WCET Analysis meets Virtual Prototyping: Improving Source-Level Timing Annotations

Martin Becker
Technical University of Munich
Real-Time Computer Systems
Munich, Germany

Marius Pazaj
Technical University of Munich
Real-Time Computer Systems
Munich, Germany

Samarjit Chakraborty
Technical University of Munich
Real-Time Computer Systems
Munich, Germany

Abstract

In this paper we discuss the problem of relating machine instructions to source level constructs, and how it has been addressed in the domains of Virtual Prototyping (VP) and Worst-Case Execution Time (WCET) analysis. It has been handled in different ways, although the goals and requirements between both domains are not far from another. This paper shows that there exists a mutual benefit in exchanging solutions between the two research domains, by demonstrating the applicability and utility of VP methods for WCET analysis, and highlighting their shortcomings.

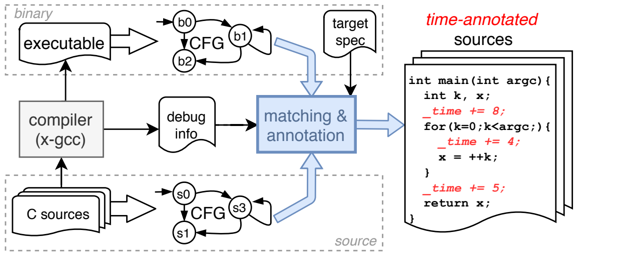
After an evaluation of existing methods, we carefully rework and combine them to a sound and generic mapping algorithm for source-level WCET analysis. As a result, we obtain WCET estimates that outperform classic binary analyzers especially under moderate compiler optimization. Our approach is based on hierarchical flow matching, control-dependency- and dominator-homomorphic maps, and dominator lumping to soundly fill the gaps in the mapping. WCET estimation is performed using Model Checking, which maximally exploits the information available in the source, and highlights remaining weaknesses in the mapping methods.

Last but not least, we discuss further chances of synergy between both research communities which could enable support for more complex microarchitectures with caches, pipelines and speculative execution in both source-level WCET analysis and VP.

CCS Concepts

• **Software and its engineering** → *Real-time systems software*; *Compilers*; *Software post-development issues*;

ACM Reference Format:
Martin Becker, Marius Pazaj, and Samarjit Chakraborty. 2019. WCET Analysis meets Virtual Prototyping: Improving Source-Level Timing Annotations. In *22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES '19), May 27–28, 2019, Sankt Goar, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3323439.3323978



```
int main(int argc){
  int k, x;
  _time += 8;
  for(k=0;k<argc;){
    _time += 4;
    x = ++k;
  }
  _time += 5;
  return x;
}
```

Figure 1: A common problem in Virtual Prototyping and WCET Analysis: matching flow graphs and back-annotation of timing from machine instructions to the source code.

1 Introduction

Relating machine instructions to source code is an often-needed capability in software engineering, that has been repeatedly addressed in different problem contexts, most famously to enable source-level debugging of software [10], so that programmers can follow a program's behavior at the easier-to-understand source code level, instead of instruction level.

The research domains of Virtual Prototyping [22] and Worst-Case Execution Time analysis [34] are also concerned with this problem. This paper is motivated by the history and current body of work in these two domains, since we noticed that the problem has been encountered at different points in time, and handled in different ways. Early work in the WCET community started off with a source-level approach to timing analysis (e.g., [25]), but has quickly been discarded in favor of instruction-level analysis, circumventing the mapping problem in the presence of compiler transformations [34]. However, the price which had been paid ever since, is that of a harder analysis and overestimation. Semantic properties, such as type and range information of variables, are obfuscated or "compiled away", and need to be reconstructed to obtain precise estimates [4, 13, 18]. Today, despite its advantages, source-level WCET analysis, is rarely applied due to this mapping problem. However, when it is applied, timing annotations are generated by tools that reverse-engineer the transformations of a specific compiler version [2, 17], or even require compiler extensions [24].

On the other hand, Virtual Prototyping [22] has the definitive need to model the timing behavior at source-code level, and can accept slight errors. The central goal there is to simulate a program's timing behavior when executed on a specific target, as quickly and accurately as possible, and without the need to set up the target hardware or running painfully slow cycle-accurate instruction set simulators [6, 8]. Instead, the time-annotated source code reflects the target's timing, but is executed on a much faster simulation host. Towards this, automated methods for source-level timing annotations have been proposed in recent years [6, 23, 27]. These can be carried over to WCET analysis, and vice versa, VP can learn from methods that meanwhile have evolved in the WCET domain.

In this paper we specifically argue that WCET analysis can benefit from methods in VP. If the mapping problem can be solved sufficiently precise, then recently emerged technology enables powerful source-level analyses (e.g., model checking [9] and theorem proving), which can exploit the semantic information in the source to produce a tighter WCET estimate than instruction-level analysis. This paper presents strong evidence for this claim when methods from VP are carefully applied.

Our contributions are as follows: (1) We compare the requirements and goals of timing annotations in VP and WCET, (2) we evaluate the applicability of VP methods in WCET applications, and propose ways to fix them and increase their precision, (3) we propose a generic instruction-to-source mapping algorithm for WCET analysis of simple processors, competitive to classic approaches, and (4) we discuss further synergy in both research communities.

1.1 Problem Setting, Challenges and Goals

The overall goal is to annotate the source code of a program with statements keeping track of its execution time, as experienced on a specific target. Fig. 1 illustrates the overall workflow: The program is first cross-compiled for the target processor, which results in a binary/executable with the machine code. We analyze this binary to obtain (1) the control flow of the machine instructions, and (2) the timing behavior of the elements in the control flow. Analogously, we obtain the control flow from the source code.

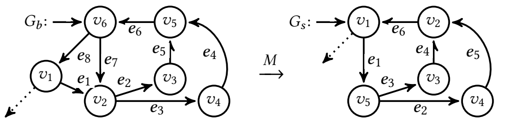


Figure 2: Example of the mapping problem.

The core of the timing annotation problem then is the following: Given two control flow graphs, G_b for the binary and G_s for the source code, we want to establish a mapping

$$M : V_b \rightarrow V_s \tag{1}$$

between the nodes V_b and V_s of the respective control flow graphs, and then annotate the nodes V_s with timing statements, to make the timing behavior visible in the source code. In other words, we want to associate blocks of machine instructions to blocks of source statements. Even without any compiler optimization, G_s and G_b can exhibit differences due to calls to library functions for which no source code exists, but also stemming from architectural constraints. For example, consider the two flow graphs in Fig. 2. The additional node v_1 in G_b must be produced on a 16-bit processor whenever a 32-bit comparison is required. Consequently, the graphs cannot even be considered isomorph in the absence of compiler optimization, and must be expected to differ significantly when optimization is enabled. It follows that the searched-for mapping is in general not bijective, nor even a mathematical function. Additionally, some nodes might be indistinguishable (e.g., v_3 and v_4 in Fig. 2) when only the graph structure is considered. Therefore, debugging information is often consulted to obtain more insights.

Since the two flow graphs are rarely isomorph, the mapping has to consolidate flow differences between them. In general, such differences could be addressed by decompiling the machine instructions back to source code [7]. However, we do not consider this option here, because it not always safe, and undercuts the goals of source-level timing annotations, as discussed later.

Once a mapping has been established, the timing of the binary blocks can be back-annotated to the source code, which subsequently can be used in various ways. In VP, it is compiled and executed on a simulation host, to get an estimate of the execution time during early design, i.e., a form a dynamic analysis. In a WCET context, it can be used to conduct a static analysis that yields an estimate, which is evidence whether the system can obey its real-time requirements. Both contexts show commonalities and differences as follows.

Common goals:

- annotations shall be as precise as possible,
- maintain structure of the source code to retain readability,
- abstract as much detail of the target as possible, to keep complexity low and speed up analysis,
- leave the compiler untouched, and support a range of them,
- avoid reverse-engineering of compiler transformations, and
- tolerate compiler optimization.

