Annotate Once – Analyze Anywhere: Context-Aware WCET Analysis by User-Defined Abstractions

Simon Schuster
Peter Wägemann
schuster@cs.fau.de
waegemann@cs.fau.de
Friedrich-Alexander University
Erlangen-Nürnberg (FAU)
Erlangen, Germany

Peter Ulbrich
Peter.Ulbrich@tu-dortmund.de
Technische Universität Dortmund
Dortmund, Germany

Wolfgang Schröder-Preikschat
wosch@cs.fau.de
Friedrich-Alexander University
Erlangen-Nürnberg (FAU)
Erlangen, Germany

Abstract

The widespread adoption of cyber-physical systems in the safety-critical (hard real-time) domain is accompanied by a rising degree of code-reuse up to actual software product lines spanning different hardware platforms. Nevertheless, the dominant tools for static worst-case execution-time (WCET) analysis operate on individual, specific system instances at the binary level, further depending on machine-code–level annotations for precise analysis. Thus, this timing verification is neither portable nor reusable.

PragMetis addresses this schism by providing an expressive source-level annotation language that enables to express context dependence at the library level using *user-defined abstractions*. These abstractions allow users to generically annotate context-dependent flow facts down to the granularity of individual loop contexts. We then use control-flow–relation graphs to transfer these facts to machine-code level for specific instances, even in the presence of certain compiler optimizations, thus achieving portability. Our evaluation results based on TACLeBench confirm that *PragMetis*'s powerful expressions yield more accurate WCET bounds.

CCS Concepts: • **Computer systems organization → Real-time system specification**; **Real-time languages**; **Embedded software**; • **Software and its engineering →** *Compilers*.

Keywords: WCET analysis, annotation language, context sensitivity, control-flow–relation graphs

1 Introduction

In safety-critical real-time settings, static worst-case execution-time (WCET) analysis is the state-of-the-art approach to the verification of temporal properties [1, 59]. In general, WCET analysis is a challenging task as it must be easy to use while providing sound yet tight bounds on the WCET. One of the challenges is the micro-architecture machine-code analysis, which is highly dependent on the current hardware state (e. g., cache, pipeline). Consequently, established WCET tools [2, 5] operate at the binary level to infer these platform-specific runtime costs effectively. Another vital challenge results from the path analysis, which is used to characterize worst-case execution paths. These tools reconstruct this knowledge (i. e., flow facts) from the binary representation by control-flow analyses and abstract interpretation. In particular, path analyses reveal the actual program's behavior, for example, by inferring loop bounds and call contexts.

While automated analyses work well in specific domains, such as aerospace [6], they put high demands on the development process: a completely static system structure, in which the analysis can infer all relevant knowledge from the machine-code level. However, in general, this is not the case in many industrial settings requiring manual annotation [13, 23, 27, 57] to bound and refine analyses. Here, standard tools require annotations at the machine-code level to circumvent the traceability issue induced by compiler optimizations. However, binary-level annotations are known to be error-prone, labor-intensive [26, 51] and are subject to collateral evolution [48].

So, there are ample motivations for an expressive annotation language at the source-code level. Our paper's motivating example and a deputy is *reuse*, which we believe is gaining more importance with the increasing proliferation of hard real-time systems in cost-sensitive domains (i. e., automotive). A prime example of this is the use of libraries that offer functions for a wide range of application scenarios and cannot be easily annotated in a generalized way. Besides generic software components, general-purpose operating systems are increasingly being used, which are no longer tailored to a specific application. Therefore, extensive configurability is inherent for applications (e. g., product lines) and their operating systems [20]. In this context, existing annotation approaches prove insufficient [34, 44], especially for the annotation of system functions and operating system code [8, 15, 46, 52].

1.1 Problem Statement

This paper's objective is the generic annotation of software modules at the source level, which is module-centric (i. e., configuration or hardware agnostic), customizable, and preserves its validity in the presence of compiler optimizations.

Our goal is to manually annotate the code once with annotations that best match the program semantics. Subsequently, these annotations become reusable in arbitrary contexts. The fundamental challenge comprises two key problems:

Problem # 1: Necessity for Machine-Independent Annotations. Existing WCET tools have to compromise on automated analyses and compiler optimizations. Consequently, they typically demand annotations of the machine code, resulting in poor maintainability and limited expressiveness. Depending on whether the abstract interpretation can trace a variable (i. e., loop state) decides whether developers must bother with its concrete binary representation (i. e., register storage). These limitations make binary-level annotations unsuitable for reuse in different deployment scenarios.

Our solution provides expressive source-level annotations that draw upon a detailed expansion of execution and call contexts. Likewise, they provide full traceability at the binary level. For one, we achieve this goal by an adequate aggregation of high-level contexts and execution sequences to occurrence frequencies and numerical flow facts. In a second step, we leverage control-flow–relation graphs (CFRGs) [30] to lower these facts, even under certain code optimizations.

Problem # 2: Reusable Annotation of Complex Inter-Component Path Dependencies. A loop constraint in a library function may depend on the call context or global state (e. g., OS). Vice versa, this function can, for example, also be associated with side effects on the global state. Moreover, such dependency patterns can vary greatly, for example, with the overall configuration. Annotations must, therefore, handle complex and potentially inter-component dependencies. Generic abstraction domains (e. g., collection or interval semantics) of standard WCET tools are unable to manage these complex interrelationships in a meaningful way.

We propose parametric annotations with user-defined abstractions that allow for case-specific and highly expressive annotations to tackle complex semantics (e. g., complex data types) and interdependencies to solve this problem. These use the call parameters in a given context, incorporate configuration, and, most importantly, allow analysis under partial knowledge. In combination, they enable a bidirectional (i. e., caller/callee, higher-order or global state) exchange of knowledge and thus a composable analysis of configurable systems.

1.2 Contribution & Outline

This paper makes the following contributions: (1) *PragMetis*, a code-level annotation language that introduces *user-defined abstractions* for the generic annotation of complex flow facts in reusable code (e. g., libraries, core functions, OS). (2) The concept of *opportunistic annotations* that allow for a selective yet tolerable indefiniteness within the analysis context. (3) A context deduplication approach for both call-context and integer linear program (ILP) level to minimize analysis overheads. (4) An open-source WCET analysis framework, based on *Platin* [49] and CFRGs [30], which ensure a sound lowering from the code to the binary-level even in the presence of certain compiler optimizations. (5) An evaluation based on our prototype and TACLeBench [18], a widely used benchmark suite.

Lisper et al. argue for better annotation languages to provide precise information [44], while Kirner et al. compared the strengths and weaknesses of annotation languages regarding their feature set [33]. Based on this comparison, *PragMetis* is the first annotation language for WCET analysis with optimization awareness that comprises all features of addressing individual loop, call, and application contexts.

The remainder is organized as follows. Our approach to context-aware analysis by user-defined abstractions is presented in Section 2, with Section 3 devoted to the implementation of *PragMetis*'s toolchain. Section 4 provides a case study based on TACLeBench and experimental results. Section 5 discusses future work and Section 6 related work. Finally, we conclude our work in Section 7.

2 Approach

In this section, we detail *PragMetis*, a source-code annotation language for expressing scoped, parametric control-flow refinement information that solves the challenges posed in Section 1.1. Deviating from that order, we begin with the design of the annotation language in Section 2.2. Here we cover *PragMetis*'s novel concepts of user-defined abstractions and opportunistic annotations to facilitate annotation locality. Building on these foundations, we detail the annotations' traceable lowering to machine-code in Section 2.3. Finally, we illustrate the ILP formulation for the worst-case bound in Section 2.4.

2.1 System Model

However, before diving into our approach, we briefly discuss our system model. *PragMetis* provides reusable, context-sensitive, source-code–level annotations for static WCET analysis. As these annotations allow to refine the control-flow information within a (single-threaded) programs' (high-level) path analysis, the approach is not bound to specific hardware properties. Therefore, *PragMetis* only assumes general timing predictability, in particular the absence of timing anomalies [14, 21, 45] from the hardware.

