrast to the related work, most benchmarks are unchanged by our approach, which is due to the simplicity of these benchmarks. Most benchmarks consist of only one or two functions and, more importantly, there are only a few system parameters, almost

all of which are hard-coded constants. Hence, there was no opportunity for scenario extraction for any of those benchmarks. Thus, the few benchmarks with large WCET reductions, namely *fir2dim*, *matrix1*, and *matrix2*, benefit from the transformation to preserve binary-level loop bounds. More specifically, in case of these benchmarks aiT was able to precisely bound all loops thanks to the transformation applied by our framework. In contrast, aiT was unable to bound any loop of the original, unchanged benchmarks. These rather extreme reductions can be partially attributed to the fact that aiT uses a default loop bound as fallback. This leads to an average reduction of WCET, as reported by aiT, of about 26%, which is slightly better than the related work (23%). In addition, WCET was never increased in contrast to the work of Barany et al. (cf. *fir2dim*).

Figure 13(b) shows results for the Mälardalen benchmark suite of Barany et al. and this work. For some benchmarks, for example *compress*, there is no result from the related work, since these benchmarks were not part of their evaluation. We were able to extract a scenario for *compress*, *crc*, *duff*, *edn*, *expint*, *fdct*, *fft1*, *ludcmp*, *minver*, and *ud*. Due to scenario-aware specialization, we achieved an average reduction of WCET of about 35% for these benchmarks. In addition, reductions for all these benchmarks, except *expint* and *minver*, were significantly better compared to related work. Moreover, predictability was never decreased, in contrast to the related work where the WCET of *crc* and *whet* is increased significantly. In case of *fft1*, due to specialization and the transformation to preserve binary-level loop bounds, aiT was able to bound all loops part of the WCEP. In addition, the compiler was much more effective during optimization, especially dead-code elimination. The WCET improvements of all other benchmarks are solely caused by the transition to preserve binary-level loop bounds. Overall, we achieve an average reduction of WCET of about 20%, which is slightly better than the related work (about 15%). Again note, that the approach presented in this work and the work by Barany et al. could be combined to achieve even better improvements.

5.2 Manual vs. Automatic Extracted Scenarios

To evaluate the performance of automatically extracted scenarios compared to manually extracted scenarios, we used the benchmarks from the TACLeBench benchmark suite. There are several reasons why we chose this benchmark suite for this evaluation. First, TACLeBench is the newest and largest set of benchmarks targeting timing analysis research. Moreover, it is widely used and contains many benchmarks from previously published suites such as Mälardalen or MiBench. Finally, TACLeBench covers a large range of application domains and contains both simple synthetic benchmarks and large application benchmarks based on real-world applications or algorithm implementations. As mentioned previously, the benchmarks of the category *parallel* could not be used due to limitations in aiT.

5.2.1 Experimental Methodology. First, each of the original benchmarks was analyzed by aiT to calculate a reference WCET for each benchmark. Then, our framework was executed to transform each benchmark according to the following configurations. First, only scenario-aware procedure cloning without any SP optimizations was performed (*Scenario only*) and then specialization with all SP optimizations (*SP optimizations*).

Next, aiT was used to calculate the WCET of each benchmark for each of those configurations. More specifically, all versions of the benchmarks (original, scenarios only, and SP optimizations) were compiled with compiler-optimization levels *O1* and *Os*. For manual as well as automated scenario extraction, we created a single scenario instance where applicable. As mentioned previously, benchmarks used in timing analysis research, such as TACLeBench or Mälardalen, are often optimized for WCET analysis. Thus, most of these benchmarks already have a good timing predictability w.r.t. WCET analysis. Therefore, we created scenarios for 16 out of 56 benchmarks and

Fig. 14. Manual vs. automated scenario extraction: results for O1.

a single instance for each scenario during manual scenario extraction. Automatic scenario extraction was performed with a *max_dist* of 1 and yielded 16 scenarios and we manually created an instance for each.