Specifics in WCET analysis:

- underestimation is forbidden, but overestimation acceptable,
- no need to support maximum optimization levels, as systems subject to WCET analysis are often verified by additional means, where optimization can be considered a threat to soundness, and is only applied in small doses [11], and
- it is acceptable to forbid certain code constructs to enable analysis [11].

These differences are enough to dictate what methods can be applied. For example, in WCET analysis unmapped nodes cannot be resolved by heuristics, as often done in VP [23, 27]. Furthermore, substituting such blocks by their local WCET is not always possible either, since the WCET of unmapped flow parts may become unbounded without its execution context.

In summary, the goals, requirements and benefits for a source-level timing analysis are slightly different in both communities, but close enough to find synergies in their methods.

2 Background

2.1 Control Flow Graphs and Basic Blocks

Each function in the program – source or binary – can be represented by a control flow graph (CFG). This directed graph G is defined by the tuple

$$G := (V, E, I, F) \tag{2}$$

where V is the set of nodes, $E = V \times V$ the set of edges between them, I is the set of entry nodes and F the set of exit nodes. Without loss of generality, we assume that $||I|| = ||F|| = 1$.

We use the notation $v \succ u$ (or $u \prec v$) to denote that v is a successor of u, possibly via intermediate nodes. Furthermore, throughout this paper, we use subscript "s" for elements in the source CFG, and "b" for those in the binary/instruction CFG.

The nodes V represent *basic blocks*. These are maximal sequences of instructions or statements, with at most one entry and one exit point. Consequently, basic blocks are terminated by branches or indirect jumps, which in turn are represented by the edges E. Therefore, we will use the terms *node* and *basic block* (BB) interchangeably. Last but not least, we assume BBs are also terminated at function calls and returns, so that the callee can be analyzed separately.

2.2 Dominators

In directed graphs G, a node u *dominates* another node v – in short, $u \operatorname{dom} v$ – if all paths from I that reach v must also go through u. In other words, the execution of v implies the (prior) execution of u, but not vice versa. Further, a node u *strictly* dominates another node v if $u \operatorname{dom} v$ and $u \neq v$. Analogously, y *postdominates* x – in short, $y \operatorname{pdom} x$ – if all paths from x towards F must also go through y. Pre- and postdominator trees are data structures that can be computed with standard methods from graph theory, and capture such domination relationships for all nodes in G [15].

```
1 | int main () {
2 |   ⟦int x,y,n,m[2];
3 |   if ((x >= 3⟧) &&
4 |       (⟦y + 3 <= 6⟧) &&
5 |       (⟦m[1] == m[2]⟧)) {
6 |             ⟦n = 2⟧;
```

(a) frontend/source

```
1 | int main () ⟦{
2 |   int x,y,n,m[2];
3 |   if ((x >= 3)⟧⟦&&
4 |       (y + 3 <= 6)⟧⟦&&
5 |       (m[1] == m[2])) {
6 |             n⟧⟦= 2;
```

(b) debug info

Figure 3: Discrepancies of basic block ranges (min='⟦', max='⟧') between compiler frontend and debug info.

2.3 Debugging Information

Debugging information, such as the DWARF [10] format, has been introduced as means to relate instructions to source statements during source-level debugging. In principle, the compiler performs a translation from source code to machine instructions while maintaining a logbook, showing which source locations are causing each instruction, and eventually includes this information in the binary.

Unfortunately, maintaining precise and complete debug information is nontrivial, especially under optimization. Not all optimizers maintain full traceability between source and machine code, which may result in incomplete (some instructions have no source equivalent and vice versa), ambiguous (multiple relations) and imprecise (slight mismatches) information [20, 26, 31].

Debug info vs. Basic blocks: Although debug information relates instructions to source locations, it does not explicitly specify equivalent source ranges for the binary BBs. Instead, potentially each instruction may have its own location info, or may share it with instructions that immediately precede or follow itself in the address space. Therefore, the source range of basic blocks must be reconstructed from the potentially multiple locations in the debug info that belong to its instructions. Let $locs(v)$ be the list of debug locations associated with a BB v, sorted by execution order (which is not necessarily instruction address). Then we can obtain at least two types of ranges: (1) begin $l_{\text{beg}}(v)$ to end $l_{\text{end}}(v)$: denotes the location info of the instructions that are the first and last in $locs(v)$, and (2) min $l_{\text{min}}(v)$ to max $l_{\text{max}}(v)$: captures the extremes of the of source locations in $locs(v)$. These may or may not be identical. For example, the compiler may choose to schedule instructions first that are not at the beginning of the source block, and thus *begin* can be greater than *min*. Furthermore, the last instruction of the block may share location info with prior instructions, and thereby not precisely capture the precise end of the basic block (and in fact, might point to the beginning of a source token, and thus not be column-precise either). Consider the example in Fig. 3 lifted from the *nsichneu* benchmark. There are malign differences in lines 3 and 4, where the source has only one BB in each line, but the binary has two. Another reason for more binary BBs on the same line as in the source may come from inlined compiler intrinsics or implicit library calls. We address all of these issues in our mapping algorithm.

Discriminators have recently been introduced into the debug format, to distinguish between multiple binary basic blocks that fall into the same source code line. For every control transfer that is detected during compilation, the according target instructions are labeled with a new discriminator value if they fall into the same source line as their predecessors. Unfortunately, the DWARF specification allows arbitrary enumeration of the binary basic blocks [10], which means that neither enumeration order nor maximum value have to agree with source locations and block counts (and indeed do differ in practice). As a consequence, it is not safe to pair up the nth source BB on a given line with the binary BB that is labeled with discriminator n. In principle the column information could be used to match binary discriminators to source blocks, but column data is imprecise, as we have seen just before. In fact, we have witnessed cases where column numbers even contradict the semantics of the assembly. Further, having more source BBs than binary discriminators carries no useful information either, since source BBs could have been optimized out.

Last but not least, source code layout has an influence on the debug info, but only to some extent. For example, writing a loop header such that its three parts occupy three different source lines, still results in only one line number for all parts. Consequently, the correctness of debug information may also depend on the source layout, and multiple BBs on one line cannot be avoided in general.

2.4 Source-Level WCET Analysis

The goal of WCET analysis is to estimate the longest time a subprogram takes to terminate, while considering all possible inputs and control flows that might occur, but excluding any waiting times caused by sleep states or interruption by other processes. There exist various methods to obtain such an estimate, with the majority working at instruction level [34]. A source-level WCET analysis can be very precise and provide *safe* (i.e., never smaller than what can be observed when running the program) and *tight* (i.e., as close as possible to observed execution) estimates, and towards that it requires timing annotations in the source code.

The source-level analysis method used here is based on our earlier work presented in [2], and similar to the approach of Kim et al. [17], both based on Model Checking. The tool CBMC [9] is used to determine the WCET of the program by repeatedly proposing a WCET candidate X at the end of the program, and letting the model checker decide whether X can be violated. If so, then a larger candidate must be proposed. If not, then a smaller X can be proposed. The searched-for WCET estimate is the largest X for which a violation can be detected. CBMC builds on SAT/SMT solver technology, and evaluates all program paths precisely. For this reason, the precision of the WCET estimate only depends on the precision of the timing annotations. Naturally, the computational complexity for such precision is higher compared to binary-level approaches, but can be addressed as demonstrated in [2]. Note that, unlike in a VP context, no host compilation takes place, since the time-annotated sources are analyzed directly.

3 Review of Existing Work

We start by reviewing the mapping algorithms from the VP community. Towards this, we define the following two properties needed for WCET analysis: lower-bounding of execution count $f(v)$ under all traces, i.e.,

$$\forall v \in V_b.\ f(v) \leq f(M(v)) \tag{3}$$

and preservation of execution order in the annotations, i.e.,

$$\forall u, v \in V_b.\ v \succ u \Leftrightarrow M(v) \succ M(u) \wedge v \prec u \Leftrightarrow M(v) \prec M(u). \tag{4}$$

Equation (3) ensures that there is no underestimation caused by the mapping, and Eq. (4) is relevant for processors with caches, since there the access order influences temporal behavior.