2.2 Annotation Language Design

Reuse implies that code is executed under various constraints and in different contexts. Consequently, we enable the developer to express these differing contexts utilizing *PragMetis*'s expressive annotations. As we build upon the T-Crest toolchain [53], *PragMetis* reuses its **`#pragma`** directive to embed annotations at the source-code level as well as the syntax of Platina [55] for annotation of control-flow properties (e. g., loop bounds, path feasibility, or indirect calls). The given original syntax is:

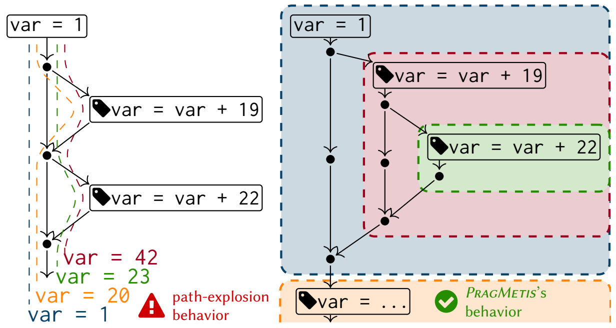

Figure 1. Example of the risk of path explosion with non-scoped visibility (left), with each annotation carrying the risk of doubling state space, as opposed to *PragMetis*'s (right).

```
#pragma platina (lbound|guard|callee) "<expression>"
```

In [55], expressions can reference and combine the values of global, symbolic parameters to specialize the analysis to a specific system state. With *PragMetis*, we extended this language and its syntax in two ways: (1) A new annotation `platina let` that allows changing symbolic parameter values used in expressions *locally*. Such a local refinement of symbolic parameters enables modeling the evolution of the execution context within the annotation language at a fine scale. (2) The ability to distinguish between different loop contexts within expressions by managed indexing variables. A key concept of *PragMetis* is defining dominance regions for such bindings and evaluating them in a purely functional manner. Therefore, we leverage scope graphs [17, 56] and extend them to derive said validity regions in the following.

2.2.1 Dominance-Based Scoping Model. Here, Figure 1 exemplifies the need for a purposeful scoping of analysis contexts. On the left, a simple (imperative-style) scoping and visibility concept is shown to annotate the *parameter* `var`. Note that parameter `var` is a symbolic parameter within the annotation language, which may have a direct relationship to an individual variable in the program, but may as well express conceptual properties, such as operating modes. Given an initial value, the subsequent annotations modify the value of `var` *globally*. Unfortunately, using `var` as a variable within annotation expressions requires a full explicit path enumeration; its observable values depend on the execution history. This explicit enumeration contrasts with established techniques, such as the implicit path enumeration technique (IPET) [41], that try hard to keep the number of examined paths low to keep the analysis problem in a manageable size.

Contrasting, *PragMetis* features a visibility concept based on dominance, as shown on the right in Figure 1. Each binding is only visible within its dominance region. That is the part of the control-flow graph that can only be accessed through the annotated program point. For example, a function entry dominates the function members, and a loop header dominates its body. We leverage scope graphs [17, 56] to express the hierarchical deconstruction of a program's contexts into scopes. In *PragMetis*, we use scope graphs at the source-code level and introduced new scopes for our annotation expressions such as `let`. Conveniently, dominance regions regularly (i. e., **`goto`**s are a manageable exception) map to blocks at the source-code level. The resulting *model graph* allows for efficient identification and specialization of the identified program contexts during analysis; we detail this generation process in Section 3. Avoiding the naive approach's path explosion, *PragMetis* facilitates a hierarchical top-down expansion of analysis contexts. While this scoping keeps the analysis state small, it impedes automatic differentiation of individual paths as part of the analysis. Therefore, *PragMetis* provides *user-defined abstractions* to express and distinguish combinations of symbolic parameters for the various paths locally, within the individual annotation expressions. In this way, we maintain meaningful expressiveness despite the state reduction. We detail this mechanism in the following section.

2.2.2 User-Defined Abstractions. In comparison with established annotations fueled by abstract-interpretation–based analyzers, our dominance-based approach requires a fundamentally different, much more declarative annotation design. Returning to our example from Figure 1 (right), the last annotation (`var = ...`) must summarize the preceding constraints and thus *abstract* from the different values of `var` at this program point. The type of abstraction is dictated by the program structure and semantics and should keep the analysis precise. The following example deepens this aspect on the example of three specifications for a loop bound that depends on the abstraction of `var`:

```
lbound "var" // ①
lbound "42 - var" // ②
lbound "if var == 16 then 1337 else 42" // ③
```

For sound yet precise worst-case approximation, each annotation calls for a different abstraction over `var`: ① requires the maximum value and ② the minimum. On the other hand, for ③ neither of these abstractions is helpful as they cannot rule out the infeasible case of `var == 16`. Therefore, a set-based abstraction of potential values is required. Fully automatic frameworks [3, 29, 42, 58] that are based on abstract interpretation attempt to track program values in a limited collection of abstract domains. Each domain models values as long as it seems feasible and then combines this knowledge to the most favorable result. While this automatic domain tracking technique is powerful, it still has its limits as soon as more complex data structures or application-specific knowledge is involved and is thus unsuited for our aims. Therefore, we lift the creation and selection of the most suitable abstraction to the annotation-language level and the developer, respectively. For our example, this implies: ① and ② are common range types, for which *PragMetis* provides a built-in interval arithmetic abstraction. In case of ③, *PragMetis* offers sets and the ability to query set membership.

In addition to these simple examples, *PragMetis* facilitates user-defined, complex data types forming records, lists, or sets thereof. Our annotation language obtains these capabilities from Dhall [22], along with its variation of CCω and type system [12]. To better adapt to the domain of static WCET analysis, we extend Dhall's core language [22] by custom types such as numeric ranges and lists of C functions to carry callee information, an extended standard lib, as well syntax extensions like the indexing variables or `undefined` values. Ultimately, this annotation language allows for *user-defined abstractions* that model even complex properties, such as approximations on library data structures or OS semantics.

2.2.3 Opportunistic Annotations. While user-defined abstractions are a flexible and powerful tool, it puts high obligations on developers. The more specific the abstraction, the more information is potentially required at the call site, which might not be available in all contexts. In turn, the analysis so far fails, although in many cases, a less accurate but more straightforward abstraction with fewer requirements may exist. Therefore, we introduce another complementary mechanism in *PragMetis*, *opportunistic annotation* through *aggregation and filtering*. This enables the same syntactic construct to be applied multiple times in varying specifications, for example:

```
for(i = 0; i < 42 - nelems; i++) {
  #pragma platina lbound "42 - nelems"
  #pragma platina lbound "42"
  // ...
}
```

Consequently, *PragMetis aggregates* all available flow facts and chooses the best specialization (e. g., lowest loop bound, most restrictive callee set, or maximum path feasibility restriction). Thus, developers are relieved from case-specific constraints enabling a the-more-the-merrier, *opportunistic* annotation approach. Furthermore, *PragMetis* allows values to be marked as `undefined` at certain program points:

```
#pragma platina let "nelems = (undefined : Integer)"
```

An `undefined` value propagates until it is resolved in another context, or the affected expressions are ultimately removed from the analysis. However, if an immediate flow fact is computed, *PragMetis filters* the corresponding annotations during the flow-fact aggregation. That way, different abstractions coexist, and annotations become truly *opportunistic*, as aggregation compensates for the lack of local knowledge.

2.2.4 Loop-Context–Indexing Variables. Loop-context dependence represents a typical annotation problem. For example, whether a path is infeasible in every second loop iteration. Therefore, to reference such loop contexts, *PragMetis* introduces a set of predefined *loop-iteration identifiers*:

```
for (i = 4; i < 8; i++) { // ④
  #pragma platina lbound "4"
  for (j = 0; j < i; j++) { // ⑤
    #pragma platina lbound "$1 + 4"
    // ... // $0 ⇔ ⑤; $1 ⇔ ④
  }
  if (i % 2) { // $0 ⇔ ④
    #pragma platina guard "Natural/odd ($0 + 4)"
    // ...
  }
}
```

These *loop-context–indexing variables* (abb. indexing variables) have the syntax `$<numeric index>`, where the index refers to the level within the loop-nesting hierarchy starting with `$0` for the innermost loop scope. Semantically, each indexed variable ranges from 0 to (`lbound - 1`). Consequently, *PragMetis* can not only accurately express the above triangle loop, which has a closed-loop formulation in terms of scalar-evolution expressions [4], but also more complicated cases where no closed-loop or polyhedral formulations exist—for example, loop iterations with conditional behavior in the listing above. While boosting analysis precision, indexing variables pose the risk of a significant increase in call contexts: A 1000 iterations loop body calling a function would spawn up to 1000 call contexts, each with its binding for `$0`. While our context-pruning logic (see Section 3) can help keeping such numbers feasible, there is a simpler, constructive solution: Indexing variables are not automatically visible across the function-call boundary. However, if needed, these can be bound to a regular scoped variable at any time.