5.2.2 Manual Scenario Extraction. Manual scenario extraction was performed similar to the approach for automatic extraction as described in Section 3.2. More specifically, to extract scenario parameters for a given benchmark, we first inspect all functions called from the main function. Next, we move along the different paths through the CG that start with those initially identified call sites. We scan each function for parameter uses and determine if knowing this parameter at analysis time may have a positive impact on timing predictability. A parameter is determined to possibly improve timing predictability if its knowledge makes it feasible to statically derive the value of a loop bound or boolean expression. A developer with proper understanding of the source code at hand is much more capable at obtaining loop bounds or boolean expressions than the heuristics presented in Section 3.2. Hence, this last step is an important difference to automatic extraction. Another difference is that we do not use something like *max_dist* when manually extracting scenario parameters. In addition, a developer may recognize or know that one or more parameters are passed down along paths of the CG and select this parameter if it may improve predictability in any of the functions along that path. Thus, when manually extracting scenarios, we may do what some of the SP optimizations, intended for automated scenario extraction, do.

5.2.3 Results and Discussion. In this section, we present and discuss the results of our evaluation to compare the effectiveness of automated scenario extraction with manual scenario extraction. All figures in this section have the same semantics, which we discuss in the following using Figure 14(a). The line parallel to the *x*-axis denotes the reference WCET of the original benchmarks. The bar plots show the WCET of the transformed benchmarks and are labeled with their exact value rounded to one decimal place. The dark-gray bars represent the WCET for manual scenario extraction, while the light-gray bars represent the WCET for automated scenario extraction.

Figure 14(a) shows the relative WCET, that results from specialization without SP optimizations, for benchmarks compiled with optimization level *O1*. The WCET of *cjpeg_wrbmp* is reduced by about 46% and there is no difference between manually and automatically extracted scenarios.

Fig. 15. Manual vs. automated scenario extraction: results for Os.

The scenario shown in Figure 5(a) was used in both cases. Thus, for *cjpeg_wrbmp*, our automated scenario extraction yielded the same scenario parameters as manual extraction. In addition, aiT was able to detect one of four loop bounds for the original benchmark. In contrast, all loop bounds were detected for the transformed benchmark. The bound detected in both the original and transformed benchmark was more precise in case of the transformed benchmark. Also, the parameter *cMap* part of the scenario for *cjpeg_wrbmp* improved infeasible paths detection. The results for the benchmarks *fir2dim*, *gsm_enc*, and *ludcmp* can all be attributed to an increased number of detected loop bounds. In case of *fir2dim*, no scenarios were extracted. Hence, the improved WCET is caused by the transformation to preserve loop bounds at the binary level (c.f. Section 4.2). The manually extracted scenario for *gsm_enc* contains a single parameter, while automatic extraction was unable to find suitable parameters. For *ludcmp*, both manual and automatic extraction yielded a scenario. Although the automatically extracted scenario is a subset of the manually extracted one, the respective WCETs are quite close. In case of *h264_dec*, the extracted scenarios differ most. The scenario created manually contains 11 parameters and the automatically extracted contains three. These three parameters are shared by both scenarios. The improvement is quite large, for manual extraction about 94% and about 75% for automatic extraction. In case of manual extraction, aiT was able to derive all loop bounds for the transformed benchmark, while only two of seven bounds were detected for the original benchmark. On the other hand, aiT was able to bound one more loop compared to the original benchmark for automatic scenario extraction.

Figure 14(b) shows the relative WCET, that results from specialization with SP optimizations, for benchmarks compiled with optimization level *O1*. SP optimizations introduced a small increase of WCET of up to 0.5% for the *adpcm* benchmarks and one of 1.3% for *gsm_dec*. The WCET of *gsm_enc* was slightly increased compared to the benchmarks without SP optimizations, while it was still reduced by about 74% compared to the original benchmark. On the other hand, many results improved, especially for automatic scenario extraction. In case of *audiobeam*, *ludcmp*, and *mpeg2*, the results for automatically extracted scenarios now are the same as for manually extracted scenarios. In addition, the results of *g723_enc* and *minver* do not significantly differ between automatically and manually extracted scenarios. This is in contrast with the results without SP optimizations: automatically extracted scenarios either performed worse than manually extracted scenarios for

these benchmarks or did not improve WCET at all. Compared to the results from unoptimized SPs, results for manually extracted scenarios could be improved as well. The WCET of *g723_enc* was further reduced, while improving the WCET of *mpeg2* was only possible with SP optimizations.

