
# Path-Sensitive Abstract Interpretation for WCET Estimation

SHANGSHANG XIAO, Shandong University, China
MENGXIA SUN, Shandong University, China
WEI ZHANG*, Shandong University, China
NAIJUN ZHAN, Peking University, China and Zhongguancun Laboratory, China
LEI JU, Shandong University, China

Worst-Case Execution Time (WCET) analysis provides an upper bound on a program's execution time and is fundamental to the design and verification of real-time systems. Accurate modeling of cache behavior is critical for WCET estimation, as cache-miss latency is typically orders of magnitude larger than cache-hit latency. Since cache behavior is path dependent, existing methods commonly use abstract interpretation to estimate cache behaviors without enumerating all paths. However, conventional abstract interpretation is context-agnostic—adopting the most conservative case across paths—and thus may produce an overestimated WCET bound. To bridge the gap between scalability and accuracy, we propose a path-sensitive abstract-interpretation-based cache analysis that maintains a set of cache states drawn from critical execution paths to derive context-aware cache behavior. This path-sensitive cache analysis integrates seamlessly into standard WCET frameworks, resulting in tight yet provably sound WCET bounds. Experiments show that our approach improves WCET accuracy by an average of 24.83% without sacrificing scalability.

CCS Concepts: • **Computer systems organization → Real-time systems**; **Embedded systems**.

Additional Key Words and Phrases: Real-time systems, WCET, Abstract interpretation, Cache analysis

**ACM Reference Format:**
Shangshang Xiao, Mengxia Sun, Wei Zhang, Naijun Zhan, and Lei Ju. 2026. Path-Sensitive Abstract Interpretation for WCET Estimation. *Proc. ACM Program. Lang.* 10, PLDI, Article 174 (June 2026), 23 pages. https://doi.org/10.1145/3808252

## 1 Introduction

Worst-case execution time (WCET) analysis produces a safe upper bound on a program's execution time on a specific hardware platform and is fundamental to the design and verification of mission-critical real-time systems, such as those used in autonomous driving, aerospace, and medical domains. The computed WCET bound must satisfy two key requirements: it must be safe, ensuring

*Corresponding author

Authors' Contact Information: Shangshang Xiao, Shandong University, School of Cyber Science and Technology; State Key Laboratory of Cryptography and Digital Economy Security; Key Laboratory of Cryptologic Technology and Information Security, Qingdao, China, 202237103@mail.sdu.edu.cn; Mengxia Sun, Shandong University, School of Cyber Science and Technology; State Key Laboratory of Cryptography and Digital Economy Security; Key Laboratory of Cryptologic Technology and Information Security, Qingdao, China, 202337037@mail.sdu.edu.cn; Wei Zhang, Shandong University, School of Cyber Science and Technology; State Key Laboratory of Cryptography and Digital Economy Security; Key Laboratory of Cryptologic Technology and Information Security, Qingdao, China, sduzhangwei@sdu.edu.cn; Naijun Zhan, Peking University, School of Computer Science; Key Laboratory of High Confidence Software Technology, Beijing, China and Zhongguancun Laboratory, Beijing, China, njzhan@pku.edu.cn; Lei Ju, Shandong University, School of Cyber Science and Technology; State Key Laboratory of Cryptography and Digital Economy Security; Key Laboratory of Cryptologic Technology and Information Security, Qingdao, China, julei@sdu.edu.cn.


ACM 2475-1421/2026/6-ART174
https://doi.org/10.1145/3808252

it never underestimates any feasible execution time, and tight, approximating the actual worst-case execution time as accurately as possible.

Meeting these requirements hinges on precise analysis of micro-architectural features, such as cache policies, pipeline hierarchies, and branch prediction. Among these factors, cache behavior plays a particularly significant role in determining a program's timing characteristics [24]. Because memory access patterns can vary substantially with different program inputs, their effects are highly uncertain; under such uncertainty, the latency of a cache miss can exceed that of a cache hit by several orders of magnitude, which can substantially affect overall program execution time. Consequently, establishing a safe and tight WCET bound is highly dependent on a robust and accurate analysis of cache behavior.

Cache behavior is inherently path-dependent, and performing a comprehensive analysis of concrete cache behavior may require exploring an exponential number of program paths. The widely used abstract-interpretation–based approach [8], which decouples cache behavior analysis from path analysis, remains the predominant method for cache analysis in current research [16]. Specifically, when determining whether a memory reference results in a hit or miss, these methods merge the cache states from all possible incoming paths into a unified abstract cache state. Subsequently, to guarantee the soundness of the estimated WCET bound, the most pessimistic scenario among all possible paths is preserved to decide whether the next memory reference is classified as a cache hit or miss.