2.2.5 Locality. In our annotation language, all flow-fact–inducing annotations (i. e., `lbound`, `guard`, and `callee`) provide locality as they are placed directly at the program point they describe, which fosters usability as well as maintainability. However, due to our scoped visibility, this is not the case for some of our `let`-based bindings, especially those describing the value of an abstraction after function calls:

```
int var = pow(x, 2);
#pragma platina let "var = x * x"
```

Instead, in this example, we must annotate `pow`'s behavior at the call site, a somewhat unrelated program point. Consequently, there is a hard-to-maintain collateral evolution between `pow` and its annotations at all call sites. Furthermore, the caller should not be required to describe the behavior of the function called. We adopted the concept of postconditions from contract-based programming to resolve this issue, introducing the new annotation type, `glet`. Thereby it is possible to define reusable, globally valid bindings from within the source-code restoring locality.

```
// Within pow()
#pragma platina glet "                                    \
 pow_post = \(x: Natural) -> \(exp: Natural) ->           \
  List/fold Natural (List/replicate exp Natural x)        \
  Natural (\(x: Natural) -> \(y: Natural) -> x * y) 1"

// After the call at the callsite
#pragma platina let "var = pow_post x 2"
```

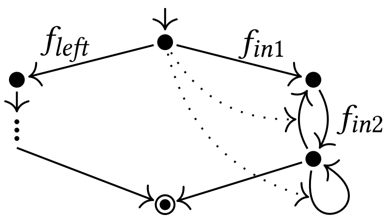

Figure 2. *PragMetis* aggregates all loop contexts within a function context. Constraints linking the back-edge frequencies (dotted) to the function entry allow the solver to assume iterations correctly.

Postconditions can call each other as long as a partial order exists within the global annotations' callgraph.

Summarizing, *scope-based context refinement*, expressive *user-defined abstractions*, *opportunistic annotations*, and *indexing variables* provide the foundations of our annotation language. Thereupon, commonly used idioms (e. g., specifying symbolic bindings for function parameters) can be added to foster usability.

2.3 Lowering under Compiler Optimizations

The second part of this section is devoted to our first problem (cf. Section 1.1): traceable lowering of flow facts to the machine program level. The challenge is relaying source-level expressions at specific program points to matching flow facts on the machine-code level. This is not a new problem, and several approaches exist [19, 30, 35, 39, 54] to address the issue. Apart from suppressing control-flow altering optimizations [54], several techniques exist that co-transform flow facts along with individual optimizations during the compilation process, either using a tailored compiler [19] or by instrumenting optimization passes within mainstream compilers [35, 39]. In contrast, CFRGs [30] represent a generic mechanism to relate execution flows at different representation levels of the program, even across semantic preserving program transformations. As CFRGs can be constructed from a partial mapping of individual program points at the different levels, as it is often available from standard compilers, the technique requires fewer modifications than the co-transformation–based approaches. The flow relations take the form of regular path relations and mathematically capture equivalencies of execution flows between joint points of execution progress. This representation holds enough information to transform numeric flow-frequency information, as it is commonly used within IPET-based static WCET analyses, to the machine-code level. While these regular relations are less expressive than manual co-transformation and the technique focuses on backend optimizations, CFRGs still support several high-level optimizations such as loop peeling, loop unswitching, or loop interchange [30]. Along with the availability of an open-source implementation [25] within a standard compiler [37] and the higher portability, this technique provides a suitable basis for *PragMetis*.

The first challenge is in flow facts for program points that occur in multiple analysis contexts, caused by a function called in either different call or loop contexts. As all primitive annotations (`guard`, `lbound`, `callee`) are locally contained to the respective function, we use specializations at the function level as the basic unit for this transformation. The approach described in [55] already facilitates lowering of program-global `guard`, `lbound` and `callee` annotations. There, a `guard` annotation at the source level is defined as a constraint to the frequency variable $f_{\texttt{guard}}$ of the corresponding program point: $f_{\texttt{guard}} \leq 0$

Such global numeric flow information is transferred by CFRGs in the presence of compiler optimizations, as long as the flow relation is valid and a progress node (or any pre/post-dominated node) exists that links $f_{\texttt{guard}}$ to one or more frequency variables at the machine-code level. This logic is easily extended to any local annotation outside of a loop context. A primitive value represents the annotation's value: a boolean indicating feasibility (`guard`) or an integer denoting execution frequency of the back edge (`lbound`). To localize these constraints, we scope those values by relating them to the frequency of the function entry:

$$f_{\texttt{guard}} \leq 0 * f_{\texttt{function}}$$
$$f_{\text{back edge}} \leq lbound * f_{\texttt{function}}$$

Annotations contained in loop contexts are a little trickier: While it is straightforward to unroll a loop at the source-code level, a CFRG is incapable of indicating the corresponding locations on the binary level for individual iterations' program points. Instead, it provides sets of frequency variables at both levels, whose execution frequency matches according to constraints spanned by joint progress nodes. However, with loop-restructuring optimizations (e. g., loop interchange), the loop contexts' execution sequences might differ among the two levels. Consequently, *PragMetis* addresses all loop contexts in a function's context collectively instead of individually by aggregating the annotation's results into a specific numeric flow fact at the source-code level that a CFRG can lower. For inner loops, this aggregation is defined by the sum of all loop bounds l_i observed for the individual outer loop's contexts, or more formally:

$$f_{\texttt{inner back edge}} \leq f_{\texttt{function}} \cdot \sum_{i \in \text{outer loop iterations}} l_i$$

However, uplifting loop-body frequencies to the function scope still requires further constraints. Considering the example given in Figure 2, the costliest path can either be the left (f_{left}) or the right (f_{in1}, f_{in2}), depending on the context-dependent loop bounds. Yet, by aggregating the back-edge frequencies to the function entry (dotted arrows), there is no indication of any incoming flow into the respective loops. Thus, an ILP solver might mistakenly assume loop iterations up to the maximal back-edge frequency. We, therefore, bound the global back-edge frequency by the maximum local loop

Loop context	$l1$	$l2$	$l3$
Callee sets	$\{CC_f, CC_g\}$	$\{CC_g, CC_h\}$	$\{CC_i\}$
Binary choices	$0 \leq f_{l1} \leq f_{\texttt{function}}$ $0 \leq g_{l1} \leq f_{\texttt{function}}$	$0 \leq g_{l2} \leq f_{\texttt{function}}$ $0 \leq h_{l2} \leq f_{\texttt{function}}$	$0 \leq i_{l3} \leq f_{\texttt{function}}$
Mutual exclusion	$f_{l1} + g_{l1} \leq f_{\texttt{function}}$	$g_{l2} + h_{l2} \leq f_{\texttt{function}}$	$i_{l3} \leq f_{\texttt{function}}$
Call edge frequencies	$call_f \leq f_{l1}$	$call_g \leq g_{l1} + g_{l2}$ $call_i \leq i_{l3}$	$call_h \leq h_{l2}$
Callsite constraint	$call_f + call_g + call_h + call_i \leq f_{\texttt{callsite}}$		

Figure 3. ILP formulation to express mutual exclusion in the presence of ambiguous callee contexts within loops

bound l_i in the various outer loop contexts (if any):

$$f_{\text{inner back edge}} \leq f_{\text{in}} \cdot \max_{i \in \text{outer loop iterations}} l_i$$

Global and local constraints complement each other. The former provides structural constraints, ensuring that the loop is only considered at incoming flow. The latter ensures tight back-edge frequencies.