Figure 15(a) shows the relative WCET for benchmarks compiled with optimization level *O_s*. SP optimizations were not performed. There are several large improvements of predictability in terms of WCET. These can be partially attributed to the fact that aiT was instructed to use a benchmark-wide default loop bound for unbounded loops. On the other hand, the transformations performed by our framework lead to an improvement even if aiT was able to bound all loops. For example, the WCET of *bsort* was reduced by about 44% due to the applied transformations. In that case, aiT derived all loop bounds for the transformed and the original benchmark. In addition, these bounds were derived with the same accuracy. Applying scenario-aware program specialization improves the ability of static analyses to determine, for example, value information or feasible paths information. Compiler optimizations like dead-code elimination profit from these transformations as well. In general, compilers can optimize more effectively due the additionally available value information provided by scenario-aware program specialization.

Figure 15(b) shows the relative WCET for benchmarks compiled with optimization level *O_s* and SP optimizations. For *adpcm_enc*, WCET was slightly increased by 0.2% independent of the used scenario extraction method. In case of *minver*, the WCET was increased by 1.5% for automatically extracted scenarios only. On the other hand, results were improved significantly for several other benchmarks. The results for automatically extracted scenarios equal those of manual extraction for *audiobeam* and *ludcmp*. Moreover, there is no significant difference between results from manual and automatic scenario extraction in case of *g723_enc*. The improvements of WCET for *mpeg2* and *rijndael_dec* were enabled by SP optimizations.

5.3 Effect of “Extraction Depth” on Automatically Extracted Scenarios

Since the algorithm for automated scenario extraction has a parameter, called *max_dist*, we evaluated the impact of this parameter on the extracted scenarios and the resulting timing predictability w.r.t. WCET analysis. We used the TACLeBench benchmark suite to perform this evaluation and the same experimental setup as in Section 5.2.

5.3.1 Experimental Methodology. To evaluate the impact of *max_dist* on timing predictability, we performed automated extraction for different values of *max_dist* with $1 \leq \text{max\_dist} \leq \text{max}_{cg\_depth}$ and $\text{max}_{cg\_depth} = \max(\{cg\_depth(b) | b \in \text{benchmarks}\})$. The evaluation to compare manual and automated scenario extraction was conducted exactly as described in Section 5.2. For each benchmark and a certain value *n* of *max_dist*, we only included the corresponding scenarios in our evaluation if there was no value *n'* < *n* that yielded the exact same scenario.

5.3.2 Results and Discussion. First, for values *n* of *max_dist* with $n \geq 6$, no new scenarios were extracted for any benchmark. More importantly, there was no significant change in timing predictability w.r.t. WCET analysis for any possible value of *max_dist* ≤ 5 compared to *max_dist* = 2. With all SP optimizations enabled, there was no difference at all and only a small difference when disabling all SP optimizations. In case of optimization level *O₁*, the relative WCET of *g723_enc* was improved to 98.4% (previously 100%) for *max_dist* ≥ 2 and is the same as for manual scenario extraction. Similarly, for optimization level *O_s*, only the relative WCET of *g723_enc* was improved to 97.9% for *max_dist* ≥ 2 , which is closer to manual scenario extraction.

