Imprecision in WCET Estimates Due to Library Calls and How to Reduce It (WIP Paper)

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

Ravindra Metta
R. Venkatesh
TCS Research
Pune, India

Abstract

One of the main difficulties in estimating the Worst Case Execution Time (WCET) at the binary level is that machine instructions do not allow inferring call contexts as precisely as source code, since compiler optimizations obfuscate control flow and type information. On the other hand, WCET estimation at source code level can be precise in tracking call contexts, but it is pessimistic for functions that are not available as source code. In this paper we propose approaches to join binary-level and source-level analyses, to get the best out of both. We present the arising problems in detail, evaluate the approaches qualitatively, and highlight their trade-offs.

CCS Concepts • **Computer systems organization** → *Real-time systems*; • **Software and its engineering**;

ACM Reference Format:
Martin Becker, Samarjit Chakraborty, Ravindra Metta, and R. Venkatesh. 2019. Imprecision in WCET Estimates Due to Library Calls and How to Reduce It (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3316482.3326353

1 Introduction

Analyzing a program for its worst-case execution time (WCET) remains a challenging task in the context of real-time or safety-critical systems. To obtain a safe upper bound on the WCET, the majority of the WCET estimation techniques work on the binary of a given program [22]. The main reasons for this are: (1) The binary is a more specific characterization of the target's execution behavior of the program than source code (e.g., it shows which instructions the compiler has used to implement high-level behavior), and (2) compiler optimizations, which impact WCET, are difficult to anticipate and account for at source level. Recent advances in source-level WCET analysis [3, 6], have however addressed some of these limitations.

On the other hand, binary level WCET analyzers also suffer from compiler optimizations, which make the binary harder to analyze [15]. In particular, computing call contexts or excluding infeasible flows is hard or even intractable at binary level, enforcing both overapproximations [5, 22] and frequent user input [15]. In contrast, source code allows for precise identification of flow constraints and call contexts, which relieves the user from specifying flow constraints and can provide better estimates due to the semantic information [3]. However, the unavailability of source code for some library functions forces source-level analyzers to assume pessimistic upper bounds on the WCET of such functions, as well as on their side effects on rest of the program. To reduce this pessimism, this paper summarizes approaches to fill the gaps in source-level analysis, by integrating the best parts from both the binary and source level analysis techniques.

2 Problem Statement

We consider a time-annotated source code, such that the execution time of the program when running on a specific target is visible alongside the source statements, e.g.,

```
1 | _time += 2;
2 | if (plt >= 0L) {
3 |   _time += 4;
4 |   fd2 = log10(wd2 + 128);  ///< library call
```

The annotations in lines 1 and 3 increment a global variable `_time` according to processor cycles taken by the underlying instructions, and can be obtained as in [3, 6].

The fundamental limitation of source-level analysis occurs in line 4 with the call to function `log10`, which is not available as source code. Such calls may be explicit, as shown here, or introduced implicitly by the compiler. The latter occurs for operations which cannot directly be done in hardware, such as multiplication of numbers larger than the architectural word size, arithmetic shifts, etc., often implemented in assembly to improve performance. Although source-level analysis can identify the precise call context, it must still assume the worst case timing for such library calls, as their behavior is beyond the realm of the analysis.

Context-Agnostic Approach: Table 1 illustrates that assuming the worst case for non-source callees leads to bad estimates. It quantifies the overestimation in a source-level WCET analysis [3] that has been carried out on the Mälardalen *adpcm* benchmark [12], listing several functions

LCTES '19, June 23, 2019, Phoenix, AZ, USA

without sources. We compare their *observed* WCETs during simulation runs (not necessarily occurring together in the same run) with their respective WCET estimates. It first should be noted that simulation and worst-case paths are close together, as indicated by the call counts. However, the number of processor cycles between simulation and WCET estimate substantially differs for the functions `__ashrdi3` and `__ashldi3`. Those functions implement arithmetic shifts using loops, which makes their execution time proportional to the shift distance. Further, from our manual inspection of the program, we know that these functions are never called with the largest possible shift distance. Consequently, large overestimation occurs when we assume the worst case.