All mapping algorithms, directly or indirectly, are based on the source locations contained in the debug info. Binary BBs which have similar locations as source BBs are considered to be related to each other. However, since debug locations can be incomplete and ambiguous, further BB properties have been proposed.

3.1 Basic Block Properties

In [6, 23, 28], BBs are eventually matched pair-wise to maximize similarities between their source location, loop membership, structural properties (control dependency and dominators) and the last branching decision. We ignore the last branching decision here, since it only facilitates a "more synchronous" annotation, but has no impact for WCET analysis. Loop membership can be implicitly captured as a side effect of a hierarchical decomposition, which we discuss in the following. Finally, structural properties have proven effective in VP, and are reviewed in detail after the decomposition.

3.2 Hierarchical Flow Partitioning

Several groups have proposed to partition the control flows hierarchically into smaller subgraphs [20, 33]. This limits the impact of annotation errors and ambiguities in the debug information, and reduces the problem size for the node matching algorithm. Specifically, loops and branches were used as partition boundaries. Unmatched regions are eventually lumped into single nodes, and substituted by their local WCET. As a side effect, such a hierarchical matching implicitly captures loop memberships, which has been shown to improve the final node matching [23].

Safety: It is unclear how in general it can be ensured that binary and source partitioning do not diverge during a hierarchical decomposition. In principle it could happen that a difference in the control flow leads two different splits in source and binary, which prevents the optimal mapping from taking place. However, under one of the following assumptions, such an approach can be justified: 1) the boundaries which are used for decomposition are guaranteed to be preserved, or 2) external information about structural changes, e.g., extended debug information, can be taken into account.

In the following, we look at two in VP commonly used properties beyond debug info that are used for matching the individual nodes.

3.3 Dominator Homomorphism Mapping

A *dominator homomorphism* [27] is a partial mapping M between two digraphs that preserves dominator relationships between pairs of nodes. The idea is that such a mapping preserves execution order. Specifically, if the mapping is defined for two binary nodes u and v, then it must hold that

$$\forall v_1, v_2 \in V_b.\ v_1 \operatorname{dom}_b v_2 \leftrightarrow M(v_1) \operatorname{dom}_s M(v_2). \tag{5}$$

Several details should be noted about the dominator homomorphism, which may refute Equations (3) and (4) if not considered.

First, the mapping is not unique. Thus, the algorithm constructing the homomorphic map has a large influence on the annotation precision. The algorithm in [27] is meant to preferably match dominated binary nodes with dominating source nodes, but we found that it cannot guarantee that. Since multiple map entries are added simultaneously during the iterative map construction, false conflicts may be detected: The homomorphism is checked for all pairs in the current map, including the (not yet verified) entries that have just been added. Checking is done one by one in an unspecified order. Whichever inconsistency w.r.t. Eq. (5) is found first, is noted as a conflict. If one of the newly added entries is a bad pick and breaks the dominance homomorphism for another newly added, but correct entry, then we reject both the correct and incorrect entries, although one of would have been the optimal choice. Together with the fact that picking *dominated* nodes first represents a topological order (which is not uniquely defined), conflicts are won by whichever dominated node is picked first.

To fix this algorithm, we must extend the map by one entry at a time. This impacts the run-time negatively, but it yields a better mapping and makes the result independent of the (also unspecified) node iteration order. With these changes, the algorithm indeed preferably ends up in pairing nodes in the chosen preference.

Second, the mapping differs depending on whether strict domination is used or not. Using *strict* domination, two nodes with $u \operatorname{dom} v$ cannot be mapped to the same source node s, since $M(u) \operatorname{dom} M(v) \neq s \operatorname{dom} s$, because s does not strictly dominate itself. Two unrelated binary nodes can however still be mapped to the same source node. This means execution order *is* preserved, as far as it is captured by the dominator relationship. On the other hand, when using a *non-strict* dominator relationship, preservation of execution order depends on whether dom is seen as a property or as an operator.

Mathematically, equation (5) requires us to also evaluate the reverse relationship, i.e., if $v_d \operatorname{dom}_b v_f$, then both of the following conditions must hold

$$v_d \operatorname{dom}_b v_f = true \leftrightarrow M(v_d) \operatorname{dom}_s M(v_f) = true \tag{6}$$

$$v_f \operatorname{dom}_b v_d = false \leftrightarrow M(v_f) \operatorname{dom}_s M(v_d) = false. \tag{7}$$

The algorithm in [27] only tests for the first condition. As a consequence, the *execution order between v_d and v_f can be lost.* This can lead to body nodes in loops to be mapped to loop headers, which *produces overestimation.* On the other hand, if Eq. (7) *is* checked, then such binary nodes are competing for the source node in question. That is, only one of multiple nodes can then be mapped to a single source node, leaving the others unmapped. This again can lead to overestimation in cases where $v_1 \operatorname{dom} v_2$, and v_1 being the only predecessor of v_2, and v_2 the only successor of v_1. Here v_2 steals the source node, leaving v_1 unmapped. This happens quite often in the binary CFG due to *jump threading.* To mitigate this issue, such nodes should be fused in both CFGs to prevent overestimation.

Execution order: Dominator relationships cannot capture all execution orders, thus this method cannot guarantee the preservation of order in the mapping. For example, consider the code fragment `if(u) {v;} z;`, where we have $u \operatorname{dom} v \wedge u \operatorname{dom} z$, and but not $v \operatorname{dom} z$. If v executes, then it must still happen before z in the associated flow, yet, the dominator tree does not carry this information. In practice, this becomes a problem only when debug info between such nodes is ambiguous, which is the case for multiple source blocks on the same line (see Section 2.3).

Execution count: In general, every time multiple BBs are not distinguishable by their debug info nor their their dominance relationships, BBs can be mapped arbitrarily, which may or may not underestimate the execution counts. Consider the one-liner `if (a) {b} else {c} z`. Further, let a and z have the same dominator relationship to the surrounding flow (in both source and binary), such that all of b, c, z are siblings in the dominator tree. Thus, one arbitrarily selected node among b, c, z will map to the source of a, and the others have to be overapproximated later on. This does not violate our execution count property. Now assume z is absent. The mapping of b and c is now arbitrary, possibly violating Eq. (3).

Apart from that, counts are only weakly constrained. If $u \operatorname{dom} v$, then the only guarantee is that $f(v) > 0 \Rightarrow f(u) > 0$, and otherwise their count is unrelated (u can be a loop header which v is not a member of, thus $f(u) > f(v)$ is possible, and vice versa, v may be member of a loop that u is not, such that $f(v) > f(u)$ is possible). Consequently, a dominator homomorphism does not maintain much of the execution count relations.

Under optimization: As the experiments in [28] have shown, the dominance relationship can be changed when the compiler splits complex conditional statements in the source into multiple binary branches. Confusion of BBs cannot be provably avoided.

3.4 Control Flow Dependency Mapping

Müller-Gritschneder et. al [23] proposed an alternative property to match BBs. The idea is to match up those BBs in source and binary that execute under the same condition. Towards this, the first step is to identify *control edges* and *controlled nodes* in both G_b and G_s. An edge $e = (u, v)$ is a control edge if not $v \operatorname{pdom} u$, i.e., if this edge enables the execution of v. Vice versa, the set of nodes $C(e)$ immediately controlled by an edge e is computed as

$$C(e) = \{w \mid w \in path(v, \operatorname{lca}(u, v))\} \setminus A, \tag{8}$$

$$A = \begin{cases} \emptyset & \text{if } u = \operatorname{lca}(u, v) \\ \{u\} & \text{otherwise} \end{cases}, \tag{9}$$

where $\operatorname{lca}(u, v)$ is the least common ancestor of u and v in the postdominator tree, and $path(u, v)$ is the sequence of nodes on the path between (and including) u and v in said tree.