While addressing the issue of path explosion, existing methods tend to produce overestimated WCET bounds due to their pessimistic cache behavior analysis. For a basic block with multiple predecessors, the cache behavior of its memory references may vary depending on the previously executed path. A given path can cause some memory references to become misses, but it can also turn references that would miss on other paths into hits. In other words, memory references may exhibit different cache miss/hit behavior with different incoming paths. However, for reasons of efficiency and soundness, the abstract-interpretation-based method takes a union of the miss relations: it joins the distinct cache states from multiple incoming paths into a single abstract cache state and applies the most pessimistic interpretation to each memory reference, assigning each reference the cache position from which it is most easily evicted and classifying all references that miss on any path as misses. This, in turn, produces an overly conservative WCET bound.

To address the aforementioned challenges, this paper presents a path-sensitive cache behavior analysis derived from traditional abstract-interpretation-based methods. Unlike conventional methods that merge all states into a single abstract representation, our approach preserves multiple cache states from different paths, each capturing a distinct slice of contextual information. To avoid the computational cost of exhaustively exploring the potentially exponential number of path-specific states, we introduce a selective merging strategy that consolidates only those states exhibiting "similar" behavior. We also design a metric called Relative-Difference as the default criterion for quantifying similarity, while retaining the flexibility to extend the framework with other similarity measures. This design guarantees that the number of maintained cache states stays below a user-specified threshold while preserving as much path information as possible, enabling a tunable trade-off between analysis precision and computational efficiency. Moreover, during the path-sensitive analysis, our method also captures execution counts of program paths that result in similar cache behavior, allowing the estimated path-sensitive cache behavior to be seamlessly integrated within the standard WCET computation framework.

We integrated the proposed method into the well-known open-source WCET analysis tool Chronos [19] and evaluated it in a real embedded environment using Texas Instruments' TMS320C66x DSP [38]. Compared with the existing approach [3], our method achieves substantial improvements

in both computed WCET bounds and analysis time, without sacrificing scalability. Across benchmarks, the estimated WCET bound was tightened by an average of 24.83%, with gains up to 76.06%, while the mean analysis time remained comparable. Compared to the measured execution time, the computed WCET bound is on average 48% higher. Since measurements may not capture the true worst case, this margin indicates that the proposed method is both safe and accurate. Moreover, experiments show that analysis time increases only linearly with the counts of retained cache states, achieving a good balance between accuracy and scalability.

## 2 Background

Table 1. Notations Introduced in Section 2

| Notation | Description |
|---|---|
| $\mathrm{BB}_i$ | A maximal sequence of consecutive instructions without branches (basic block). |
| $\mathrm{ra}_i$ | Relative age of memory block $\mathrm{m}_i$ in ACS. |
| $m_i$ | A memory block. |
| $\mathrm{Ref}_i$ | A memory reference. |
| $T_i$ | WCET of $\mathrm{BB}_i$. |
| $C_i$ | Execution count of $\mathrm{BB}_i$. |
| $\mathrm{P}^x$ | Ordered sequence of BBs representing an execution path. |
| $\mathrm{ACS}_i^x$ | Abstract cache state of $\mathrm{BB}_i$ along path $\mathrm{P}^x$. |
| $T_i^x$ | WCET of $\mathrm{BB}_i$ along path $\mathrm{P}^x$. |
| $ET_i$ | Accumulated execution time of $\mathrm{BB}_i$ over all paths. |
| $C^x$ | Execution count of path $\mathrm{P}^x$. |
| $C_i^x$ | Execution count of $\mathrm{BB}_i$ along path $\mathrm{P}^x$. |

Existing WCET analysis frameworks can be decomposed into two main components [1]: micro-architectural analysis and control-flow analysis. The results produced by these components are subsequently imported into the Implicit Path Enumeration Technique (IPET) solver as parameters and constraints to compute the WCET. The detailed architecture is shown in Figure 1. This section provides an overview of the key elements of the existing WCET analysis framework as background knowledge.

### 2.1 Abstract Interpretation-Based Cache Behavior Analysis

Existing methods generally adopt the abstract interpretation to predict the cache behavior efficiently. In abstract interpretation–based cache analysis, the abstract cache state (ACS) is introduced to represent a join over all possible concrete cache states that may arise at each basic block. The ACS is defined as:

*Definition 2.1 (Abstract Cache State).* The Abstract Cache State (**ACS**) is a compact abstraction of the cache configurations that may occur at a specific program point during execution. It is represented as a vector of n elements, i.e., ACS[0, …, n-1], where $n$ denotes the cache associativity. The entry $\mathrm{ACS}^i[\mathrm{x}] = \mathrm{m}_j$ means it contains a reference to memory block j with relative age x at basic block $i$. If relative age $x$ does not contain any memory block, then $\mathrm{ACS}^i[\mathrm{x}] = \perp$.