In general, such a context-agnostic approach can also completely fail if a callee is truly unbounded when call parameters are unknown (e.g., `memcpy`). Consequently, not all programs can be handled with this approach, making it incomplete.

Table 1. 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[a]	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

[a]callees included

This problem in WCET estimation gets compounded further when the processor has performance-enhancing features, such as pipelines and caches. The WCET of library calls then depends on the cache state, which is a product of both user and library codes, since both are sharing and competing for the available cache space. As a result, timing analysis of the source and of library codes can no longer be separated without making overly pessimistic assumptions.

3 Approaches

In the following, we discuss approaches towards a source-level WCET estimation supporting a *context-sensitive* analysis of callees whose source code is not available. We specifically focus on issues with caches as an example of performance-enhancing features, since then a solution is no longer trivial.

3.1 Preliminary: A Source-Level Cache Model

First, to leverage the precision of source code analysis for a tighter WCET estimate, the timing behavior of the caches should be made visible to the source code analyzer.

Caches: Without loss of generality, let us consider an A-way-associative cache with n cache lines for instructions or data. By definition of its associativity, there are exactly A different cache lines where a given memory block could be stored. Vice versa, the cache is logically partitioned into n/A groups, called *cache sets*. All blocks of a set compete for its A cache lines. Thus, if more than A blocks map to the same set, then these are *conflicting* with each other, and a replacement policy (such as LRU) is used to decide which one is evicted in favor of a new access. Specifically, let m be the memory address of an instruction or data block, then its cache set is computed as $set(m) = m\%(n/A)$ [11]. Accessing a block that is currently residing in the cache results in a low access time ("hit"). Otherwise ("miss"), the block has to be fetched from slower memory into the cache, possibly triggering an eviction. Note that blocks of different sets cannot influence each other, thus each cache set can be analyzed individually.

Source model: To capture the timing effects of caches, we track memory accesses and the states of their associated cache sets along with the source statements. For the remainder of this paper, we assume that the memory addresses m being accessed by a piece of source code are known, and that the cache implements an LRU replacement policy. The model consists of one array `CSET` for each cache set, and its elements indicate memory blocks being in the cache. Further, let their order reflect their last access time ("age"). With $A = 2$, the source cache model could look as follows:

```
1  | long long CSET[A]={-1, -1}; // initially empty
2  | statement1; // some code accessing address 0x4fd8
3  |   _time += (CSET[0]==0x4fd8 || CSET[1]==0x4fd8) ? HIT : MISS;
4  |   CSET[0] = 0x4fd8; // this block is now cached
5  | statement2; // some code accessing address 0x6f26
6  |   _time += (CSET[0]==0x6f26 || CSET[1]==0x6f26) ? HIT : MISS;
7  |   CSET[1]=CSET[0]; CSET[0]=0x6f26; // 0x4fd8 ages
8  | statement3; // some code accessing address 0x882c
9  |   _time += (CSET[0]==0x882c || CSET[1]==0x882c) ? HIT : MISS;
10 |   CSET[1]=CSET[0]; CSET[0]=0x882c; // 0x4fd8 is now evicted
```

For each potential memory access, we update `CSET` to reflect the age of the blocks in the set (lines 4 and 7), and evict a block if the set is exhausted (line 10). Note that this might not be the most efficient encoding, since that depends on the used source analysis method. To enhance scalability, semantic simplifications and flow-aware overapproximations can be applied [3], but are beyond the scope of this paper.

Difficulties: In general source and binary flows have different structure, primarily due to compiler optimizations. Thus, annotating the source with *precise* cache behavior is in general not possible, necessitating overapproximations.

3.2 The Decompilation Workaround

One obvious way to circumvent the problem of missing sources, is to decompile all such functions back to source code. There have been many attempts at decompiling assembly code to source code, both specifically to timing analysis and as a general tool [17]. However, the decompiled code is typically less precise than the original, due to the information that is lost during compilation [9]. While novel search-based techniques can reconstruct sources closer to the original, decompilers still introduce some overapproximation due to lacking type info and other contextual knowledge, and thus would weaken the WCET estimate. More importantly, decompilation has been shown to be incorrect for some corner cases [4], which could refute the WCET estimate.