In a second step, they assign labels to all control edges in source and binary, such that edges representing the same decision and outcome obtain the same label. Let $c_X(q)$ be a label representing "decision X, outcome q", and $\mathcal{MC}_{c_X(q)}$ be the set of edges that fall into this label. Given any node v, the new *control dependency property* is formally defined as

$$p_{\text{ctrl}}(v) = \left\{ c_X(q) \,\middle|\, v \in C(e) \wedge e \in \mathcal{MC}_{c_X(q)} \right\}. \tag{10}$$

In other words, the property $p_{\text{ctrl}}(v)$ holds all labels representing the necessary conditions for the immediate execution of the basic block v, whereas multiple labels describe a logical disjunction. Thus, BBs in binary and source which have been assigned the same labels are executing under the same conditions. Note, however, that the mapping is not fully defined, since we may have multiple source nodes matching each binary node.

The key in this approach lies in how the labels X and q are assigned to the edges. Since we want to assign common labels for edges representing the same decision and outcome in binary and source, some form of edge matching is required. The authors propose to use the debug locations of the blocks that are at the outgoing and incoming end of branching edges. That is, both X and q are essentially defined by their source line numbers. Note that this allows multiple binary edges to obtain the same label, as long as they all jump to/from the same lines.

Execution count: Under the assumption that the edge labels represent the decisions and outcomes correctly, each binary node maps to a source node that executes under the same immediate condition. If each of the decision nodes itself maps to their correct conditions, we fulfill Eq. (3) by transitive property. However, using source lines to label the decisions and outcomes as in [23], is *unsafe*. Edges can be confused when multiple BBs share the same line info. Furthermore, there are multiple ways to select the line number, since there is more than one way to define source ranges (see Sec. 2.3). By definition of what constitutes a basic block, we could be tempted to use the location of the entry/exit of the binary BB, which however does not necessarily coincide with the begin/end of the source BB. Apart from that, execution count is well-defined in relation to the controlling edges and co-controlled nodes. Except for loop header nodes, for a control-dependent node v, we know that $f(v) = \sum_{v \in C(e)} f(e)$. In other words, control-dependent nodes execute as often as their controlling edges, except loop headers, which execute more often. In summary, there is no execution count guarantee with the proposed edge labeling.

Execution order is only ensured towards the node that precedes the immediate control edge, and that node in turn is either mapped, thereby obeying its own execution order, or unmapped, and must be overapproximated later. Execution order towards other nodes controlled by the same edge and enclosing conditionals is undefined.

Under optimization: This method can tolerate more compiler optimization than the homomorphism, as shown in [23]. Since the mapping is semantically tied to execution conditions, this method should not produce false mappings under optimization, if labels can be assigned correctly. However, it may still fail to map some blocks. Further, it is in principle possible to detect invariant conditions that have been been optimized out.

3.5 Handling Optimization

Since the mapping methods may either make errors or produce fewer map entries under optimization, several specific optimizations have been addressed in the VP community. For example, in [20, 32], structural loop changes are addressed by mimicking the transformation rules of a certain compiler. This way they handle loop splitting, do-while transformations, unswitching, and blocking. In [28], full loop unrolling is handled in a similar way.

However, a myriad of effects on the CFG is possible, since the optimizers can interact with each other in unexpected ways. It is therefore tedious and likely unsafe to reconstruct them on a case-by-case basis. While some unsupported optimizations could be forbidden to alleviate this problem, it is often not possible to prevent all of them, even when optimization is turned off. Therefore, a generic mapping method must still be able to handle some effects.

In general, all optimization can be seen as either (1) change of execution order (e.g., instruction, trace and superblock scheduling; note that this is different from the *annotation* order required in Eq. (4)), (2) change of execution conditions (e.g., hoisting), (3) duplication (e.g., tail duplication) or (4) complete omission ("optimized out"). These need to be supported in a generic way, to avoid reverse-engineering compiler-specific patterns.

4 A generic mapping algorithm for WCET

This section describes our compiler-independent instruction-to-source mapping algorithm for WCET analysis. Our algorithm builds on all the mapping strategies described in the previous section, but carefully adapts and extends them to prevent underestimation and ensure tight results. The source code is made publicly available at https://github.com/tum-ei-rcs/vigilant-insn2src-mapper.

Our mapping workflow is comprising the following major steps:

(1) Parse binary and source, and compute CFGs.
(2) Annotate binary CFG with debug information.
(3) Process inlining and loop transformations.
(4) Hierarchical decomposition and matching.
(5) Computation of partial mapping.
(6) Overapproximative completion of mapping.
(7) Back-annotation of instruction timing to source.

Detailed elaborations follow now.

4.1 Building annotated control flow graphs

4.1.1 Source code. The control flow graph of an imperative language can be obtained by parsing the source code and processing the abstract syntax tree. This processing is language-specific, since the semantics of the statements define what constitutes a source block. While the CFG is constructed, the nodes must be annotated with the counterpart of the debugging information. In this paper, we use the C language, and the control flows are obtained using the LLVM/clang frontend [19]. The nodes of the resulting CFG are annotated with the following information: 1) range (begin, end) of source code location and 2) list of function calls (referenced CFGs). The resulting annotated sources are ready to be mapped to the binary CFG, which we obtain next.

4.1.2 Machine code. The binary CFG is reconstructed by first cross-compiling the program for the intended target, and then analyzing the semantics of the contained machine instructions. Towards this, we build on existing tools that decode the binary into the individual assembly instructions. We then compute the CFG on an abstraction of the machine instructions, to allow a uniform processing for different targets. In particular, it is only necessary to identify instructions that impact the shape of the CFG, whereas others can be ignored. Specifically, we only discover 1) function calls, 2) jumps and branches, and 3) return instructions. Special care is required for indirect jumps and anonymous function calls (where one instruction may be part of more than one function). Finally, we annotate nodes in the CFG with the debug information, specifically source code locations, and inlining stacks [10]. The resulting annotated binary CFGs are now agnostic to target- and compiler-details, and ready to be mapped to the annotated source CFGs obtained earlier.

4.2 Preprocessing

To prepare both the source and binary CFGs for the upcoming mapping, we pair them by identifier, take note of user-specified loop transformations, and process inlining stacks by copying nodes of inlined functions into their caller, as in [23, 28]. These steps are not further detailed here for space reasons. Next, we solve the problem of multiple BBs per source line by matching the discriminators.

4.2.1 Discriminator Matching. To avoid that the mapping algorithm confuses BBs located at the source line (neither discriminators nor columns are sufficient, see Sec. 2.3), we establish a mapping between binary and source discriminators using their structural information in the CFG. Towards this, we first compute discriminators for the source code by enumerating sets of BBs at each source line individually. Next, we establish a mapping from source to binary discriminators using the corrected dominator homomorphism. Let $D_b(l) = \{d_b^1, d_b^2, \ldots\}$ be the set of binary discriminators at line l, and $D_s(l)$ the source counterpart. Note that binary discriminators d_b may be shared by multiple binary BBs, unlike source discriminators. Therefore, we arbitrarily pick one binary BB from each $d_b \in D_b(l)$, denoted as $pick(d_b)$. Further, let $G'_b(l) = \left(V'_b, E'_b, I, F\right)$ be a reduced graph as follows:

$$V'_b = \{v \mid v = pick(d_b(l)) \wedge d_b \in D_b(l)\} \;\cup\; \{I_b, F_b\}\,, \tag{11}$$

$$E'_b = \{(u,v) \mid path(u,v) \in G_b\}\,. \tag{12}$$

This graph contains the original entry I and exit nodes F of the whole binary flow, all BBs from the current line l to be mapped, and edges between them iff there exists a path between them in G_b. For the source code, $G'_s(l)$ is obtained similarly. We finally apply the dominator homomorphism to compute a map $M'(l) : V'_b \to V'_s$ between the reduced graphs (and thus between the discriminators). The mapping algorithm was corrected as proposed in Sec. 3.4, and additionally we removed ambiguous map entries (siblings in the dominator tree and have an indistinguishable dominator relationship to the surrounding code). The matching preference we use is *dominated* binary blocks to *dominated* source blocks.