When a memory block is accessed, its relative age in ACS is set to 0. Each time a block within the same cache set is accessed, the ages of all other blocks in that set whose ages are less than the accessed block's age are incremented by 1. If a memory block's age reaches or exceeds $n$, it is considered evicted from the cache.

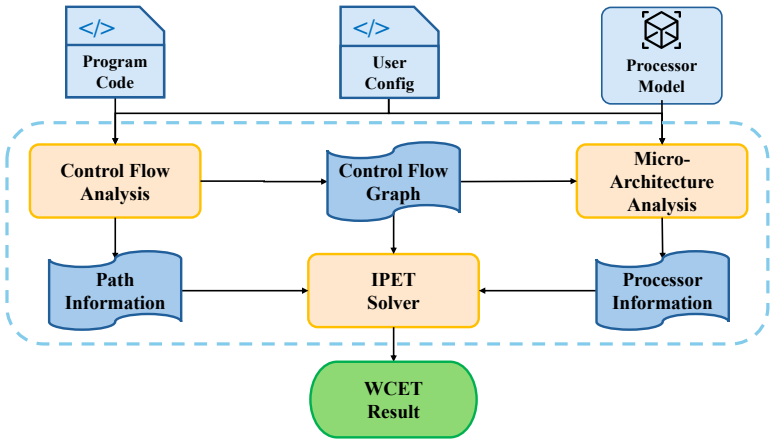



Fig. 1. Analysis Framework

Then, the analysis framework traverses the control flow graph of the target program and computes three different types of ACS for each basic block:

- $\mathbf{ACS}^i_{\mathbf{in}}$ represents the abstract cache state when reaching basic block $\mathrm{BB}_i$, which is computed by merging $\mathrm{ACS}_{\mathrm{out}}$ of all the predecessor basic blocks of $\mathrm{BB}_i$ using the $\mathcal{J}oin$ function. For the entry node of a program fragment, the $\mathrm{ACS}^i_{\mathrm{in}}$ satisfies, for all $x = 0, \ldots, n-1$, $\mathrm{acs}[\mathrm{x}] = \bot$.
- $\mathbf{Gen}^i$ includes the memory blocks accessed by the basic block $\mathrm{BB}_i$.
- $\mathbf{ACS}^i_{\mathbf{out}}$ represents the abstract cache state when leaving $\mathrm{BB}_i$, which is computed by applying the *Update* function based on $\mathrm{ACS}^i_{\mathrm{in}}$ and $\mathrm{Gen}^i$.

As cache analysis predominantly characterizes three categories of memory-access behavior, Always Hit (AH), Always Miss (AM), and Persistent/First Miss (PS/FM), the framework correspondingly adopts three analytical strategies:

- **Must Analysis**: It identifies the set of all memory blocks that are guaranteed to be present in the cache at a given program point. This analysis employs abstract cache states in which a block's position provides an upper bound on its age.
- **May Analysis**: It identifies the set of all memory blocks that may reside in the cache at a given program point. This analysis employs abstract cache states in which a block's position provides a lower bound on its age.
- **Persistent Analysis**: It identifies the set of all memory blocks that are guaranteed to be present in the cache at a given program point after the first reference. This analysis employs abstract cache states in which a block's position provides an upper bound on its age.

Then, two functions, $\mathcal{J}oin$ and *Update*, are defined to operate on ACS:

- ***Update***: The Update function takes two input ACSs, i.e., $\mathrm{ACS}^i_{\mathrm{in}}$ and $\mathrm{Gen}^i$ and produces $\mathrm{ACS}^i_{\mathrm{out}}$, where $\mathrm{ACS}^i_{\mathrm{out}}$ represents the abstract cache state of $\mathrm{BB}_i$ with the initial cache state $\mathrm{ACS}^i_{\mathrm{in}}$ after accessing memory blocks in $\mathrm{Gen}^i$ following the LRU replacement policy.
- ***Join***: The Join function is defined with two input ACSs, i.e., $\mathrm{ACS}^i_{\mathrm{out}}$ and $\mathrm{ACS}^j_{\mathrm{out}}$. Depending on the analysis strategy, the Join function takes different forms:
  - $\mathcal{J}oin_{must}$: It returns an $\mathrm{ACS}_{\mathrm{out}}$ that includes the memory block if it exists in both input ACSs, preserving the oldest age.