For `guard` annotations, the aggregation yields: "How often is the program point `pp` *feasible*?" Based on the set of loop-iteration instances, *PragMetis* determines how often the guard expression's values g_i are feasible or undefined within the different loop contexts:

$$f_{\texttt{pp}} \leq f_{\texttt{function}} \cdot |\{i \in \text{loop contexts} | g_i \in \{\text{undef., feasible}\}\}|$$

Both aggregations do not require a closed formulation (e. g., Gauss sums for triangular loops) of a loop's execution frequencies: While closed forms can speed up the aggregation of flow-frequency constraints, we otherwise resort to a full symbolic evaluation of all individual loop contexts.

Another hurdle is the aggregation of outgoing calls due to the more complex information: evaluating a call site results in a set of (potential) outgoing mutually exclusive call contexts, each represented as a tuple (model context, function). In an ILP-based WCET analysis, the decision which of these contexts constitutes the actual worst case is the numeric solver's responsibility. Consequently, we must encode the mutual exclusion as ILP constraints, which is best illustrated by an example. Figure 3 depicts the formulation of an annotated call site within a loop of bound three. The analysis derived three different sets of call contexts for the three loop contexts $l_{1...3}$ based on the evaluated callee annotations. In this example the callee in l_1 and l_2 is ambiguous. To express this ambiguity, we introduce a selection-state variable $\in \{0, 1\}$ that represents whether a specific callee, and thus outgoing call context, was chosen and restrict the global frequency to one contained context per set. As this decision happens once per function invocation, this global frequency is thus limited by the functions entry node's frequency, $f_{\texttt{function}}$. The sum of each set's selection-state variables then governs this particular call edge's overall execution frequency. *PragMetis* uses a call-tagging mechanism such as [55] to attribute call edges to call sites at the machine-code level. However, this complex encoding is only necessary if any set is ambiguous. Otherwise, a much simpler encoding is sufficient that limits each call edge by the overall frequency of a particular call context among the callee sets.

This call-edge mapping scheme assumes that the call itself persists at the machine-code level, which disallows inlining optimizations. In practice, however, this is not an issue. As all of our annotation expressions reside at the source-code level, all inlining operations can be performed before constructing the CFRG without loss of generality.

2.4 IPET-Based ILP Formulation

We showed how to evaluate the call graph into a graph of context-specific function-specializations and infer flow facts by our annotation expressions from Section 2.2. Further, we detailed their lowering to the machine-code level by CFRGs in Section 2.3. However, we still miss the IPET's final analysis step: the ILP formulation.

There, *PragMetis* employs straight duplication: that is, a unique instance-id identifies each function specialization. For the latter, all variables are tagged with the respective instance-id when emitting both structural constraints and flow facts. A final adaption to this process is required for global, non-*PragMetis* flow facts (e. g., global constraints), which typically come with multiple, tagged copies of the program points they refer to. We address this issue by introducing global representatives for those program points as sums of the associated program points' execution frequencies within the individual copies. While this automatic mapping works reliably, specifying global flow-facts at the ILP level, outside of *PragMetis*, carries the risk of re-introducing unnecessary pessimism. As *PragMetis* aggregates loop-context–sensitive flow information (e. g., conditional infeasibility) across all loop iterations determined by *PragMetis*'s annotations, a later tightening of the loop bound at ILP level will not be reflected within *PragMetis*'s aggregated flow facts; actually infeasible iterations might thus introduce pessimism into the analysis. To circumvent this issue, as much information as possible should be expressed as *PragMetis* annotations.

In sum, *PragMetis* provides a usable, generic, expressive, source-level annotation language for IPET-based WCET analyses, even in the presence of certain compiler optimizations.

3 Implementation

In this section, we dive into *PragMetis*'s efficient implementation, which we based on the T-Crest toolchain [53] and provide as open-source[1]. *PragMetis*'s toolchain consists of a clang/LLVM-based compiler [37] to generate the CFRGs [30] and the *Platin* WCET analyzer [25, 49, 55], where most of the contextual analysis is implemented. Conceptually, CFRGs

[1]The implementation of *PragMetis* is publicly available under an open-source license: gitlab.cs.fau.de/pragmetis

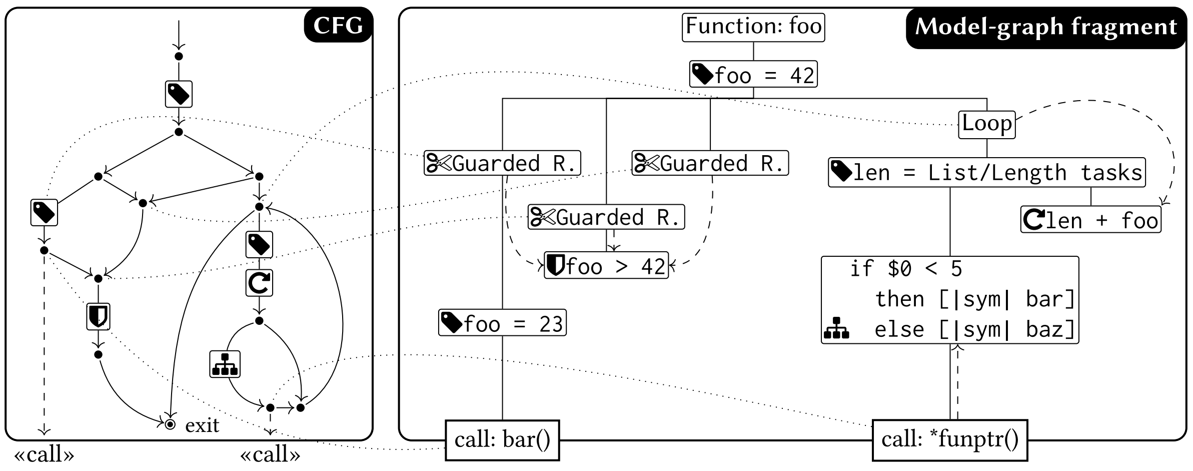

Figure 4. Model-graph fragments use dominance and abstract a function's annotation-enriched control flow into a scope-based tree representation for efficient evaluation. Every fragment node is linked to a CFG node (dotted, some edges omitted for legibility), which might either be control-flow structures or let annotations (🏷). Additionally, some nodes (guarded regions/✂, calls, loops) reference their associated annotation nodes (guard/🛡, callee/品, lbound/↻) that determine the node's behavior.

can correctly handle several high-level optimizations [30], such as loop peeling, loop unswitching, or loop interchange. In their actual implementation within the T-Crest toolchain, CFRGs link LLVM's bitcode and machine-code–level program representations [30] and thus only encompass LLVM's backend optimizations. However, the T-Crest toolchain still supports source-level annotations for absolute, numeric loop bounds, expressed as custom pragmas. To ensure the validity of those annotations over high-level, loop-restructuring optimizations that might occur outside of the constructed CFRG, the toolchain locally suppresses these optimizations for annotated loops [25]. As *PragMetis*'s prototypical implementation reuses this mechanism, it shares this behavior. Thus, *PragMetis*'s modifications to T-Crest's LLVM-based compiler are limited to parsing of *PragMetis*'s custom pragmas, and their extraction at the CFRG level.

The contextual analysis is done as a pre-processing step in *Platin*'s timing analysis. Here, the set of individual call contexts (i. e., function and its *model context*) is determined from the initial context and the analysis's entry function. As our optimization-aware lowering operates on functions, call contexts constitute the atomic unit of processing. Consequently, our analysis is performed on *model-graph fragments*, a condensed representation we derive from the source-level control-flow graph (CFG). Figure 4 gives an example. Based on dominance information, these mode-graph fragments cluster CFG nodes (function entries, loops, annotations, and calls) hierarchically by their area of effect into scopes. We further insert *guarded region* markers based on post-dominance information, as path infeasibility (and thus `guard` annotations) acts bidirectionally on the execution path.