There are several reasons why increasing *max_dist* did not help for almost all benchmarks whereas SP optimizations did. First, the algorithm shown in Figure 2(a) is a heuristic that returns a subset of parameters that has the largest possible score and there exists no set, which has the same score and a lower cardinality. Thus, the automatically extracted scenario may be missing

Benchmark	$WCET_{none}$	$WCET_{all}$	$WCET_{s0}$	$\frac{WCET_{all}}{WCET_{s0}}$	$\frac{WCET_{all}}{WCET_{none}}$	$\frac{size_{all}}{size_{none}}$
<i>cjpeg_wrbmp</i> O1	474424	321510	321417	1.00	0.68	1.03
<i>cjpeg_wrbmp</i> Os	541941	374766	374707	1.00	0.69	1.01
<i>h264_dec</i> O1	344981846	255370472	255340208	1.00	0.74	1.06
<i>h264_dec</i> Os	347561523	6271352	6271200	1.00	0.02	1.03

Fig. 16. Multi scenario-instance specialization results for *h264_dec* and *cjpeg_wrbmp*.

parameters because the heuristic prefers smaller subsets, which is done to keep the complexity of manual scenario-instance creation as low as possible. Moreover, the heuristic chooses *one* of possibly many sets with the same cardinality and the highest score. Hence, it is possible that a different subset has a greater impact on predictability. More importantly, to calculate the score of a subset, the number of influenced control-flow statements is derived, which is done by a very simple static analysis. In contrast, an experienced developer can perform much more complex analyses to determine the analyzability of expressions or statements. In conclusion, it is most likely that the simplicity of the heuristic, which is intentional since the number of subsets to be inspected can become quite large, is the reason why increasing *max_dist* does not further improve WCET.

Again, the purpose of scenario extraction is to support a user during scenario extraction without increasing complexity by returning large sets of scenario-parameter candidates. Moreover, automatic scenario extraction is intended to be used with SP optimizations, especially SP propagation, to maximize the effect of specialization throughout the entire application. Finally, SP optimizations indeed close the gap between manual and automatic scenario extraction in most cases, for example, *audiobeam* and *duff* for optimization level *O1* or *g723_enc* for *Os* and hence, work as intended.

5.4 Case Study: Multi Scenario-Instance Specialization

Since support for multiple scenario instances is beneficial for real-world embedded software, we performed a case study to evaluate if scenario guards significantly decrease predictability or introduce a large run-time overhead.

5.4.1 Experimental Methodology. To investigate the effect of multi scenario-instance specialization compared to single scenario instances, we used the benchmarks *cjpeg_wrbmp* and *h264_dec*. These are among the best performing benchmarks for single scenario instances. Thus, they are best suited to show the effect of specialization for multiple scenario instances on the improvement of WCET compared to single-instance specialization. For both benchmarks, the same manually extracted scenarios as in the previous section were used. In contrast to real-world software, benchmarks often contain values, that are unknown in general, as hard-coded constants. For example, as local or global variables with `const` qualifier or preprocessor defines. Thus, to be able to specialize these benchmarks for multiple scenario instances, we replaced some hard-coded constants with variables.

Four scenario instances with different parameter-value maps were created and six versions of each benchmark. More specifically, the original benchmark corresponds to the version *none*. We also created a version *all*, that was specialized for all four scenario instances. Hence, all functions that were specialized were cloned four times and the corresponding call sites were guarded using scenario guards. In addition, we created a single-instance version of each benchmark for each scenario instance, named s_i , $i \in \{0, 1, 2, 3\}$. To evaluate the impact of multi-instance specialization

Version	WCET [%]	WCET [cycles]	Unbounded Loops	Tighter Bounds
Original	100	711193776568	167	0
Specialized	92.53	658035171598	144	8

Fig. 17. Results for ARM CMSIS-NN CIFAR-10 application.

on timing predictability, we need to verify that the following equation holds:

$$WCET_{all} \approx \max_{0 \leq i \leq 3} WCET_{s_i}. \quad (1)$$

Thus, the WCET of the multi scenario-instance variant should be as close as possible to the WCET of the single scenario-instance variant with the largest WCET. If that is the case, specializing for multiple scenario instances does not lead to a (significant) loss of predictability w.r.t. WCET. Note that the scenario instances were constructed such that $WCET_{s_0} = \max_{0 \leq i \leq 3} WCET_{s_i}$. No SP optimizations were performed, and each version was compiled with *O1* and *O5*.