- $\mathcal{J}oin_{may}$: It returns an $ACS_{out}$ that includes a memory block if it appears in at least one of the input ACSs; if the block is present in all inputs, the youngest age is preserved.
- $\mathcal{J}oin_{pers}$: It returns an $ACS_{out}$ that includes a memory block if it appears in at least one of the input ACSs; if the block is present in all inputs, the oldest age is preserved.

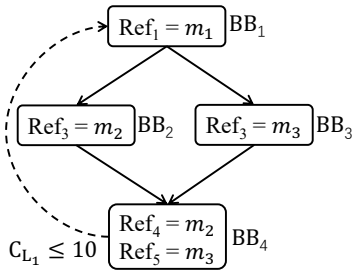



Fig. 2. CFG Illustrating Abstract Interpretation Cache Behaviors

We use a classic example with branching and looping, shown in Figure 2, to illustrate how abstract-interpretation–based cache behavior analysis operates. For simplicity, we assume a cache associativity of 2, focus solely on the must analysis, and consider a fully associative cache configuration. The iterative computation of $ACS_{in}$ and $ACS_{out}$ for each basic block is presented in Table 2.

Table 2. Iterative computation in abstract-interpretation-based cache analysis.

| Iteration | Block | $ACS_{in}$ | $ACS_{out}$ |
|---|---|---|---|
| **First** | $BB_1$ | $\{\bot, \bot\}$ | $\{m_1, \bot\}$ |
| | $BB_2$ | $\{\bot, \bot\}$ | $\{m_2, \bot\}$ |
| | $BB_3$ | $\{\bot, \bot\}$ | $\{m_3, \bot\}$ |
| | $BB_4$ | $\{\bot, \bot\}$ | $\{m_3, m_2\}$ |
| **Second** | $BB_1$ | $\{m_3, m_2\}$ | $\{m_1, m_3\}$ |
| | $BB_2$ | $\{m_1, \bot\}$ | $\{m_2, m_1\}$ |
| | $BB_3$ | $\{m_1, \bot\}$ | $\{m_3, m_1\}$ |
| | $BB_4$ | $\{\bot, \bot\}$ | $\{m_3, m_2\}$ |
| **Third** | $BB_1$ | $\{m_3, m_2\}$ | $\{m_1, m_2\}$ |
| | $BB_2$ | $\{m_1, m_3\}$ | $\{m_2, m_1\}$ |
| | $BB_3$ | $\{m_1, m_3\}$ | $\{m_3, m_1\}$ |
| | $BB_4$ | $\{\bot, m_1\}$ | $\{m_3, m_2\}$ |

Initially, the $ACS_{in}$ of each basic block is empty, denoted by $\{\bot, \bot\}$. During the computation, $ACS_{out}$ is derived from $ACS_{in}$ and Gen. For example, in the first iteration,

$$ACS^1_{out} = \mathit{Update}(ACS^1_{in}, Gen^1) = \{m_1, \bot\}$$

Similarly, $ACS_{in}$ is computed with the $\mathcal{J}oin$ function based on the $ACS_{out}$ of all the predecessor basic blocks. For instance, in the third round,

$$ACS^4_{in} = \mathcal{J}oin_{must}(ACS^2_{out}, ACS^3_{out}) = \mathcal{J}oin_{must}(\{m_2, m_1\}, \{m_3, m_1\})$$

Among the two input ACSs, only $m_1$ appears in both; therefore, $\mathrm{ACS}^4_{\mathrm{in}} = \{\bot, \mathrm{m}_1\}$.

After three iterations, ACS no longer changes; we say it has reached a fixed point [37]. The cache analysis then terminates, yielding the hit/miss outcome for each individual memory access.

With the cache analysis results, the timing of previously most-uncertain memory-access instructions can be determined. Combined with a pipeline model of the processor, the WCET of each individual basic block can be estimated.

### 2.2 Implicit Path Enumeration Technique

With the WCET of each basic block, existing methods leverage the Implicit Path Enumeration Technique (IPET) to compute program-level WCET [41]. IPET formulates WCET computation as an Integer Linear Programming (ILP) problem, thereby avoiding explicit path enumeration and sidestepping the path-explosion problem.