We only accept complete maps for discriminators. Otherwise, we explicitly mark all binary and source BBs of the given line l as incompatible, forcing overapproximations later on. Although a proof for the correctness of discriminator matching is still pending, we have not seen any BB confusions in our benchmarks. Alternatively, one could use other parts of the debug info to match the discriminators (i.e., accessed variables), or compare the semantics of source and binary blocks.

Finally, we group matching discriminators under a common label $\mathcal{D}_i(l)$ that is unique per line l, based on the mapping $M'(l)$, i.e.

$$\mathcal{D}_i(l) = \left\{d_s^i\right\} \cup \left\{d_b \;\middle|\; M'(l)(d_b) = d_s\right\} \text{ for } i = 1 \ldots ||D_s(l)||. \tag{13}$$

4.3 Hierarchical decomposition & matching

Before the mapping algorithm starts, we apply a hierarchical decomposition of the CFGs to reduce the overall mapping problem into a number of smaller ones, as described in Sec. 3.2. As a side effect, this enables a safe WCET analysis, as shown later. We recursively partition the CFG into *subflows* along its loops. That is, if a graph G contains a loop, then we replace the loop by a surrogate node, and create a new subflow for the loop, which becomes a child of G. Towards that, we compute a loop nesting tree [15], in which each node represents a loop, and child nodes are contained loops.

The decomposition process is repeated recursively on all subflows, until no further loops are found. This decomposition results in two per-subgraph properties that we can leverage: 1) $u \operatorname{dom} v$ implies $f(u) \geq f(v)$, i.e., dominating nodes execute at least as often as dominated ones, 2) entries nodes are either loop headers or initial nodes in the top-level CFG, both of which can be taken as *fixed-points* in the upcoming node mapping.

Loop optimization: In this algorithm, we only require that existing loops are not vanishing, or that otherwise the user can provide that information during the preprocessing. This requirement is usually satisfied at all except the highest optimization levels. Since those usually clash with WCET analysis, this is not considered a limitation. If support for such aggressive loop optimization is still needed, this could be realized using the optimization reports generated by modern compilers, which indicate which loops have been peeled, blocked and unrolled [12, 29]. Nevertheless, some loops might still vanish if the compiler detects their infeasibility, or even get introduced [2]. To be safe, we check for loop preservation and consult the user for any deviations between source and binary.

4.4 Precise & partial mapping

The actual mapping between nodes in binary and source graphs is now applied independently on pairs of subflows in the hierarchy. We establish a precise but partial mapping, containing only those nodes which have an unambiguous and safe match, as discussed in detail before. Additionally, the map is pre-populated with the fixed-points obtained during the previous step, ensuring that each subflow has at least one mapped node (its entry).

We use the control dependency mapper as described in Sec. 3.4, since it can be modified to preserve execution counts and order (Eq. (3) and (4)), as described in the following. First, a correct edge labeling is ensured by making some corrections to the original control dependency mapping algorithm. Towards this, we require that the lines (excluding columns and discriminators) in the debug info are correct, but missing information is still allowed. Edges (u,v) in binary and source are grouped under labels $\mathcal{MC}_{c_X(q)}$ as

$$\mathcal{MC}_{c_X(q)} = \{(u,v) \mid a(u) = X \wedge b(v) = q\} \tag{14}$$

where functions $a()$ and $b()$ enumerate code locations of both source and binary nodes v according to

$$a(u) = (line(l_{\max}(u)), i)\,, \qquad \text{s.t. } disc(l_{\max}(u)) \in \mathcal{D}_i(line(l_{\max}(u)))$$

$$b(v) = (line(l_{\min}(v)), i)\,, \qquad \text{s.t. } disc(l_{\min}(v)) \in \mathcal{D}_i(line(l_{\min}(v))),$$

with *line* and *disc* yielding only the line number respectively discriminator of a debug location, $l_{\min}(v)$, $l_{\max}(v)$ being these locations as defined in Sec. 2.3, and $\mathcal{D}_i(l)$ is the discriminator label from Eq. (13). Using discriminator labels ensures that only edges between basic blocks that are equivalent in source and binary get the same label, removing the BB confusion that appears in [23]. The BB properties are then computed as defined by Eq. (10).

Furthermore, when computing the controlled nodes of an edge, we exclude self-dependencies of loop headers by always applying the second condition in Eq. (9). This is justified by the hierarchical decomposition, since entries of subgraphs are always loop headers, and the information is therefore not lost. This results in precisely constrained execution counts, and allows us to map a binary BB to any of the matching source BBs whilst guaranteeing preservation of execution count. If execution order shall also be maintained (depends on the microarchitecture, see Sec. 3), this can be established using our modified dominator homomorphism.

4.5 Overapproximative completion of mapping

At this point, some binary BBs may remain unmapped, e.g., due to missing debug information. However, for WCET analysis we must still annotate their timing. We therefore lump their timing into the closest mapped binary BB, where it gets annotated in the corresponding source block.

In particular, we first check for "simple paths". Let P be a path in G_b, with u being one unmapped node in P, and such that P can only be entered at the first node, and only left at the last. Assume further, that P is loop-free, which is guaranteed by our decomposition. Consequently, all nodes on P have the same execution frequency. Given the unmapped node u, we walk along P in both directions. If we encounter a mapped node v, then the timing of u is lumped into v. If none can be found, we lump the timing into the closest ancestor of u in the dominator tree. Note that this is safe only in our mapping, since in the worst case the entry of each subflow is reached, which is a fixed-point in the mapping and therefore guaranteed to be represented in the source code. Furthermore, the graph hierarchy also ensures that dominators are always guaranteed to execute at least as often as their dominated nodes, and thus underestimation is also avoided in the latter case. Execution order can be maintained by lumping iteratively in reverse topological order on G_b.

4.6 Back-annotation of timing to source code

The final step to enable source analysis, is to annotate each source BB with the timing of the mapped binary BBs. How this timing is obtained is beyond the scope of this paper, and we refer the reader to [2, 6]. Since our WCET analysis is a static one, we cannot use dynamically resolved timing control statements as in VP [8].

Additionally, we also propose a different timing annotation format. We make use of C's *comma operator*, which allows making annotations in the middle of complex expressions, and in particular solves the problem with first loop iterations in [23] and the imprecision under complex conditional statements encountered in [28]. Consider the following example, with time annotations as `TIC(x)`:

```
1 | #define TIC(X) __t += (X);
2 | for (int k = 0; TIC(7), k < 100; TIC(2), k++) {
3 |   if ((TIC(2), p != 0) &&
4 |       (TIC(40), strcmp(p, "list") == 0)) {
```

The timing for the loop check can be annotated directly to where it belongs (line 2), such that a split is not necessary. Furthermore, short-circuit expressions can be handled in the same way, as shown.

Table 1: Tightest WCET estimates per method and benchmark. Entries marked in bold where our approach is within 1% of binary-level WCET estimate or better.