The analysis process itself is mostly hierarchical, with `lets` only refining the model contexts locally, except loops and guarded regions. Here, a child node can retroactively influence the parents' loop bounds or feasibility. Therefore, loop scopes and guarded regions feature a set of *associated* annotations, which are evaluated depth-first and whose results are then aggregated before continuing. This helps to discard locally infeasible subtrees early in the evaluation process. For loops, *PragMetis* performs a loop-bound–sensitivity analysis on the loop body: if no references to `$0` are found, a single evaluation is conducted and results are weighted by the loop bound; therefore, we call this a *loop-context folding* optimization. Otherwise, all iterations within $[0; (lbound - 1)]$ are elaborated, symbolically evaluated, and aggregated. For calls, the analysis determines the set of outgoing call contexts, that is, callee set and model context passed to the called functions.

Once a model-graph fragment is completely evaluated, we aggregate and store all results that are required for lowering according to the scheme detailed in Section 2.3. That is, feasibility count for `guards`, observed back-edge frequencies for loops, and call contexts for outgoing calls. Subsequently, the analysis proceeds following the outgoing call contexts. This call-context–specific refinement continues hierarchically until the *model graph* is complete.

Our implementation is tuned to analyze each context only once to minimize overheads. Accordingly, *PragMetis* aims for deduplication of model contexts, which is done in two steps. First, it is performed based on the abstraction values within the model contexts. However, this approach is not exhaustive as differences between model contexts might concern properties not referenced by any annotation in the called function. Therefore, *PragMetis* uses a *second level* of deduplication before generating the ILP: When the model graph is enumerated, all instances are processed from leaves to the root in reverse topological order. For each instance, flow facts that include loop bounds, infeasible paths, and called function variants, are lowered to the machine-code level. These facts are subsequently reused to reliably deduplicate any two instances of a function that will behave the same way within the ILP. This deduplication reduces the number of ILP variables and thereby the complexity of the optimization problem considerably. Finally, the ILP is generated as described in Section 2.4.

Besides showing the feasibility of the *PragMetis* approach, we also exploit efficient data structures, detection of idempotent loop iterations, and the multi-level deduplication of identical contexts in order to improve *PragMetis*'s efficiency.

4 Evaluation

In this section, we present *PragMetis*'s evaluation. After detailing our experimental setup, we provide both a quantitative evaluation based on the widely used TACLeBench [18] benchmark suite and a qualitative discussion of expressiveness and usage of our annotations.

4.1 Experimental Setup

In this evaluation, we use our open-source prototype from Section 3. Measurements were conducted on an Intel Core i7-8550U using Ruby (2.7.2p137) and Gurobi (v9.1.1) as the ILP solver. We took the upstream of TACLeBench at commit `dcc2501` as our reference, to which we applied our annotations where beneficial. When measuring overall analysis times, we performed 11 measurements per configuration, dropped the first to mitigate filesystem-caching effects, and averaged the remaining measurements. All studied programs were compiled with hardfloat and analyzed for an ARM Cortex-M4 processor family using the Infineon XMC4500 platform with caches disabled. For the analysis results, we validated flow frequencies of annotated loops against execution counts for the reference input.

4.2 Quantitative Evaluation

The quantitative analysis was performed on TACLeBench's *kernel* category of benchmarks. These benchmarks provide a wide range of implementations of typical operations commonly found in embedded systems, from more general sorting mechanisms to checksum as well as matrix functions and filtering applications, all written in different implementation styles, and thus provide a good testbed for our source-based annotation approach. As *Platin*, and thus *PragMetis*, does not offer an explicit concept for analyzing recursion, only 24 of the 29 benchmarks, in their original form, are analyzable. TACLeBench already provides global, context-insensitive flow facts via annotations (especially for loop bounds) for its benchmarks. These flow facts were mostly generated by tracing and recoding the benchmarks' execution on their sample input and then deriving the bounds and thus are already input-specific. This means that for *PragMetis* to provide a tighter bound than those provided annotations, which we use as our baseline, during quantitative analysis, a benchmark has to exhibit internal context-dependence within itself, in terms of either function or loop contexts. This is the case for at least seven benchmarks. We refer to the remaining 17 benchmarks without improvements as *no effect* within our evaluation. Their analysis result, as well as the incurred runtime overhead, is shown in Table 1. There, the actual improvement of generated worst-case bounds varies considerably: *PragMetis* can only provide tighter bounds by eliminating pessimism that was introduced by overprotective globalization of flow facts of multiple contexts within the original reference annotations. The extent of this internal pessimism depends on the benchmark, ranging from a mere 3.08 % with the `minver` benchmark to up to a 99.36 % with the `fft` analysis, which exhibits a deeply nested, fluctuating loop structure that benefits greatly from a fine-grained differentiation between the individual loop iterations. Of course, this fine-grained differentiation within a highly repetitive, not folded loop incurs more work on the analysis, as it has to symbolically evaluate these annotations within those different contexts for aggregation. This analysis makes `fft` the second most expensive benchmark within the set, with an increase in analysis time of factor 2.73×. In absolute terms, the total analysis time for the seven benchmarks ranges from 0.5 s (`insertsort`) to 13.4 s (`prime`). Yet, cost and saving do not always correlate, as seen for the `prime` benchmark, where the analysis overhead is disproportionally high. Here, the analysis time increases by 24.46× and the pessimism is reduced by 9.31 % (to a bound of 90.69 %).

To understand this in more detail, it is beneficial to look into where execution time is spent during analysis across the different benchmarks. Figure 5 shows this relative distribution. Again, the exact distribution is benchmark-specific. However, for `fft`, the measurement shows that most of the time is spent on constructing the model graph and thus actually expanding those loop contexts; with approximately

Table 1. Analysis results for the annotated benchmarks, their runtime overhead relative to context-insensitive analysis, and the number of context-sensitive annotations; the *no effect* category represents benchmarks without improvement.

benchmark	*WCET bound (norm.)*	*analysis time (norm.)*	*# contextsensitive annotations*
no effect	100 %	1×	0
bsort	52.30 %	1.52×	3
fft	0.64 %	2.73×	8
insertsort	72.03 %	1.05×	1
ludcmp	38.00 %	1.20×	11
md5	32.83 %	1.29×	24
minver	96.92 %	1.30×	2
prime	90.69 %	24.46×	3

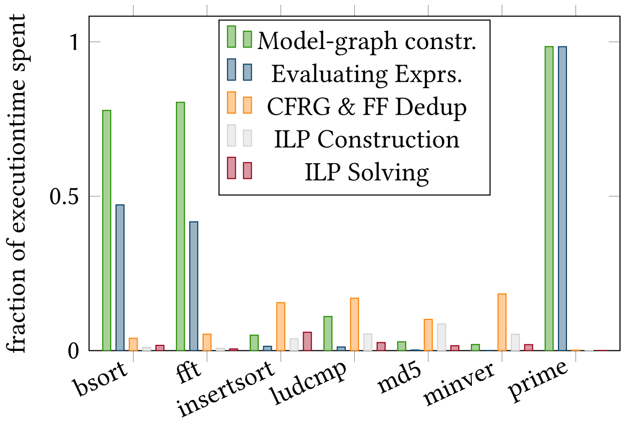

Figure 5. Runtime distribution of analysis phases. Expression evaluation is a part of the model-graph construction

half of that time in turn being spent on evaluating the annotation expressions themselves. Generally speaking, this is the case for most of the benchmarks that exhibit nested, loop-context–dependent loops, so `bsort` is part of this group as well. Among the remaining benchmarks, the number of loop contexts is a lot smaller, so other parts of the process, most prominently the CFRG-based transformation of the generated flow facts, dominate the process. Again, `prime` is the exception to the rule, and nearly all time there is spent on annotation-expression evaluation. The reason is that `prime` itself is quite short, but the annotation uses a rather complex and expensive to compute abstraction, which puts a lot of load on our tree-visiting annotation-language interpreter. The abstraction of `prime` is further discussed in the qualitative analysis (see Section 4.3).