The objective function comprises the sum of the products of the execution counts and the WCET of each basic block, expressed as:

$$\textbf{maximize} \qquad \mathrm{WCET} = \sum_{i=1}^{N} T_i C_i, \tag{1}$$

The execution-count constraints for each basic block are derived from the program's control flow. For simple sequential structures, they follow a straightforward principle: for every basic block, the total count of incoming edges equals the total count of outgoing edges. For the example shown in Figure 2, the constraints are written as:

$$\begin{aligned} \textbf{subject to} \qquad & C_1 = C_{L_1} + 1, \\ & C_2 + C_3 = C_1, \\ & C_4 = C_2 + C_3, \end{aligned} \tag{2}$$

The foregoing constraints do not encompass loop structures. For loops, the maximum iteration count must be supplied—typically by the user or inferred via static analysis. This bound can be encoded as linear relations among execution counts and incorporated into the ILP. For example, in this case, we have:

$$\textbf{subject to} \qquad C_{L_1} \leq 10. \tag{3}$$

By combining the objective function with the aforementioned execution-count constraints, the solver can compute the program's overall WCET.

## 3 Motivation

Current cache behavior analysis conservatively adopts the worst-case scenario (i.e., the cache behavior corresponding to the largest age among all program paths) when encountering multiple program paths to avoid exploring an exponential number of program paths. During cache behavior analysis, it has been observed that maintaining a specific number of distinct cache states instead of merging all cache states into a single conservative abstract state can result in a more precise estimated bound.

For brevity, we use the example in Figure 3, a path-aware version of Figure 2, to illustrate the pessimism of existing methods. The example program has two divergent paths when leaving $\mathrm{BB}_1$, namely, $\mathrm{Path}_1$ (comprising $\mathrm{BB}_1$ and $\mathrm{BB}_2$) and $\mathrm{Path}_2$ (comprising $\mathrm{BB}_1$ and $\mathrm{BB}_3$). In case $\mathrm{Path}_1$ is taken, memory blocks $\mathrm{m}_1$ and $\mathrm{m}_2$ are accessed in $\mathrm{BB}_1$ and $\mathrm{BB}_2$. Subsequently, memory reference $\mathrm{Ref}_4$ in basic block $\mathrm{BB}_4$ hits the cache while $\mathrm{Ref}_5$ misses the cache. Similarly, when $\mathrm{Path}_2$ is taken, memory blocks $\mathrm{m}_1$ and $\mathrm{m}_3$ are accessed, and thus memory reference $\mathrm{Ref}_5$ in basic block $\mathrm{BB}_4$ hits the cache. Therefore, in basic block $\mathrm{BB}_4$, either $\mathrm{Ref}_4$ or $\mathrm{Ref}_5$ hits the cache, and the program experiences 10

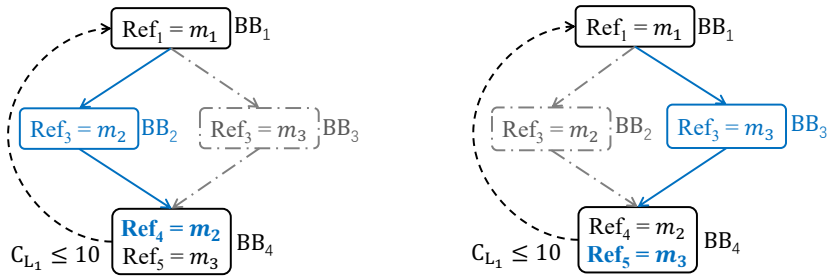



Fig. 3. CFG Illustrating Path-Dependent Cache Behaviors

cache misses in practice. However, as Table 2 shows, existing methods classify both of them as NC and pessimistically assume that neither will hit the cache to obtain a sound WCET bound. Thus, the existing method concludes that this loop incurs 20 cache misses—twice the actual number.

Then, we attempt to maintain two different cache states for basic block $BB_4$ to investigate the pessimism of the conventional method. Here, $T_4^{2\text{-}4}$ denotes the execution time of $BB_4$ when $Path_1$ is taken; $T_4^{3\text{-}4}$ is defined analogously. Thus, the IPET expression for computing the WCET of the program can be extended to:

$$\begin{aligned}
&\textbf{maximize} && \text{WCET} = T_1C_1 + T_2C_2 + T_3C_3 \\
&&& \qquad + T_4^{2\text{-}4}C_4^{2\text{-}4} + T_4^{3\text{-}4}C_4^{3\text{-}4} \\
&\textbf{subject to} && C_1 = C_{L_1} + 1, \\
&&& C_2 + C_3 = C_1, \\
&&& C_4^{2\text{-}4} = C_2, \\
&&& C_4^{3\text{-}4} = C_3, \\
&&& C_{L_1} \leq 10.
\end{aligned} \tag{4}$$