3.3 Basic Approach: Parametric Estimates

WCET estimates can be parameterized on the execution context, as shown by Cerný et al. [6] for simple processors. This

would allow pre-computing parametric timing formulae for all library functions, which subsequently are used in the source code analysis in lieu of the callees.

While this would enable a precise *path* analysis, the problem of interdependence in caching behavior between user and library code remains. The cache effects of the library calls are only known after the user code has been compiled and linked with the library, since the instruction addresses m, and therefore the cache sets $set(m)$ are only defined during this process. A pure source-level analysis would therefore have to assume that the cache is in the worst possible state both immediately before and upon return of the call[1]. Hence, the need for a safe analysis could result in an overestimation of the WCET by several orders of magnitude [22].

In the following, we compare different approaches to solve the remaining cache analysis problem in a precise way, by extending parametric WCET estimation such that potential cache states before and after each library call are considered. In principle, the mutual interaction of user and library code calls for a joint parametric analysis, as opposed to precomputing library estimates.

3.4 Approach 1: Path-wise Exploration

Using techniques like symbolic execution [20] and path-wise SAT/SMT formula generation in CBMC [8], paths in the program can be explored one by one. To analyze library calls, we could interleave source and binary analyzers along with the executed code. While such a brute force exploration could be maximally precise, its complexity would be exponential in the number of paths, and thus not scale to real world programs. Even if symbolic execution could handle unknown states and thus avoid to enumerate all paths, analysis time would be proportional to the execution time of the program, yet orders of magnitudes slower than native execution [22].

To cut down on the number of paths to be analyzed, there have been attempts at analyzing only those with the maximum execution time [16]. They typically require a preprocessing stage, which identifies and discards infeasible paths as well as those that lead to cache hits, using techniques like Abstract Interpretation (AI) and Model Checking (MC). The path-wise exploration is then performed only on the remaining paths. Such approaches face two problems: (1) Path exploration is merely shifted to the preprocessing stage, leaving us with the difficulty of selecting a safe subset, and (2) the preprocessing stage suffers from the same problem of having to analyze an obfuscated binary.

Challenges: Selection of candidate paths, pruning unnecessary paths, coupling of/data exchange between binary and source analyzers, handling unknown states.

3.5 Approach 2: Path Summary Estimation

Instead of enumerating all paths in the program, techniques based on Abstract Interpretation (AI) [10] can be used to compute *path summaries*. These are invariants on the states that hold at each program point. Since multiple paths may lead to the same point, the invariants summarize all of them, often by overapproximation. As opposed to the exponential complexity of path-wise exploration, studies show that AI scales linearly in the best case [14], and thus is well suited for larger programs. The workflow would be as follows:

1. Run source analysis to compute call contexts and cache states at each callsite.
2. Run binary analysis for each callee as in [11], considering the current cache state as the initial one, and back-annotate cache sets upon return to the source.
3. Resume source analysis until next callsite is reached.
4. Repeat Steps 2 and 3 until a fix point is reached, that is, all possible cache states have been identified.
5. Use the cache invariants to estimate the WCET of sources and library calls individually with the respective analyzers.

In other words, source and binary analyses are performed alternatingly and iteratively, until all possible cache interactions have been collected. Towards this, support for nondeterminism is required from both analyzers. This approach can build on existing analyzers, e.g., CBMC [8] for source, and OTAWA [1] for binary level.

Challenges: Similarly to path-wise exploration, we need an interface between both analyzers. Additionally, the analysis engine needs to implement a *join* operation to integrate new cache states after each iteration. Moreover, the summaries must be kept precise enough to yield a tight WCET, but AI techniques tend to overapproximate in the presence of complex control flows and data dependencies [5, 21]. Antichains [19] could be used to the cache states with less overapproximation than traditional cache analyses [11], while still maintaining scalability. Additionally, the use of relational domains like in APRON [13] could alleviate the imprecision, but comes at the cost of higher computational complexity.