		Binary Level			Source Level			
		Sim.	ILP/IPET		Generic Map		Precise Map	
benchmark	opt.	WCET≥	WCET≤	Δ%	WCET≤	Δ%	WCET≤	Δ%
adpcm	O0	44,045	88,872	+101.8	84,744	+99.2	63,710	**+44.6**
	O1	31,699	75,370	+137.8	73,512	**+131.9**	–	–
cnt	O0	8,318	8,376	**+0.7**	8,915	+6.4	8,376	**+0.7**
	O1	1,621	1,663	**+2.6**	1,983	+22.3	–	–
cover-50	O0	3,524	4,100	**+16.3**	58,029	+1,546.6	3,524	≈ **0**
	O1	1,369	2,205	**+61.0**	10,985	+702.4	–	–
crc	O0	130,325	143,646	+10.2	137,163	+5.2	131,652	**+1.0**
	O1	40,612	43,953	**+8.2**	48,052	+18.3	–	–
fdct	O0	22,097	22,097	≈ **0**	25,013	+13.1	22,097	≈ **0**
	O1	7,691	8,628	**+12.2**	8,648	**+12.4**	–	–
fibcall	O0	1,820	1,830	+0.5	1,904	+4.6	1,830	**+0.5**
	O1	6	6	≈ **0**	6	≈ **0**	–	–
insertsort	O0	5,476	5,476	≈ **0**	6,236	+13.8	5,476	≈ **0**
	O1	943	1,519	+61.1	1,142	**+21.1**	–	–
jfdctint	O0	14,143	14,143	≈ **0**	15,906	+12.4	14,143	≈ **0**
	O1	7,427	7,427	≈ **0**	8,361	+12.5	–	–
matmult	O0	984,816	984,816	≈ **0**	1,040,705	+5.6	984,816	≈ **0**
	O1	294,413	294,413	≈ **0**	309,571	+5.1	–	–
ns	O0	56,434	56,450	≈ **0**	58,473	+3.6	56,438	≈ **0**
	O1	9,757	11,410	**+16.9**	14,133	+44.8	–	–
nsichneu	O0	33,203	75,383[b]	+127.0	53,130	+60.0	35,195	**+5.9**
	O1	21,625	44,341	+105.0	35,024	**+40.7**	–	–
ud	O0	34,153	87,560	+156.4	49,907	+46.1	37,304	**+9.2**
	O1	24,885	58,452	+134.9	38,429	**+54.4**	–	–

Δ: upper bound WCET tightness; [b]const. & arith. analysis disabled to avoid timeout

4.7 Experiments

We evaluated our mapping algorithm on the widely-used Mälardalen WCET benchmarks [14], using the WCET estimation method described in Section 2.4. Our target processor is an Atmel Atmega128, which implements a cache-less, in-order, pipelined microarchitecture. As a compiler, we used gcc 7.3 without any changes.

As baselines for our WCET estimates we have performed both random simulations with a cycle-accurate simulator, and WCET estimations with a traditional binary-level WCET analyzer, the *Bound-T* tool [16]. This tool is based on the widely-used IPET/ILP approach [34], but additionally tries to exclude infeasible paths by a combination of call context separation, constant propagation, and arithmetic analysis. The simulation is performed with known worst-case inputs and thus close to the WCET path, which highlights overestimation caused by the mapping, as well as unsafe results.

Moreover, we compare the results of the here-proposed general, compiler-independent mapping with our existing mapper from [2], which was crafted specifically for one compiler version and flag setting, and is thus referred to as *precise mapping*. Since that mapper cannot handle other compiler versions or flags than the ones being reverse-engineered, it can only serve as a reference for optimization level O0. As we will shortly show, the results indicate that our generic mapping is in fact close to that precise mapper, but naturally a generic mapping does not need to be re-developed for every triple of target, compiler (version) and its flags.

The results are summarized in Table 1. For each of the three WCET estimation methods we also give an upper bound on the tightness of the estimate, which is obtained comparing the estimate to the simulated value. Note that this can only be an *upper* bound, since there is no guarantee that the simulation indeed has reached the WCET case at instruction level. In other words, column Δ gives a possibly pessimistic overestimation error for each method.

Notes on Benchmarks The programs have been selected to stress-test various aspects of the mapping. (1) The benchmarks *crc, fdct, fibcall, jfdctint, matmult* and *ud* are single-path programs in the source code. That is, there are no flow dependencies on external variables. Therefore, the binary-level analyzer should produce tight results, and mapping imprecisions become obvious. (2) The benchmark *fibcall* shows what happens if a function is completely optimized out, due to constant propagation and inlining. (3) The benchmark *nsichneu* is a smoke test for BB confusion, missing hierarchy and further also scalability. It consists of 378 nested, often multi-clause if-statements, wrapped in a single loop. The compiler can optimize aggressively here, and debug information is often ambiguous. (4) The benchmark *cover-50* consists of a loop containing a switch-case statement with 50 consequents. We have turned off jump tables, to force the compiler to implement a binary search which differs heavily from the source in its CFG structure. (5) The benchmarks *adpcm, jfdcting, matmult* and *ud* contain implicit library calls caused by arithmetic operations, which source-level analysis can only over-approximate.

Overall impressions: As Tab. 1 shows, the generic mapping is unsurprisingly less precise than a custom-built mapping. Nevertheless, it is only little worse in most cases. Some benchmarks seem to be mapped too coarsely, e.g., *cover-50*, with more than thousand percent overestimation, but also *adpcm*, where the precise mapper had at most 44% overestimation, but the generic mapping more than twice as much. This is analyzed in detail in the next section.

More importantly, the generic mapper also works with optimization, unlike the precise/custom one. It is interesting that half of the time, despite its apparent weaknesses, the generic mapping still outperforms the binary-level WCET analyzer. This suggests that mapping imprecision can be compensated by virtue of detecting more infeasible paths, and makes the *generic and compiler-independent* strategy, as presented here, attractive for WCET analysis.

Results for original VP mapping: Although the unmodified homomorphism and control dependency mappers are potentially unsafe as discussed during our review in Section 3, they seem to work in practice, as well, performing even better. Since this might be useful for VP applications, we give their results in the appendix.

5 Discussion

Although the results show room for improvement, they support the feasibility and usefulness of implementing a generic, compiler-independent mapping and timing annotation method. As we argue in the following, methods from VP can already be beneficial in source-level WCET analysis, but there are opportunities for improving them for both VP and WCET analysis.

5.1 Imprecision of estimates

Since the results of our WCET analysis are maximally precise w.r.t. the source annotations (see Section 2.4), any imprecision of the generic mapping can be attributed to either a bad mapping quality, context beyond the analysis domain, or a combination thereof.

5.1.1 Mapping imprecision. When source and binary CFGs start to diverge, handling of flow differences becomes the key factor for precise results, often enforcing approximations. Unlike applications in virtual prototyping, WCET analysis is required to always *over*approximate, therefore approximation errors do not cancel out, and lead to greater deviations from cycle-accurate simulations.

Consider the two benchmarks *nsichneu* and *cover-50*, which we had chosen because they were considered likely candidates for overapproximation. Indeed, the results show large overestimation compared to the binary-level analyzer and the simulation. However, *nsichneu* is still better than the binary-level analyzer, possibly caused by the overestimation of the latter. A clear case presents itself with *cover-50*. The switch-case was implemented as binary search, and the control-dependency mapping failed to map any of the cases. Consequently, all timing has been lumped in the switch header, accumulating a large error due to the surrounding loop. As a side note, the pure dominator homomorphism (see appendix) was able to map the final case nodes, but also failed for all the binary search decision nodes. In contrast, our *precise mapper* implements a special switch-case handling and ends up with zero overestimation. The strategy used there is to attribute the timing of all decision nodes that lead to a case, to the case itself. Note that this evades any dominator relationship, and therefore cannot be handled by either of the mappers. In fact, such handling could be easily generalized for the completion of the partial mapping.

Another weakness of the mapping are Y-structures in the CFG. Neither of the two joining paths is a dominator of the following code, therefore not a control dependency. Consequently, nodes after such joins remain unmapped. This could be enhanced using dominator *fronts*.

5.1.2 Context beyond analysis domain. Source level analysis suffers a fundamental limitation when it comes to complex control flows that have no source code equivalent, as happening for library calls. For this, consider the *adpcm* benchmark. Although the results are better than the binary-level analyzer, it still is far off the simulation. This program performs many numeric operations that cannot directly be done in hardware, such as multiplication of numbers larger than the architectural word size and arithmetic shifts. Such functionality is provided by the target's C and math libraries, and implemented in assembly language for better performance. A pure source-level analysis thus cannot track the control flow and data dependencies for such functions, and thus must assume the worst-case timing for such library calls. In fact, the same effect occurs in the binary-level analyzer, but for different reasons. Other benchmarks subject to the same limitation are *cnt*, *jfdctint*, *matmult* and *ud*, although those estimates are acceptable in their WCET tightness and still mostly close or better than the binary-level analysis.