5.4.2 Results and Discussion. To check against the metric shown in Equation (1), we calculate the ratios presented in Figure 16 for both benchmarks and optimization levels. Note that Figure 16 shows all absolute WCETs, used to calculate these ratios. The first two rows correspond to *cjpeg_wrbmp* and the following rows to *h264_dec*, while all ratios are rounded to two decimal places. Independent of optimization level, the ratio $\frac{WCET_{all}}{WCET_{s_0}}$ indicates no significant loss of predictability w.r.t. WCET. Also, WCET of the multi scenario-instance variant could be improved significantly compared to the original benchmark (*none*), as shown by ratio $\frac{WCET_{all}}{WCET_{none}}$. Again, aiT was able to precisely bound all loops for all specialized variants, while only a single bound was derived for the original benchmark. Of course, specializing for multiple scenario instances may increase binary size significantly. As the latter is an important property for many embedded systems, the last column shows the ratio $\frac{size_{all}}{size_{none}}$. There is no significant overhead in binary size for any used optimization level or benchmark. This can be explained in a similar way as the improvement of WCET: additionally available value and data-flow information have a significant effect on the ability of the compiler to apply optimizations. For example, the *cMap* parameter of *write_colormap* (cf. Figure 4) allows GCC to classify a significant part of this function as unreachable. A similar effect is seen in the specialization of parameter *map_entry_size* of the same function.

5.5 Case Study: Neural Network Inference on Edge Devices

To demonstrate the applicability and maturity of our approach, we performed a case study using an application that performs **neural network** (NN) inference on a Cortex-M4-based device. There is an increasing need for NN inference within the domain of embedded systems, for example, in IoT where inference is moved to the edge due to security or connectivity concerns among other issues [28]. Another example is autonomous drones that are deployed as mobile IoT nodes, where inference of **convolutional neural networks** (CNNs), such as DroNet [23], is performed on device [25]. We used the ARM CMSIS-NN CIFAR-10 example application [1], which utilizes the ARM CMSIS-NN library for Cortex-M CPUs [20]. The application captures an image and associates it to 1 of 10 classes. Moreover, images are displayed and then resized to fit the input size of the CNN. Finally, a prediction is performed and displayed as well. The used CNN consists of 8 layers and operates on images with a resolution of 32×32 pixels. The software also consists of low-level device drivers and performs all necessary initialization of connected peripherals, for example, the display. The CG of the application consists of 61 functions and 101 edges.

5.5.1 Experimental Methodology. We used information about the image sizes and the predictability-relevant parameters of the CNN model for manual scenario creation. Important model parameters are input/output dimensions, kernel dimensions, number of channels, and stride. Since the application contains a large set of functions and has a complex CG, we used specialization with all SP optimizations. Then, we used aiT to calculate a WCET estimate for the original as well as the specialized benchmark.

5.5.2 Results and Discussion. Figure 17 shows the results of our case study. In addition to relative and absolute WCET, it shows the number of unbounded loops for each version of the application. Furthermore, for those loops, aiT was able to bound in both versions, it shows the number of bounds that were more precise.

WCET was reduced by about 7.5%. The number of loops aiT was unable to bound was reduced by 23 for the specialized application. In addition, precision of loop bound analysis was improved for 8 loops. Hence, predictability w.r.t. WCET and loop bound analysis was improved significantly compared to the original version of the application.

6 CONCLUSION AND FUTURE WORK

In this article, an approach to scenario-aware program specialization for improving (timing) predictability is presented, along with an implementation for C and C++. In the area of timing analysis, embedded software that is hard to statically analyze is a major obstacle in the way of fast and precise analyses. The reason for this challenge is two-fold. First, software is usually developed generically to keep development cost as low as possible. Hence, although some flow facts are known at compile time, it is not possible to express these, as this would contradict the generality of the software. Furthermore, compiler optimizations, which are important for performance and binary size, make preservation of flow facts to binary level much harder in many cases.