As the final step in our quantitative analysis, we studied the efficiency of the different optimization techniques we introduced in Section 3. Table 2 shows the results. Again, the loop-heavy benchmarks (`bsort`, `fft`, `insertsort`) only have few call contexts and thus do not profit from the call-context deduplication. In the case of `fft`, *PragMetis*'s loop-context folding significantly reduces the analysis time. As both `bsort` and `insertsort` are dominated by a triangular

Table 2. Efficiency of optimized evaluation techniques: Deduplication reduces the number of call-contexts at model-graph and ILP level, loop-context folding reduces the number of loop contexts actually evaluated

benchmark	*# call contexts*	*# model-graph ctxs (rem. %)*	*# ILP-level ctxs (rem. %)*	*# loop contexts (eval. %)*
`bsort`	2	2 (100.00%)	2 (100.00%)	5247 (100%)
`fft`	2	2 (100.00%)	2 (100.00%)	43040 (35.79%)
`insertsort`	1	1 (100.00%)	1 (100.00%)	64 (100%)
`ludcmp`	7	3 (42.86%)	3 (42.86%)	130 (100%)
`md5`	58645	24 (0.04%)	23 (0.04%)	671 (2.09%)
`minver`	13	4 (30.77%)	4 (30.77%)	118 (31.36%)
`prime`	37	8 (21.62%)	6 (16.22%)	29 (6.90%)

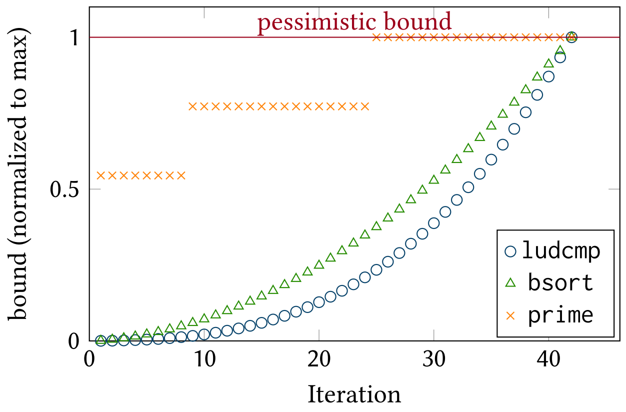

Figure 6. Parameter study in the range of 1 to 42

loop, our basic folding was not applicable. The other benchmarks make use of function calls more extensively and thus profit from the call-context deduplication a lot more, with between 57.14 % to up to 99.96 % of the call contexts deduplicated during elaboration at the source level. The duplication at the ILP level based on flow facts did not prove too effective within our measurements. We partly attribute this to the rather compact and specific benchmarks, and expect this number to rise within larger programs.

Finally, we did test our solution against different optimization levels (`-O1`, `-O2`), where all 7 annotated benchmarks remained analyzeable. These results confirm that our approach of aggregating fine-grained flow information at the function scope and lowering interacts gracefully with the implemented CFRGs.

4.3 Qualitative Evaluation

However, a purely quantitative analysis misses out on two important aspects of *PragMetis*: The ability to define custom abstractions and to reusably reanalyze a library in different contexts parametrically. Among the benchmarks studied, the most common abstractions used are maximum abstractions on input data, either in terms of numerical values or array lengths. But there are two interesting exceptions: `md5` passes a complex computation context throughout its functions, and `prime` does not require the maximum value, but the number of quadradic values below a certain maximal input value. Especially the latter shows the potential in user-defined abstractions at the annotation level: Such specific information is unlikely to be provided by any fully automatic analysis tool. Furthermore, this issue is invariably linked to the issue of maintainability: Loops in TACLeBench are tagged with numeric min/max bounds observed for an execution of the reference input. However, in contrast to our approach, there is no indication that, for instance, 25 was the number of square numbers below the max input value of 2759. Here *PragMetis* is able to provide an additional benefit: besides the context-specific, tailored bound, *PragMetis* is capable of deriving flow facts for a context-insensitive analysis as well. So the benchmark suite TACLeBench could be annotated more expressively, without comprising functionality, and thus, for instance, answer the question of a comment within upstream's dijkstra benchmark on the origin of the loop bound in a hard-to-understand loop-annotation statement.

The other aspect we studied was whether *PragMetis* can precisely express generic flow interdependencies and then analyze them in different contexts. Figure 6 provides insight into this by means of a parameter study. Here, we used three of the benchmarks and varied the value backing the abstraction governing their runtime bound, that is varying the maximum input size (in terms of array length or maximal numeric input value), during the WCET analysis. Here, *PragMetis* is capable of expressing not only more traditional loop formats such as within `bsort`, which exhibits the expected, approx. quadratic growth of a triangular loop, and possesses a closed formulation, but also far more irregular patterns, such as for the `ludcmp` benchmark, where a closed formulation does not exists at all. Finally, the result for `prime`, which follows the far more erratic execution time pattern outlined above, consists of a series of constant segments of non-uniform length, that step up in their WCET bound whenever the maximum input abstraction passes a new value in the series of square values of odd numbers larger than 3, which means $3^2 = 9$ and $5^2 = 25$ for our study's parameter range. As evident in the Figure 6, this context sensitivity provides significant potential for reducing analysis pessimism, which is the area between the individual measurement series and the global pessimistic bound. But, more importantly, as the study did not modify any annotations themselves, this shows that *PragMetis* can live up to its motto: Annotate (just) once, but analyze everywhere, no matter in what context.

5 Discussion & Future Work

PragMetis provides the ability to concisely annotate high-level contextual knowledge by the virtues of an expressive annotation language and thus allows the programmer to choose her own abstraction to annotate a benchmark as precisely as possible, with Section 4 showing its potential.

Yet, there still exist several potentially worthwhile avenues of further research. The first is integrating a concept for recursive applications into *PragMetis*. Even though safe coding standards such as MisraC [47] often explicitly forbid, there are still uses, as witnessed for instance by the 5 skipped benchmarks within TACLeBench. Here, the problem is more one of usability over semantic. Conceptually recursion represents a loop on the call-graph level and integrating and thus could be handled in a similar fashion to how we handle loop contexts. However, this possibly makes annotations more complicated to use than formulating the respective new context abstractions at the recursive call site.

Independent of *PragMetis*'s annotations, we observed that the T-Crest toolchain occasionally generated either invalid or too trivial CFRGs. Generally, such glitches can be manually fixed by modifying the corresponding code. Note that no such issues were observed in the benchmarks used in our evaluation. That said, an in-depth study on the factors influencing CFRG validity, their practical performance on high-level optimizations, and ways to improve their applicability is part of our future work.

To deliver tighter WCET bounds, *PragMetis*'s annotations allow to introduce contextual knowledge into the analysis. However, to reduce pessimism locally, a fine-grained analysis state is required; this is the primary driver in runtime and memory overhead. We specifically designed *PragMetis* to mitigate these effects by (1) the dominance-based scoping concept, (2) strictly limiting visibility of indexing variables, and (3) deduplicating contexts on the loop, model-graph, and ILP level. These automatic techniques proved effective in our evaluation, even with our current implementation trading deduplication ratio for a faster model-context comparison. However, there exist certain critical instances where the choice of annotation and abstraction can undermine our optimizations. For example, nested loops whose innermost body spawns a call hierarchy that (indirectly) references the various loop-context–indexing variables within its annotations may result in numerous contexts that cannot be deduplicated. Here, *PragMetis* is at a disadvantage compared to automatic, abstract-interpretation–based WCET analyzers with a limited set of abstractions: As we lifted the choice of abstraction to the annotation-language level, we also delegated the trade-off between analysis pessimism and overhead. In general, there is no readily available way to summarize and reduce different analysis contexts automatically. That said, *PragMetis* allows the programmer to control said trade-off between analysis precision and runtime explicitly by manually choosing a more pessimistic abstraction within annotation expressions.

Furthermore, in this paper, we regarded *PragMetis* and automatic analyzers as opposing concepts, however, we see great a perspective in symbiosis there. While user-defined abstractions deliver the ability to create unique, complex abstractions for difficult to bound programs, our evaluations hints that several abstractions at the library boundary are rather basic, such as maximum values for call parameters or data structures' lengths. Such information is usually well within reach of automatic data-flow analysis such as Astreé [16]. The influence of thread-local data-flow analysis on WCET analysis is a well-studied subject [9]. However, the precision of such data-flow analyses usually improve the higher the abstraction level [7, 36]; we expect those tools to harmonize well with *PragMetis*'s source-level abstractions and to outperform automatic analysis approaches operating exclusively on the machine-code level.