Tab. 2 quantifies the overestimation in *adpcm* caused by non-source functions. Here we show the *observed* WCETs for some functions in the simulation (not necessarily occurring together in the same run), and compare them with their WCET estimates. It first should be noted that simulation and worst-case paths are very close together, as indicated by the call counts. However, the number of processor cycles substantially differs for the functions `__ash*di3`, which implement arithmetic shifts using loops. Their execution time is proportional to the shift distance, and thus substituting these calls by a context-agnostic WCET does introduce large overestimation errors. Although it would be possible that the simulation did not include a trace in which these functions were called with their individual worst-case inputs, we know from the source code of this program that the majority of shift operations are invoked with parameters that do not result in their individual WCET. Clearly, this explains why the simulated value is well below the estimates, and confirms that this is not caused by mapping imprecision.

5.1.3 Chances of source-level analysis. On the other hand, Tab. 2 also shows that source-level analysis is able to bound the number of some calls more precisely than the binary-level analyzer. This suggests that source-level analysis can identify more infeasible paths, and thereby does not need to assume that the global WCET is the sum of the WCETs of the individual functions.

Table 2: Upper bound on WCET overestimation Δ due to functions without source code in *adpcm*.

	Sim.max.	WCET Binary-Level		WCET Source-Level	
function	time (calls)	time (calls)	Δ	time (calls)	Δ
__ashrdi3	1,887 (17)	27,013 (17)	**25,126**	27,013 (17)	**25,126**
__ashldi3	880 (10)	15,230 (10)	**14,350**	15,230 (10)	**14,350**
__muldi3	18,420 (60)	19,467 (63)	1,047	15,141 (49)	-3,279
sum	21,187 (87)	61,710 (90)	40,523	57,384 (76)	36,197

5.2 Threats to safety

In this work, we require the debug information to be correct, yet it is allowed to be incomplete and imprecise. Otherwise map entries become incorrect, which might result in an underestimation of the WCET. In the context of WCET analysis this seems an acceptable prerequisite, as usually tools have to be certified whenever timing analysis requires a certain level of assurance [11].

Under optimization: Assuming the labels are correct, the four general optimization effects identified earlier seem to be covered: Change in execution order is tracked by debug locations, and otherwise safely overapproximated. A change of execution conditions can either lead to missing precise maps, or, if debug info is available, detected and carried to the new binary location. Blocks that are optimized out are obviously not part of the mapping domain, and thus supported. Last but not least, duplication can be detected if debug info allows, and is otherwise overapproximated. Despite this verbal argument, a formal proof is still pending and required to qualify this mapping for WCET analysis.

5.3 Further improvements

Beyond the already mentioned refinements in completing the mapping and handling Y-structures, further improvements are possible. For example, this mapping assumes the worst case for time-variable instructions. This mainly concerns branch instructions, which often vary in their timing depending on which edge was taken. To precisely annotate such behavior, the branching instruction needs to be analyzed for its polarity, and considered during edge matching. However, the bigger challenge is to annotate this in the source code, since not all binary edges have equivalent source edges/locations.

More improvements can be made by allowing to alter the source code. Flow differences which cannot be handled, could always be "lifted back" into the source, by modeling the unmappable instructions using source statements. This, in principle, is decompilation. It increases the source complexity, but reduces the overestimation, thereby offering an opportunity for a trade-off.

5.4 Open issues in both research communities

5.4.1 Execution contexts for non-sources. We have illustrated in Sec. 5.1.2 that call contexts should be considered for better timing models. In general, this problem is also present in binary-level analysis and is notoriously difficult, since it traces back to the *Halting Problem*. Still, this is not only a theoretic issue, but some functions, especially library routines, can be truly unbounded without execution context. Thus, WCET analyzers usually are forced to consider execution contexts to some extent [16, 34].

Here, we want to bring the attention to the specifics of execution context in source-level analysis. Further research should focus on establishing an interface between source-level and binary analysis. A mixed analysis might in fact have the best properties from both sides, so that it can exclude more infeasible paths, and eventually provide a tighter WCET estimate than either of them individually.

But neither the VP nor the WCET community have a ready-to-use solution to this problem at the moment. One possible approach could be to calculate parametric WCET estimates, as proposed in [30], and use parametric annotations in the source. We have qualitatively evaluated this approach in [3].

5.4.2 Path-Dependent Timing. Although not the focus of this paper, obtaining the timing for the individual binary BBs can create a henn-and-egg problem. In general BB timing can depend on the execution path, which is however only known after the analysis. Thus, determining the timing of each BB individually cannot always be done with precision. This issue especially needs to be considered in the presence of caches, as discussed below. The limitations otherwise are elaborated in [2].

5.4.3 Conditional instructions. Compilers may perform "if conversions" if the target supports conditionally executed instructions. These cannot be efficiently handled with the presented methods, since they do not appear as branches in the binary CFG. Such instructions can be found in the ARM ISA, and the compiler may use those even when optimization is turned off.

5.4.4 Broader architectural spectrum. Mapping approaches in the VP domain have covered solutions for complex processors with features such as caches, branch prediction and buses [22]. While some of them might be transferable to WCET analysis, others are missing. For example, there are models for data caches, we but are not aware of any instruction cache models. The latter are more problematic, because the exact flow of the binary CFG dictates the timing behavior, and annotations might deviate from that. We are currently working on a source-level instruction cache model which can handle such differences soundly, however at the cost of reduced precision. For maximum precision, it seems inevitable to resort to the decompilation approach.

Furthermore, both domains are currently limited to in-order architectures and bare-metal platforms, at least in deterministic analysis. It seems that these limitations will stay, since out-of-order processing is likely intractable (if even a microarchitectural model exists), and since the use of operating systems implies virtual memory, which would leave memory addresses and thus caching behavior undefined prior to run-time.

6 Related work

Before concluding, we briefly mention further work that was not already referenced earlier.

Compiler modifications: There are many approaches that rely on extensions to the compiler. Some modify existing optimization passes to maintain better debug information [18], whereas others propose methods to transform meta-information among different levels of program representation, such as in the T-CREST project [26].

Simplifying the mapping problem itself: Some approaches modify the compiler such that source and binary flows are guaranteed to be isomorph [5]. Others, in both the WCET and VP areas [6, 26], change the original program by inserting markers that propagate through to the binary. These approaches are orthogonal to this work, since the effect of markers is conceptually similar to having better debug information.

IR-level back-annotation: Several groups have proposed mappings from binary to compiler IR (thus, only considering backend optimization) [8, 21, 32], or only from IR to source (thus, only considering high-level optimization), as in [31]. Both of these approaches have their place next to a full binary-to-source mapping as proposed here, since they can be used as fallback solutions, and are likely necessary to realize precise source-level cache models.

Model identification: Another line of work avoids solving the mapping problem for each program individually by deriving a source-level model once, and subsequently obtaining timing annotations from this model only [1]. The timing behavior is measured on a set of training programs, and then used to derive a timing model for source constructs. Only the model is subsequently used to analyze and annotate new programs. Similarly, there are approaches based on machine learning [22]. We did not consider them for WCET analysis, because there is no guarantee for the correctness of such timing models.

7 Concluding remarks

We have shown that instruction-to-source mapping and timing annotation methods from the Virtual Prototyping (VP) domain can be used for WCET analysis, if applied carefully. The results are quite promising, suggesting that a generic, compiler-independent back-annotation with sufficient precision is possible, especially under moderate optimization, where source-level analysis can identify more infeasible paths than binary analyzers.