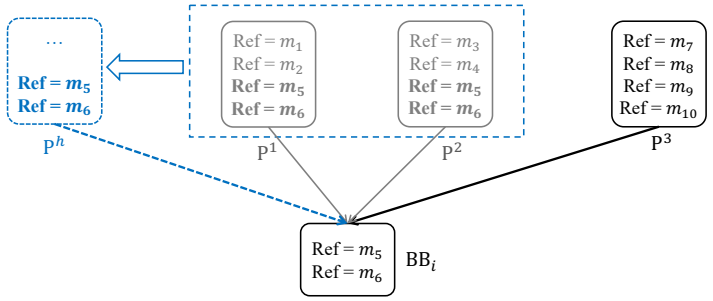
$T_4^{2\text{-}4}$ and $T_4^{3\text{-}4}$ in Equation 4 correspond to the cache states of specific execution paths and are not pessimistically merged by the *Join* function. In this example, $T_4^{2\text{-}4}$ and $T_4^{3\text{-}4}$ both suffer one cache-miss penalty, whereas $T_4$ in Equation 2 suffers two. Moreover, the sum of $C_4^{2\text{-}4}$ and $C_4^{3\text{-}4}$ equals $C_4$. Therefore, the new term $T_4^{2\text{-}4}C_4^{2\text{-}4} + T_4^{3\text{-}4}C_4^{3\text{-}4}$ in Equation 4 is smaller than $T_4C_4$ in Equation 2 while preserving soundness, and thus the WCET bound estimated by Equation 4 is tighter.

The above analysis shows that preserving different cache behaviors of each basic block can result in a more precise estimated WCET bound. However, a basic block may exhibit an exponentially large number of diverse cache behaviors due to potentially encountering an exponential number of incoming paths. Attempting to retain all possible cache behaviors could result in the path explosion issue, rendering it computationally unfeasible. Thus, developing techniques to maintain path sensitivity in cache behavior analysis without compromising computational efficiency is challenging.

## 4 Overview

To address the pessimism introduced by conventional abstract-interpretation-based cache analysis, we present a path-sensitive cache behavior analysis to maintain different cache behaviors among different paths. In this section, we begin by presenting an overview of the proposed method.

The example in Figure 4 depicts a basic block $BB_i$ that has three distinct incoming execution paths, denoted by $P^1$, $P^2$, and $P^3$. During the program's execution, the execution counts of $BB_i$




| | $ACS_{out}$ |
|---|---|
| $P^1$ | $\{m_5, m_6\}$ |
| $P^2$ | $\{m_5, m_6\}$ |
| $P^3$ | $\{m_9, m_{10}\}$ |
| $P^h$ | $\{m_5, m_6\}$ |

Fig. 4. Basic block $BB_i$ with several incoming paths

along a given execution path are contingent upon those of its incoming execution paths. Hence, the accumulated execution time of basic block $BB_i$ can be expressed as follows:

$$ET_i^c = \sum_{x=1}^{n} T_i^x C_i^x = T_i^1 C_i^1 + T_i^2 C_i^2 + T_i^3 C_i^3, \tag{5}$$

The above analysis shows that computing exact execution times over all program paths can lead to precise WCET estimation. However, this also incurs high analysis overhead, as it may need to explore an exponentially large number of program paths. In the example, we find that the cache has the same state prior to entering $BB_i$, namely m6, m5. Regardless of whether the execution follows P1 or P2, both memory references in $BB_i$ are cache hits. Thus, we merge these two states into a single cache state, $ACS^h$, as shown in Figure 4 (the line highlighted in blue). In this case, the execution time of $BB_i$ can be refined as:

$$\begin{aligned} ET_i^h &= T_i^h C_i^h + T_i^3 C_i^3, \\ T_i^h &= T_i^1 = T_i^2, \\ C_i^h &= C_i^1 + C_i^2, \end{aligned} \tag{6}$$

We denote by $T_i^a$ the WCET corresponding to the abstract cache state obtained using the traditional *Join* function. It is easy to see that, since $T_i^a$ corresponds to the most pessimistic case, $T_i^h \leq T_i^a$ and $T_i^3 \leq T_i^a$. Therefore, a tighter WCET bound is derived by refining the execution-time expression of each basic block in equation 2.

The example above shows that, although a program may reach a basic block via an exponential number of paths, the resulting cache behaviors are often highly similar due to the program locality principle. To balance accuracy and scalability, we introduce a path-sensitive cache analysis that merges similar cache states to improve scalability, and retains multiple critical cache states to preserve accuracy.

## 5 Methodology

This section presents the proposed WCET computation method. We first formally define the path-sensitive abstract cache state in Section 5.1 to capture critical cache behaviors. Next, Section 5.2 describes how path-sensitive abstract cache states from different paths are merged through a function called *Shrink*. Then, in Section 5.3, we explain how the WCET is computed based on the analyzed path-sensitive abstract cache states. Finally, Section 5.4 proves the correctness of the proposed method.