6 Related Work

A huge body of related work exists in the context of static WCET tools [2, 19, 24, 28, 40, 43, 50] and their annotation languages [32–35, 54]. However, to the best of our knowledge, no annotation approach exists so far that enables users to express user-defined abstractions and/or opportunistic annotations. For example, the FFX annotation language [10] formulates several powerful annotation types (e. g., loop-context–sensitive annotation), however all related WCET tools or implemented annotation languages [5, 11, 31, 36] only support a subset of FFX (e. g., constant loop bounds). In contrast to proposed annotation-language designs, *PragMetis* is an implemented, publicly available approach handling source-code annotations, such as loop-context–dependent information, in the presence of certain compiler optimizations.

Several approaches exist to transfer flow facts from source level under code optimizations [19, 30, 35, 39, 54]. Schommer et al. presented a source-level annotation approach using compiler-specific builtin function calls [54]. Their approach relies on the CompCert compiler [38] and the aiT WCET analyzer. However, their approach is incapable of loop optimizations as they would jeopardize the annotations' meaning. An approach to circumvent this issue is the co-transformation of flow facts along with individual compiler optimizations [19, 35, 39]. The WCC compiler framework, for instance, constitutes a framework considering WCET-aware code generation and optimization techniques for WCET reduction [19]. It is in particular related, as it employs a syntax for specifying flow information within the source code as custom pragmas. To transfer this information across loop optimizations, such as loop unrolling, WCC relies on specific compiler optimizations that are aware of the respective flow facts. A back-annotation mechanism enables WCC to furthermore transform data from the machine-code back to the source-code level. Sharing the same goal of WCC of supporting compiler optimizations while enabling annotations on source code level, *PragMetis* builds on a CFRG-based transformation to map flow facts between different representation levels of the program. Additionally, *PragMetis* has an annotation language that is even more expressive than stating polyhedral formulas as loop bounds.

Most other WCET analysis tools operate on the binary code, such as *aiT* [2]. The major drawback of this approach is the need to cope with binary code and apply annotations on this level. In contrast, *PragMetis* supports the convenient annotation of the application's source code.

For *PragMetis*, we reuse the syntax of the SWAN framework [55], which targets WCET analyzes of operating systems. While this framework enables users to annotate operating-system-state–specific knowledge at the granularity of system operations (e. g., system-calls), it lacks the features of expressing fine-grained context sensitivity and context differentiation at arbitrary program points *within* a *single operation's* analysis. In contrast to this system-level WCET analyzer, *PragMetis* enables context-sensitive analyses of library functions depending on the callers' behavior, even at loop-context level. SWAN's system-level view and notion of scheduling semantics together with *PragMetis*'s expressivity based on user-defined abstractions are two essential ingredients for future analyses of whole real-time systems.

Kirner et al. compared annotation languages for WCET analysis [33]. Relying on this comparison, *PragMetis* is the first compiler-aware toolchain that supports all annotation features of directly expressing non-rectangular loops, call contexts, loop contexts, and application contexts.

7 Conclusion

The analyses of hard real-time systems demands powerful annotation language in order to determine accurate WCET bounds. In this paper, we presented the *PragMetis* approach that enables users to state user-defined abstractions that are linked to the system's code. Furthermore, *PragMetis*'s concept of opportunistic annotations achieves the aggregation and filtering of knowledge. The open-source implementation of *PragMetis* exploits scope-based representations for the effective evaluation of annotation expressions. The integration of *PragMetis*'s prototype into a WCET- and optimization-aware framework [30] allows developers to annotate at the source-code level while preserving backend compiler optimizations of the clang/LLVM compiler infrastructure. Our evaluation results confirm our major claim: more expressive annotation statements yield more accurate WCET bounds.

Acknowledgments

The authors thank Johannes Kern for his support validating early stages of this research. This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) TRR 89 *Invasive Computing* (project number 146371743) and grants no. SCHR 603/15-2 *LARN*, SCHR 603/10-2 *COKE*, and SCHR 603/9-2 *AORTA*.

Source code of PragMetis*: gitlab.cs.fau.de/pragmetis*

References

[1] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega. 2015. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *Proceedings of the 10th International Symposium on Industrial Embedded Systems (SIES '15)*. 1–10. https://doi.org/10.1109/SIES.2015.7185039

[2] AbsInt. 2021. aiT WCET Analyzers. https://www.absint.com/ait/.

[3] S. Altmeyer, C. Hümbert, B. Lisper, and R. Wilhelm. 2008. Parametric Timing Analysis for Complex Architectures. In *Proceedings of the 14th Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '08)*. 367–376. https://doi.org/10.1109/RTCSA.2008.7

[4] O. Bachmann, P. S. Wang, and E. V. Zima. 1994. Chains of Recurrences – a method to expedite the evaluation of closed-form functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC '94)*. 1–8. https://doi.org/10.1145/190347.190423

[5] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. 2010. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Proceedings of the 8th International Workshop on Software Technolgies for Embedded and Ubiquitous Systems (SEUS '10)*. 35–46. https://doi.org/10.1007/978-3-642-16256-5_6

[6] D. Barkah, A. Ermedahl, J. Gustafsson, B. Lisper, and C. Sandberg. 2008. Evaluation of automatic flow analysis for WCET calculation on industrial real-time system code. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS '08)*. 331–340. https://doi.org/10.1109/ECRTS.2008.37

[7] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. 2011. Static Analysis by Abstract Interpretation of Embedded Critical Software. *SIGSOFT Software Engineering Notes* 36, 1 (2011), 1–8. https://doi.org/10.1145/1921532.1921553

[8] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. 2011. Timing analysis of a protected operating system kernel. In *Proceedings of the 32th Real-Time Systems Symposium (RTSS '11)*. 339–348. https://doi.org/10.1109/RTSS.2011.38

[9] J. Blieberger. 2002. Data-flow frameworks for worst-case execution time analysis. *Real-Time Systems* 22, 3 (2002), 183–227. https://doi.org/10.1023/A:1014535317056

[10] A. Bonenfant, H. Cassé, M. De Michiel, J. Knoop, L. Kovács, and J. Zwirchmayr. 2012. FFX: A Portable WCET Annotation Language. In *Proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS '12)*. 91–100. https://doi.org/10.1145/2392987.2392999

[11] A. Bonenfant, M. de Michiel, and P. Sainrat. 2008. oRange: A Tool For Static Loop Bound Analysis. In *Proceedings of the Workshop on Resource Analysis*. 1–6.

[12] S. Boulier, P.-M. Pédrot, and N. Tabareau. 2017. The next 700 syntactical models of type theory. In *Proceedings of the Conference on Certified Programs and Proofs (CPP '17)*. 182–194. https://doi.org/10.1145/3018610.3018620

[13] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. 2005. Applying static WCET analysis to automotive communication software. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. IEEE, 249–258. https://doi.org/10.1109/ECRTS.2005.7

[14] F. Cassez, R. Rydhof Hansen, and M. C. Olesen. 2012. What is a Timing Anomaly?. In *Proceedings of the 12th International Workshop on Worst-Case Execution Time Analysis (WCET '12)*. 1–12. https://doi.org/10.4230/OASIcs.WCET.2012.1

[15] A. Colin and I. Puaut. 2001. Worst-case execution time analysis of the RTEMS real-time operating system. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS '01)*. 191–198. https://doi.org/10.1109/EMRTS.2001.934029

[16] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2005. The ASTREÉ Analyzer. In *Proceedings of the European Symposium on Programming (ESOP '05)*. 21–30. https://doi.org/10.1007/978-3-540-31987-0_3

[17] A. Ermedahl, F. Stappert, and J. Engblom. 2003. Clustered calculation of worst-case execution times. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '03)*. 51–62. https://doi.org/10.1145/951710.951720

[18] 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 (WCET '16)*. 1–10. https://doi.org/10.4230/OASIcs.WCET.2016.2

[19] H. Falk and P. Lokuciejewski. 2010. A Compiler Framework for the Reduction of Worst-case Execution Times. *Real-time Systems* 46, 2 (2010), 251–300. https://doi.org/10.1007/s11241-010-9101-x

[20] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. Haskins. 2001. A survey of configurable, component-based operating systems for embedded applications. *IEEE Micro* 21, 3 (2001), 54–68. https://doi.org/10.1109/40.928765

[21] G. Gebhard. 2010. Timing Anomalies Reloaded. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET '10)*. 1–10. https://doi.org/10.4230/OASIcs.WCET.2010.1

[22] G. Gonzalez. 2018. dhall: A configuration language guaranteed to terminate. https://github.com/dhall-lang/dhall-haskell/releases/tag/1.14.0.