To conquer these problems, we propose scenario-aware program specialization of embedded software. Moreover, we provide a source-to-source transformation to preserve loop bounds from source to binary level. We also developed a framework for scenario handling, which supports automated scenario extraction and allows automatic scenario-aware code transformations to improve predictability. We implemented optimizations with the goal of maximizing the effectiveness of scenarios from automated scenario extraction. To make sure that optimizations improve predictability, a heuristic was developed to prevent cases where predictability might be decreased.

We performed a thorough evaluation-based comparison with related work. More specifically, we compared our approach with that of Barany et al. [2] and Kafshdooz et al. [18]. The comparison with Kafshdooz et al. was conducted on a subset of the benchmarks from MiBench and Mälardalen. While achieving a slightly smaller average reduction of WCET, because many of the evaluated benchmarks are already optimized for predictability, it showed that our approach caused a significantly larger improvement of predictability w.r.t. WCET for those benchmarks that allowed manual extraction of a scenario. In case of the approach of Barany et al., we used all benchmarks from Mälardalen as well as the fixed-point benchmarks from DSPStone. Although many of these are already optimized for timing analysis, we achieved a larger average reduction of WCET for both benchmark suites. For those benchmarks where scenario extraction was possible, an average reduction of about 45% and 35% was reached for DSPStone, respectively, Mälardalen. In conclusion, the comparison with related work showed that our approach performs similarly w.r.t. WCET predictability and significantly better in most of those cases where scenario extraction was feasible.

An extensive evaluation was performed to show the effectiveness of automated compared to manual scenario extraction using the TACLeBench benchmark suite. Average improvements of up to 29% were achieved for automatically extracted scenarios and of up to 31% for manually extracted

scenarios. In addition, applying the developed optimizations caused automatically extracted scenarios to perform very close to manually extracted scenarios. More specifically, manually extracted scenarios perform significantly better for only 2 out of 12 benchmarks for optimization level *O1*. In case of optimization level *O0*, results for only 4 out of 12 benchmarks are significantly better for manually extracted scenarios. Hence, it was possible to significantly improve predictability w.r.t. WCET for both manually and automatically extracted scenarios.

Evaluating the impact of the *max_dist* parameter on extracted scenario-parameter candidates, and therefore the possible influence on predictability w.r.t. WCET, shows that increasing *max_dist* is not necessary. Since increasing *max_dist* is likely to cause larger sets of scenario-parameter candidates, this is intended. More specifically, the set of parameters returned by the algorithm for automated scenario extraction should be as small as possible for manual scenario-instance creation to be feasible. Of course, the positive effect on timing predictability of such a set of parameters should be as large as possible. In addition, automated scenario extraction should be used together with SP optimizations for optimal performance. Therefore, the purpose of automated scenario extraction, which is usability, would be invalidated if a user had to find an optimal value for *max_dist* without assistance. In conclusion, the evaluation regarding the impact of *max_dist* on predictability has shown that the latter works as intended.

Our first case study indicates that multi scenario-instance specialization performs similarly to single scenario-instance specialization, since there is no difference between WCETs for corresponding variants of the used benchmarks. Moreover, performing specialization for multiple scenario instances did not create a large overhead in binary size, as the latter increased by at most 6% although four different versions were generated for each specialized function.

Finally, our case study based on a large real-world application demonstrates that our approach can be used with complex embedded software that consists of thousands of lines of code. It was possible to reduce the WCET estimate and the number of unbounded loops. In addition, precision of loop bound analysis was improved as well.

In conclusion, we showed that our approach can be successfully used to significantly improve the timing predictability of embedded software w.r.t. WCET, as reductions of up to 99% were achieved. Although some improvements were moderate, these may be all that is needed to reduce the WCET of an application to satisfy associated timing requirements. In comparison with related work, our approach performed as good as or better on average, even though many benchmarks were already optimized for WCET analysis on source level. In case of those benchmarks that allowed scenario-aware program specialization, reduction of WCET was significantly larger in almost all cases compared to related work.