### 5.1 Path-Sensitive Abstract Cache State

Our method maintains multiple cache states per basic block, some of which may not correspond to any single concrete program path. For example, $ACS^h$ is a virtual cache state corresponding to $P^h$, an abstract path whose cache effects subsume those of both $P^1$ and $P^2$. To model such virtual entities, we introduce a new abstract domain along with corresponding abstraction functions.

*Definition 5.1 (Hyper-Path).* The Hyper-Path(HP) is a sequence of basic blocks, denoted as $\{BB_m, \ldots, BB_n\}$. It represents an abstract path, which is the merging result of one or more concrete paths ending with this sequence.

Building on the concept of Hyper-Path, we introduce a formal definition of Path-Sensitive Abstract Cache State:

*Definition 5.2 (Path-Sensitive Abstract Cache State).* The Path-Sensitive Abstract Cache State (PS-ACS) is defined as a triple $(HP^x, ACS_i^x, C_i^x)$, which is a cache state representation for $BB_i$ associated with $HP^x$. Specifically:

- $HP^x$ is an ordered sequence of basic blocks that ends with $BB_i$;
- $ACS_i^x$ denotes the abstract cache state of $BB_i$ resulting from executing program paths in $HP^x$;
- $C_i^x$ denotes the sum of execution counts of $BB_i$ over all paths in $HP^x$.

The proposed method maintains a set of PS-ACSs from different paths for each basic block; therefore, Hyper State Space is defined to retain these PS-ACSs for each basic block.

*Definition 5.3 (Hyper State Space).* The Hyper State Space(HSP) is a set of PS-ACSs. It is denoted as $HSP_i = \{\text{PS-ACS}_i^1, \ldots, \text{PS-ACS}_i^m\}$, where $m$ represents the number of Hyper-Paths reaching the basic block $BB_i$.

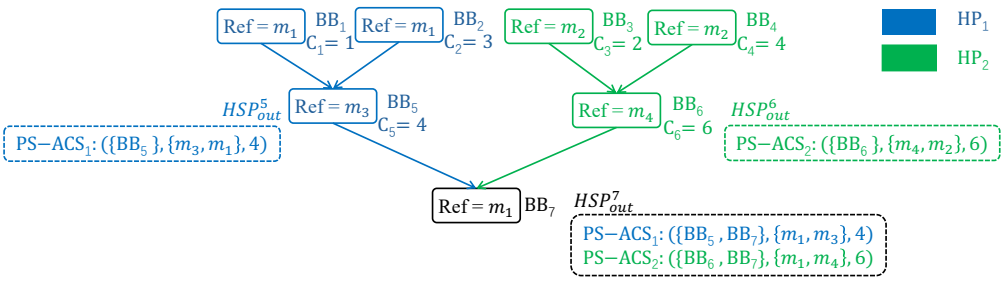



Fig. 5. Relationship between HSP, PS-ACS and HP

Similar to the conventional approach, we define the $HSP_{out}$, $HSP_{in}$ for each basic block, and initialize both $HSP_{in}$ and $HSP_{out}$ of all basic blocks as empty sets. Then, $HSP_{out}$ of each basic block can be computed by:

$$HSP_{out}^i = \textit{H-Update}(HSP_{in}^i, Gen^i)$$

where the definition of the function *H-Update*() is detailed in Algorithm 1.

The *H-Update* computes the change in the cache behavior after executing a specific basic block, and therefore we adopt the *Update* function from the conventional WCET analysis method to compute the update for each PS-ACS (line 3 in Algorithm 1). In addition, the corresponding path of each PS-ACS is also updated by concatenating $BB_i$ (line 4 in Algorithm 1), and the execution counts associated with each PS-ACS remain the same (line 5 in Algorithm 1).

**Algorithm 1:** *H-Update*()

**Input:** $\mathrm{HSP}^i_{\mathrm{in}}, \mathrm{Gen}^i$

1 $\mathrm{HSP}^i_{\mathrm{out}} \leftarrow \varnothing$
2 **for** each PS-ACS$^x$ = ($\mathrm{HP}^x$, $\mathrm{ACS}^x$, $C^x$) $\in \mathrm{HSP}^i_{\mathrm{in}}$ **do**
3  $\mathrm{ACS}^{x'} \leftarrow \mathit{Update}(\mathrm{ACS}^x, \mathrm{Gen}^i)$
4  $\mathrm{HP}^{x'} \leftarrow \mathrm{Append}(\mathrm{HP}^x, \mathrm{BB}_i)$
5  $C^{x'} \leftarrow C^x$
6  PS-ACS$^{x'} \leftarrow (\mathrm{HP}^{x'}, \mathrm{ACS}^{x'}, C^{x'})$
7  $\mathrm{HSP}^i_{\mathrm{out}} \leftarrow \mathrm{HSP}^i_{\mathrm{out}} \cup \{\text{PS-ACS}^{x'}\}$
8 **end**
9 **return** $\mathrm{HSP}^i_{\mathrm{out}}$