During our experiments, we have found several weaknesses that exist for both the VP and WCET domains. Some are easy to address and promise even better annotations. However, others remain unanswered in both domains, namely complete cache models – which is part of our ongoing work – and execution contexts for library calls. Addressing these issues provides further chances of synergy between both research domains.

References

[1] P. Altenbernd et al. Early execution time-estimation through automatically generated timing models. *Real-Time Systems* 52, 6 (2016), 731–760.

[2] M. Becker et al. Scalable and precise estimation and debugging of the worst-case execution time for analysis-friendly processors. *International Journal on Software Tools for Technology Transfer* (2018), 1–29.

[3] M. Becker, R. Metta, R. Venkatesh, and S. Chakraborty. 2019. Imprecision in WCET estimates due to library calls and how to reduce it (WIP Paper). In *Proc. Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, N.N. (Ed.). ACM.

[4] A. Bonenfant et al. When the worst-case execution time estimation gains from the application semantics. (2016).

[5] A. Bouchhima, P. Gerin, and F. Pétrot. 2009. Automatic instrumentation of embedded software for high level hardware/software co-simulation. In *Proc. Asia South Pacific Design Automation Conference*, K. Wakabayashi (Ed.). IEEE, 546–551.

[6] O. Bringmann et al. 2015. The next generation of virtual prototyping: ultra-fast yet accurate simulation of HW/SW systems. In *Proc. Design, Automation & Test in Europe Conference & Exhibition*, W. Nebel and D. Atienza (Eds.). ACM, 1698–1707.

[7] D. Brumley et al. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security Symposium*. 353–368.

[8] S. Chakravarty, Z. Zhao, and A. Gerstlauer. 2013. Automated, retargetable back-annotation for host compiled performance and power modeling. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis*.

[9] E.M. Clarke, D. Kroening, and F. Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, K. Jensen and A. Podelski (Eds.), Vol. 2988. Springer, 168–176.

[10] DWARF Debugging Information Format Committee et al. DWARF debugging information format, version 4. *Free Standards Group* (2010).

[11] R.B. França et al. 2011. Towards Formally Verified Optimizing Compilation in Flight Control Software. In *Proc. Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop (OASICS)*, P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm (Eds.), Vol. 18. Schloss Dagstuhl, 59–68.

[12] Free Software Foundation 2019. *Using the GNU Compiler Collection (GCC)*. Free Software Foundation. https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html (GCC Developer Options).

[13] J. Gustafsson et al. 2006. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *Proc. Real-Time Systems Symposium*. IEEE Computer Society, 57–66.

[14] J. Gustafsson et al. 2010. The Mälardalen WCET Benchmarks: Past, Present And Future. In *International Workshop on Worst-Case Execution Time Analysis (OASICS)*, B. Lisper (Ed.), Vol. 15. Schloss Dagstuhl, Germany, 136–146.

[15] P. Havlak. Nesting of Reducible and Irreducible Loops. *ACM Trans. Program. Lang. Syst.* 19, 4 (1997), 557–567.

[16] N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. *Space Systems Finland Ltd* (2002).

[17] S. Kim, H.D. Patel, and S.A. Edwards. Using a model checker to determine worst-case execution time. *Computer Science Technical Report CUCS-038–09, Columbia University* (2009).

[18] R. Kirner, P.P. Puschner, and A. Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems* 45, 1-2 (2010), 72–105.

[19] C. Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*. 1–2.

[20] K. Lu, D. Müller-Gritschneder, and U. Schlichtmann. 2012. Hierarchical control flow matching for source-level simulation of embedded software. In *Proc. International Symposium on System on Chip*. IEEE, 1–5.

[21] O. Matoussi and F. Pétrot. 2018. A mapping approach between IR and binary CFGs dealing with aggressive compiler optimizations for performance estimation. In *Proc. Asia and South Pacific Design Automation Conference*, Y. Shin (Ed.). IEEE, 452–457.

[22] D. Müller-Gritschneder and A. Gerstlauer. Host-compiled simulation. *Handbook of Hardware/Software Codesign* (2017), 1–27.

[23] D. Müller-Gritschneder, K. Lu, and U. Schlichtmann. 2011. Control-Flow-Driven Source Level Timing Annotation for Embedded Software Models on Transaction Level. In *Proc. Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. IEEE Computer Society, 600–607.

[24] P.P. Puschner. 1998. A tool for high-level language analysis of worst-case execution times. In *Proc. Euromicro Conference on Real-Time Systems*. IEEE Computer Society, 130–137.

[25] P.P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems* 1, 2 (1989), 159–176.

[26] M. Schoeberl et al. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture - Embedded Systems Design* 61, 9 (2015), 449–471.

[27] S. Stattelmann, O. Bringmann, and W. Rosenstiel. 2011. Dominator homomorphism based code matching for source-level simulation of embedded software. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis*, R.P. Dick and J. Madsen (Eds.). ACM, 305–314.

[28] S. Stattelmann, O. Bringmann, and W. Rosenstiel. 2011. Fast and accurate source-level simulation of software timing considering complex code optimizations. In *Proc. Design Automation Conference*, L. Stok, N. Dutt, and S. Hassoun (Eds.). ACM, 486–491.

[29] The Clang Team 2019. *Clang compiler User's Manual*. The Clang Team. https://clang.llvm.org/docs/UsersManual.html (Optimization Reports).

[30] E. Vivancos et al. 2001. Parametric Timing Analysis. In *Proc. Workshop on Languages, Compilers, and Tools for Embedded Systems*, S. Hong and S. Pande (Eds.). ACM, 88–93.

[31] Z. Wang and J. Henkel. 2012. Accurate source-level simulation of embedded software with respect to compiler optimizations. In *Proc. Design, Automation & Test in Europe Conference*, W. Rosenstiel and L. Thiele (Eds.). IEEE, 382–387.

[32] Z. Wang and A. Herkersdorf. 2009. An efficient approach for system-level timing simulation of compiler-optimized embedded software. In *Proc. Design Automation Conference*. ACM, 220–225.

[33] Z. Wang, K. Lu, and A. Herkersdorf. 2011. An approach to improve accuracy of source-level TLMs of embedded software. In *Proc. Design, Automation and Test in Europe*. IEEE, 216–221.

[34] R. Wilhelm and D. Grund. Computation takes time, but how much? *Commun. ACM* 57, 2 (2014), 94–103.

A Results with unsafe VP mapping

		Sim.	Hom. Map		Ctrl Dep Map	
benchmark	opt.	WCET≥	WCET≤	Δ%	WCET≤	Δ%
adpcm	O0	44,045	82,490	+87.2	84,744	+99.2
	O1	31,699	71,812	+126.5	73,512	+131.9
cnt	O0	8,318	8,586	+3.2	8,915	+6.4
	O1	1,621	2,377	+46.6	1,943	+19.8
cover-50	O0	3,524	41,107	+1,066.4	58,029	+1,546.6
	O1	1,369	14,369	+949.5	10,985	+702.4
crc	O0	130,325	137,073	+5.1	137,163	+5.2
	O1	40,612	47,799	+17.6	48,052	+18.3
fdct	O0	22,097	22,257	+0.7	25,013	+13.1
	O1	7,691	8,644	+12.3	8,648	+12.4
insertsort	O0	5,476	5,530	+0.9	6,236	+13.8
	O1	943	1,142	+21.1	1,142	+21.1
jfdctint	O0	14,143	14,159	+0.1	15,906	+12.4
	O1	7,427	8,357	+12.5	8,361	+12.5
matmult	O0	984,816	993,236	+0.8	1,040,705	+5.6
	O1	294,413	309,382	+5.0	309,571	+5.1
ns	O0	56,434	57,839	+2.4	58,473	+3.6
	O1	9,757	13,673	+40.1	14,133	+44.8
nsichneu	O0	33,203	70,945	+113.6	53,130	+60.0
	O1	21,625	43,798	+100.2	35,024	+40.7
ud	O0	34,153	37,888	+10.9	49,907	+46.1
	O1	24,885	30,314	+21.8	38,429	+54.4