For future work, we plan to improve our algorithm for automated scenario extraction by implementing a more advanced loop bound analysis. Also, we plan to evaluate the impact of different metrics on parameter extraction. For example, the results of a parametric WCET analysis, as presented by Bygde [5], could be used to identify function parameters which impact WCET. We will develop additional heuristics to prevent cases where predictability decreases. More specifically, we plan to implement feedback-oriented source-to-source transformations and use feedback from these metrics to selectively apply transformations. To achieve a complete preservation of flow facts from source to binary level, we will investigate the possibility to use matching approaches similar to Becker et al. [4] or previous work by Stettlmann and Bringmann et al. [29]. Finally, we will extend scenario support, such that scenarios may capture information that go beyond simple parameter-value maps. For example, scenarios may be used to improve predictability of embedded software that makes use of dynamic polymorphism, such as virtual inheritance in C++.

REFERENCES

- [1] ARM Software. 2019. CMSIS-NN CIFAR10 Example. Retrieved November 10, 2019 from <https://github.com/ARM-software/ML-examples/tree/master/cmsisnn-cifar10>.
- [2] G. Barany and A. Prantl. 2010. Source-level support for timing analysis. In *Leveraging Applications of Formal Methods, Verification, and Validation*. T. Margaria and B. Steffen (Eds.), Lecture Notes in Computer Science, Vol. 6416, Springer, Berlin, 434–448.
- [3] M. Becker and S. Chakraborty. 2018. Optimizing worst-case execution times using mainstream compilers. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. ACM, New York, NY. 10–13. DOI: <https://doi.org/10.1145/3207719.3207739>
- [4] M. Becker, M. Pazaj, and S. Chakraborty. 2019. WCET analysis meets virtual prototyping: Improving source-level timing annotations. In *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*. ACM, New York, NY. 13–22. DOI: <https://doi.org/10.1145/3323439.3323978>
- [5] S. Bygde. 2010. *Static WCET Analysis Based on Abstract Interpretation and Counting of Elements*. Licentiate Theses No. 115. Mälardalen University Press, Västerås, Sweden.
- [6] Cooper, K. D., Hall, M. W., and Kennedy, K. 1992. Procedure cloning. In *Proceedings of the 1992 International Conference on Computer Languages*.
- [7] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. 2016. TACLeBench: A benchmark collection to support worst-case execution time research. In *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis*.
- [8] H. Falk, P. Lokuciejewski, and H. Theiling. 2006. Design of a WCET-aware c compiler. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis*. 121–126.
- [9] S. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakakis, T. Basten, L. Eeckhout, H. Corporaal, F. Cathoor, F. Vandeputte, and K. De Bosschere. 2009. System-scenario-based design of dynamic embedded systems. *ACM Transactions on Design Automation of Electronic Systems* 14, 1 (2009), 1–45.
- [10] S. V. Gheorghita, S. Stuijk, T. Basten, and H. Corporaal. 2005. Automatic scenario detection for improved WCET estimation. In *Proceedings of the 42nd Annual Design Automation Conference*. 101–104. DOI: <https://doi.org/10.1145/1065579.1065610>
- [11] AbsInt Angewandte Informatik GmbH. 2020. The Industry Standard for Static Timing Analysis. Retrieved July 02, 2020 from <https://www.absint.com/ait/>.
- [12] AbsInt Angewandte Informatik GmbH. 2021. aiT Worst-Case Execution Time Analyzers: Workflow. Retrieved January 05, 2021 from <https://www.absint.com/ait/features.htm>.
- [13] J. Gustafsson. 2013. WCET Project / Benchmarks. Retrieved January 05, 2021 from <http://www.mrtc.mdh.se/projects/wcet/benchmarks.html>.
- [14] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. 2010. The Mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis*. B. Lisper (Ed.), OpenAccess Series in Informatics, Vol. 15, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 136–146. DOI: <https://doi.org/10.4230/OASICS.WCET.2010.136>
- [15] M. R. Guthaus, J. S. Ringenberg, T. M. Austin, T. Mudge, and R. B. Brown. 2002. MiBench Version 1. Retrieved January 07, 2021 from <http://vhosts.eecs.umich.edu/mibench/index.html>.
- [16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization*. IEEE, 3–14. DOI: <https://doi.org/10.1109/WWC.2001.990739>
- [17] RWTH Aachen ICE. 1996. Institute for Communication Technologies and Embedded Systems: DSPStone. Retrieved January 07, 2021 from <https://www.ice.rwth-aachen.de/research/tools-projects/closed-projects/dspstone/>.
- [18] M. M. Kafshdooz, M. Taram, S. Assadi, and A. Ejlali. 2016. A compile-time optimization method for WCET reduction in real-time embedded systems through block formation. *ACM Transactions on Architecture and Code Optimization* 12, 4 (2016), 1–25.
- [19] R. Kirner. 2003. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. Ph.D. Dissertation. Technische Universität Wien, Vienna.
- [20] L. Lai, N. Suda, and V. Chandra. 2018. CMSIS-NN: Efficient neural network kernels for arm cortex-m CPUs. arXiv:1801.06601. Retrieved from <https://arxiv.org/abs/1801.06601>.
- [21] P. Lokuciejewski, H. Falk, P. Marwedel, and H. Theiling. 2008. WCET-driven, code-size critical procedure cloning. In *Proceedings of the 11th International Workshop on Software & Compilers for Embedded Systems*.
- [22] P. Lokuciejewski, F. Gedikli, P. Marwedel, and K. Morik. 2009. Automatic WCET reduction by machine learning based heuristics for function inlining. In *Proceedings of the 3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation*.