3.5.1 Path Summaries using Model Checking

To improve precision, path summaries in principle could be determined through Model Checking (MC), at either source or binary level. This technique can precisely [7] identify all possible cache states. However, in contrast to AI, it does not directly yield the set of all possible program states at a given program location, since it was conceived to evaluate Boolean predicates at a given location. Thus, we would have to query possible states, as the following example illustrates:

```
1 | fd2 = log10(wd4);  ///< library call
2 | if (CSET[0] == state1 && CSET[1] == state2 || ...) {
3 |   _time += time_of_log10_1;
4 | } else if (...) {
5 |   _time += time_of_log10_2;
```

The model checker could thus pick the time increment according to the reachable states. Vice versa, we would have to model the cache effects of the callee to update the cache sets for the source code analysis right after this call. However,

[1]Note that, in general, the worst possible state is not necessarily when all blocks are evicted, because of possible *timing anomalies* [22].

computing such an if-circuitry is not possible in the first place, due to the interdependence of user code and library code cache behaviors. Furthermore, this would result in an enumeration of all cache states, if done naively.

3.6 Refining Path Summaries

We can, however, use MC as a secondary analysis method to improve the precision of AI path summaries. Overapproximation can be removed by testing the summaries for their feasibility using MC, as demonstrated for a cache analysis in [2]. Suppose, at a particular library call, AI computed a summary that the cache can be in any of the states {state1, ..., stateN} immediately before the call. MC could determine which of these N states is actually reachable at the call site, by answering reachability questions like the following:

```
1 | assert (CSET[0] != state1 && CSET[0] != state2 && ...);
```

Note that this still separates cache and path analysis, and thus cannot remove all overapproximation of cache states, since the cache analysis assumes that all paths are feasible.

To address the scalability problem of MC, we propose to use *precise abstractions* as follows. From the set of all possible cache states, some are identical from the view of the callee. For LRU policies, we only need to determine the age of the *library* memory blocks in the cache set, whereas the specific user blocks are irrelevant. For example, it does not matter whether the state of `CSET` is `{u1, u2, l1}` or `{u2, u1, l1}` prior to accessing block $l1$ – where u are memory blocks from user code, and l blocks from library code.

Challenges: The same challenges as in the previous approach apply. Additionally, we require abstractions (e.g., as proposed) to alleviate the exponential complexity of MC [7].

4 Comparison of Approaches

We now compare the different approaches qualitatively, to rank their effectiveness. We include a pure source analysis in the comparison, both the context-agnostic approach and the decompilation workaround. The other approaches are called "hybrid", since they combine source- and binary-level analysis as discussed. We the following aspects:

- **Soundness**: an approach is considered sound iff the resulting WCET estimate is always safe.
- **Completeness**: an approach is complete if it yields an upper bound for any program that is indeed bounded.
- **Assumptions**: needs for soundness and completeness.
- **Precision**: degree of avoiding overapproximation.
- **Analysis effort**: expected complexity of analysis. Note that this does not necessarily reflect analysis *time* [7].
- **Implementation effort**: expected amount of work to implement the overall analysis.
- **User inputs**: expected amount of required user interaction, i.e., degree of non-automatism.

Table 2 summarizes our preliminary findings. As it can be seen, there is no clear winner among the presented approaches. A high precision naturally requires spending more analysis effort and requiring more user interaction. Since soundness (subject to assumptions) seems attainable in all approaches, the precision and analysis complexity are deciding factors in practice. Thus, a hybrid path summary suggests itself as compromise.

Table 2. Qualitative comparison of solution approaches.

	Source only		Hybrid Path-wise		Hybrid Path Summary	
	Ctx-Agnostic	Decompile	SymEx	MC	AI	AI+MC
soundness	✓	?	✓	✓	✓	✓
completeness	×	?	×	✓	×	✓
assumptions	(I)	(II)	(I)	(I)+(III)	–	(III)
precision	lowest	high	highest		medium	high
ana. complexity	lowest	(integrated)	exponential		(a)	exponential
impl. effort	lowest	high	medium	high	medium	high
user inputs	none	none	none	(b)	none	(b)

(I) no timing anomalies, (II) tool-specific, (III) completeness threshold known
(a) linear to exponential, depending on domain, (b) completeness threshold

However, considering that flow differences between source and binary virtually always exist as a side effect of compilation and impede an exact back-annotation of timing, it seems that decompilation could be even more practical to solve the two problems at once. Flow differences are simply back-annotated in the source, and thereafter any of the approaches can still be chosen for source-level analysis, yet without the need of coupling source and binary analyzers. However, this puts more burden on the decompiler, terms of how it is allowed to reconstruct source control flows from binary ones. Moreover, the decompiler must be sound itself, which appears to be difficult with available tools [17].