[23] J. Gustafsson and A. Ermedahl. 2007. Experiences from Applying WCET Analysis in Industrial Settings. In *Proceedings of the 10th International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '07)*. 382–392. https://doi.org/10.1109/ISORC.2007.36

[24] D. Hardy, B. Rouxel, and I. Puaut. 2017. The Heptane Static Worst-Case Execution Time Estimation Tool. In *Proceedings of the 17th International Workshop on Worst-Case Execution Time Analysis (WCET '17)*. 8:1–8:12. https://doi.org/10.4230/OASIcs.WCET.2017.8

[25] S. Hepp, B. Huber, D. Prokesch, and P. Puschner. 2015. The platin Tool Kit – The T-CREST Approach for Compiler and WCET Integration. In *Proceedings of the 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS '15)*. 277–292.

[26] N. Holsti, T. Långbacka, and S. Saarinen. 2000. Using a Worst-Case Execution Time Tool for Real-Time Verification of the DEBIE Software. In *Proceedings of the Data Systems in Aerospace Conference (DASIA '00)*. 1–6.

[27] N. Holsti, T. Långbacka, and S. Saarinen. 2015. Worst-case execution-time analysis for digital signal processors. In *Proceedings of the European Signal Processing Conference (EUSIPCO '21)*. 1–5.

[28] N. Holsti and S. Saarinen. 2002. Status of the Bound-T WCET Tool. In *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET '02)*. 36–41.

[29] B. Huber, D. Prokesch, and P. Puschner. 2012. A Formal Framework for Precise Parametric WCET Formulas. In *Proceedings of the 12th Workshop on Worst-Case Execution Time Analysis (WCET '12)*. 91–102. https://doi.org/10.4230/OASIcs.WCET.2012.91

[30] B. Huber, D. Prokesch, and P. Puschner. 2013. Combined WCET Analysis of Bitcode and Machine Code Using Control-flow Relation Graphs. In *Proceedings of the 14th Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '13)*. 163–172. https://doi.org/10.1145/2465554.2465567

[31] R. Kirner. 2012. The WCET Analysis Tool CalcWcet167. In *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '12)*. 158–172. https://doi.org/10.1007/978-3-642-34032-1_17

[32] R. Kirner, A. Kadlec, A. Prantl, M. Schordan, and J. Knoop. 2008. Towards a Common WCET Annotation Languge: Essential Ingredients. In *Proceedings of the 8th International Workshop on Worst-Case Execution Time WCET Analysis (WCET '08)*. 1–13. https://doi.org/10.4230/OASIcs.WCET.2008.1657

[33] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and A. Kadlec. 2011. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Software & Systems Modeling (SoSyM)* 10, 3 (2011), 411–437. https://doi.org/10.1007/s10270-010-0161-0

[34] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel. 2007. WCET Analysis: The Annotation Language Challenge. In *Proceedings of the 7th Workshop on Worst-Case Execution Time Analysis (WCET '07)*. 1–17. https://doi.org/10.4230/OASIcs.WCET.2007.1197

[35] R. Kirner, P. Puschner, and A. Prantl. 2010. Transforming flow information during code optimization for timing analysis. *Real-Time Systems* 45, 1-2 (2010), 72–105. https://doi.org/10.1007/s11241-010-9091-8

[36] J. Knoop, L. Kovács, and J. Zwirchmayr. 2012. r-TuBound: Loop Bounds for WCET Analysis. In *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '12)*. 435–444. https://doi.org/10.1007/978-3-642-28717-6_34

[37] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO '04)*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

[38] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *Proceedings of the 8th European Embedded Real Time Software and Systems Congress (ERTS '16)*. 1–8.

[39] H. Li, I. Puaut, and E. Rohou. 2014. Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS'14)*. 97–106. https://doi.org/10.1145/2659787.2659805

[40] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. 2007. Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming* 69, 1 (2007), 56–67. https://doi.org/10.1016/j.scico.2007.01.014

[41] Y.-T. S. Li and S. Malik. 1995. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *ACM SIGPLAN Notices*, Vol. 30. 88–98. https://doi.org/10.1145/216636.216666

[42] B. Lisper. 2003. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proceedings of the 3rd Workshop on Worst-Case Execution Time Analysis, (WCET '03)*. 99–102.

[43] B. Lisper. 2014. SWEET – A Tool for WCET Flow Analysis. In *Proceedings of the 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '14)*. 482–485. https://doi.org/10.1007/978-3-662-45231-8_38

[44] B. Lisper, A. Ermedahl, D. Schreiner, J. Knoop, and P. Gliwa. 2010. Practical experiences of applying source-level WCET flow analysis on industrial code. In *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA '10)*. 449–463. https://doi.org/10.1007/978-3-642-16561-0_41

[45] T. Lundqvist and P. Stenstrom. 1999. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th Real-Time Systems Symposium (RTSS '99)*. 12–21. https://doi.org/10.1109/REAL.1999.818824

[46] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang. 2009. A survey of WCET analysis of real-time operating systems. In *Proceedings of the International Conference on Embedded Software and Systems (ICESS '09)*. 65–72. https://doi.org/10.1109/ICESS.2009.24

[47] MISRA Consortium. 2004. *Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004)*.

[48] P. Montag, S. Goerzig, and P. Levi. 2006. Challenges of timing verification tools in the automotive domain. In *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA '06)*. 227–232. https://doi.org/10.1109/ISoLA.2006.52

[49] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard. 2013. The T-CREST Approach of Compiler and WCET-Analysis Integration. In *Proceedings of the 16th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC '13)*. 1–8. https://doi.org/10.1109/ISORC.2013.6913220

[50] Rapita Systems Ltd. [n.d.]. RapiTime - Worst-case execution time (WCET) analysis for critical systems. http://www.rapitasystems.com/products/RapiTime.

[51] M. Rodríguez, N. Silva, J. Esteves, L. Henriques, D. Costa, N. Holsti, and K. Hjortnaes. 2003. Challenges in Calculating the WCET of a Complex On-board Satellite Application. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis (WCET '03)*. 11–15.

[52] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. 2004. Static Timing Analysis of Real-Time Operating System Code. In *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '04)*. 146–160. https://doi.org/10.1007/11925040_10

[53] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi. 2015. T-CREST: Time-predictable Multi-Core Architecture for Embedded Systems. *Journal of Systems Architecture* 61, 9 (2015), 449–471. https://doi.org/10.1016/j.sysarc.2015.04.002

[54] B. Schommer, C. Cullmann, G. Gebhard, X. Leroy, M. Schmidt, and S. Wegener. 2018. Embedded Program Annotations for WCET Analysis. In *Proceedings of the 18th International Workshop on Worst-Case Execution Time Analysis (WCET '18)*. 8:1–8:13. https://doi.org/10.4230/OASIcs.WCET.2018.8

[55] S. Schuster, P. Wägemann, P. Ulbrich, and W. Schröder-Preikschat. 2019. Proving Real-Time Capability of Generic Operating Systems by System-Aware Timing Analysis. In *Proceedings of the 25th Real-Time and Embedded Technology and Applications Symposium (RTAS '19)*. 318–330. https://doi.org/10.1109/RTAS.2019.00034

[56] F. Stappert, A. Ermedahl, and J. Engblom. 2001. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '21)*. 132–140. https://doi.org/10.1145/502217.502240

[57] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. 2003. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*. 625–632. https://doi.org/10.1109/DSN.2003.1209972

[58] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. 2001. Parametric Timing Analysis. In *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '01)*. 88–93. https://doi.org/10.1145/384198.384230

[59] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 1–53. https://doi.org/10.1145/1347375.1347389