- [23] A. Loquercio, A. I. Maqueda, C. R. del Blanco, and D. Scaramuzza. 2018. DroNet: Learning to fly by driving. *IEEE Robotics and Automation Letters* 3, 2 (2018), 1088–1095. DOI : <https://doi.org/10.1109/LRA.2018.2795643>
- [24] V. Nirkhe and W. Pugh. 1993. A partial evaluator for the Maruti hard real-time system. *Real-Time Systems* 5, 1 (1993), 13–30. DOI : <https://doi.org/10.1007/BF01088695>
- [25] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini. 2019. A 64mW DNN-based Visual Navigation Engine for Autonomous Nano-Drones. arXiv:1805.01831v1. Retrieved from <https://arxiv.org/abs/1805.01831v1>.
- [26] C. Y. Park. 1993. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems* 5, 1 (1993), 31–62. DOI : <https://doi.org/10.1007/BF01088696>
- [27] P. Puschner, R. Kirner, B. Huber, and D. Prokesch. 2012. Compiling for time predictability. In *Computer Safety, Reliability, and Security*. F. Ortmeier and P. Daniel (Eds.), Lecture Notes in Computer Science, Vol. 7613, Springer, Berlin.
- [28] D. Saguil and A. Azim. 2020. A layer-partitioning approach for faster execution of neural network-based embedded applications in edge networks. *IEEE Access* 8 (2020), 59456–59469. DOI : <https://doi.org/10.1109/ACCESS.2020.2981411>
- [29] S. Stattelmann, O. Bringmann, and W. Rosenstiel. 2011. Dominator homomorphism based code matching for source-level simulation of embedded software. In *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 305. DOI : <https://doi.org/10.1145/2039370.2039417>
- [30] The Clang Team. 2020. LibTooling. Retrieved July 02, 2020 from <https://clang.llvm.org/docs/LibTooling.html>.
- [31] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. 2006. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proceedings of the 4th ACM/IEEE International Conference on Formal Methods and Models for Co-Design*.
- [32] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. 2007. POET: Parameterized optimizations for empirical tuning. In *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–8. DOI : <https://doi.org/10.1109/IPDPS.2007.370637>
- [33] V. Zivojnovic, J. Martinez, C. Schläger, and H. Meyr. 1994. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Applications & Technology*.

Received July 2020; revised May 2021; accepted June 2021