Thus, the compromise between both extremes, a hybrid path summary based on AI, is probably the most amenable to implement, and likely to yield "good enough" results in practice, especially if it turns out similarly effective than its binary-level sibling [21]. Otherwise, a straight-forward extension would be to use more advanced analysis domains in AI (e.g., APRON), or to use MC selectively whenever AI fails to bound the WCET. The specifics, however, are unclear at this point and subject to running experiments.

5 Concluding Remarks

Binary-level WCET estimation suffers from the inability to precisely identify calling contexts of library calls, while source-level WCET estimation suffers from the inability to compute the WCET of library functions. This paper described high-level approaches based on Abstract Interpretation and Model Checking to combine the best of source and binary-level analyses, in order to estimate the WCET more precisely. We presented the involved challenges in making these approaches practicable, and highlighted their relative strengths and weaknesses. From a practical standpoint, approaches based on path summary appear to be a good compromise, yet there does not seem to be a dominating solution. Our ongoing research therefore focuses on a quantitative comparison of the discussed approaches.

References

[1] C. Ballabriga et al. 2010. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Proc. Software Technologies for Embedded and Ubiquitous Systems (LNCS)*, S.L. Min et al. (Eds.), Vol. 6399. Springer, 35–46.

[2] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury. 2013. Precise micro-architectural modeling for WCET analysis via AI+SAT. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, 87–96.

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

[4] D Brumley et al. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Presented as part of the 22nd USENIX Security Symposium*. 353–368.

[5] H. Cassé et al. 2015. A Framework to Quantify the Overestimations of Static WCET Analysis. In *Proc. Workshop on Worst-Case Execution Time Analysis*, F.J. Cazorla (Ed.), Vol. 47. Schloss Dagstuhl, 1–10.

[6] P. Cerný et al. 2015. Segment Abstraction for Worst-Case Execution Time Analysis. In *Proc. European Symposium on Programming (LNCS)*, J. Vitek (Ed.), Vol. 9032. Springer, 105–131.

[7] E.M. Clarke et al. 2004. Completeness and Complexity of Bounded Model Checking, See [18], 85–96.

[8] 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 (TACAS) (LNCS)*, K. Jensen and A. Podelski (Eds.), Vol. 2988. Springer, 168–176.

[9] C. Collberg, C. Thomborson, and D. Low. 1997. *A taxonomy of obfuscating transformations*. Technical Report. Department of Computer Science, The University of Auckland, New Zealand.

[10] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Symposium on Principles of Programming Languages*, R.M. Graham et al. (Eds.). ACM, 238–252.

[11] C. Ferdinand and R. Wilhelm. 1999. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems* 17, 2-3 (1999), 131–181.

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

[13] B. Jeannet and A. Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Proc. Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.), Vol. 5643. Springer, 661–667.

[14] A. Kanamori and D. Weise. 1992. *An Empirical Study of an Abstract Interpretation of Scheme Programs*. Technical Report. Stanford, USA.

[15] B. Lisper et al. 2013. Practical experiences of applying source-level WCET flow analysis to industrial code. *Journal on Software Tools for Technology Transfer* 15, 1 (2013), 53–63.

[16] K. Nagar and Y.N. Srikant. 2015. Path Sensitive Cache Analysis Using Cache Miss Paths. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 43–60.

[17] E. Schulte et al. 2018. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*.

[18] B. Steffen and G. Levi (Eds.). 2004. *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI)*. LNCS, Vol. 2937. Springer.

[19] V Touzeau et al. 2019. Fast and exact analysis for LRU caches. In *Proc. Principles of Programming Languages*, Vol. 3. ACM, 54.

[20] K. Volodymyr et al. 2012. Efficient State Merging in Symbolic Execution. In *Proc. Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 193–204.

[21] R. Wilhelm. 2004. Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone, See [18], 309–322.

[22] R. Wilhelm et al. 2008. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.* 7, 3 (2008), 36:1–36:53.