Unlike the traditional method that merges the $\mathrm{ACS}_{\mathrm{out}}$ of all predecessor basic blocks into a single $\mathrm{ACS}_{\mathrm{in}}$, our approach first preserves all the $\mathrm{HSP}_{\mathrm{out}}$ of predecessor basic blocks when computing $\mathrm{HSP}^i_{\mathrm{in}}$, and then computes $\mathrm{HSP}_{\mathrm{in}}$ of each basic block by:

$$\mathrm{HSP}^i_{\mathrm{in}} = \mathit{H\text{-}Join}(\{\, \mathrm{HSP}^j_{\mathrm{out}} \mid \mathrm{BB}_j \in \mathrm{pred}(\mathrm{BB}_i) \,\})$$

where the definition of function *H-Join*() is shown in Algorithm 2.

**Algorithm 2:** *H-Join*()

**Input:** $\{\, \mathrm{HSP}^j_{\mathrm{out}} \mid \mathrm{BB}_j \in \mathrm{pred}(\mathrm{BB}_i) \,\}$

1 $\mathrm{HSP}^i_{\mathrm{temp}} \leftarrow \bigcup\{\, \mathrm{HSP}^j_{\mathrm{out}} \mid \mathrm{BB}_j \in \mathrm{pred}(\mathrm{BB}_i) \,\}$
2 **if** $|\mathrm{HSP}_{\mathrm{temp}^i}| >$ S-Thres **then**
3  $\mathrm{HSP}_{\mathrm{temp}^i} \leftarrow \mathit{Shrink}(\mathrm{HSP}_{\mathrm{temp}^i})$
4 **end**
5 $\mathrm{HSP}^i_{\mathrm{in}} \leftarrow \mathrm{HSP}_{\mathrm{temp}^i}$
6 **return** $\mathrm{HSP}^i_{\mathrm{in}}$

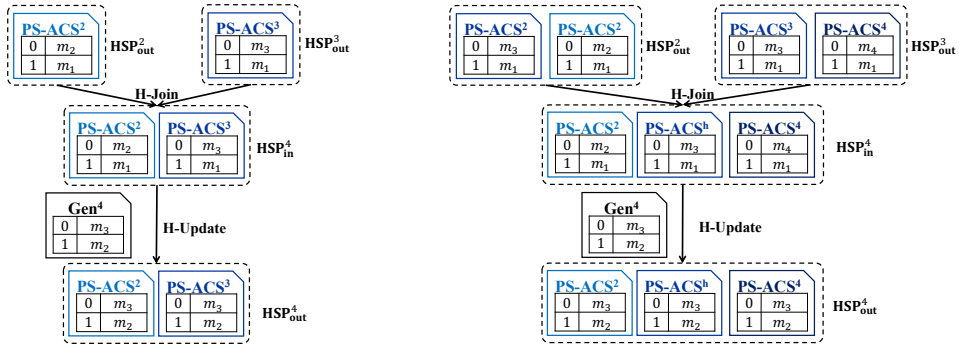
A simple example of the hierarchical relationship between HSP, PS-ACS, and HP is shown in Figure 5. The *H-Join* first keeps all the PS-ACSs from all the predecessor's $\mathrm{HSP}_{\mathrm{out}}$ sets. To prevent the number of maintained PS-ACSs from growing exponentially, we introduce a threshold, denoted by S-Thres, to restrict the quantity of remained PS-ACSs. If the count of the remaining PS-ACSs exceeds S-Thres, we employ a function *Shrink* to combine certain PS-ACSs (line 2-3 in Algorithm 2). The function *Shrink* is detailed in Section 5.2.

We again use the program in Figure 3 to illustrate the *H-Join* and *H-Update* of the path-sensitive cache behavior analysis when $|\mathrm{HSP}_{\mathrm{in}}|$ does not exceed S-Thres. The detailed process is depicted in Figure 6a. When computing $\mathrm{HSP}^4_{\mathrm{in}}$, since there are only two different PS-ACSs at this point, *H-Join* simply adds them into the set.

$$\mathrm{HSP}^4_{\mathrm{in}} = \mathit{H\text{-}Join}(\mathrm{HSP}^2_{\mathrm{out}}, \mathrm{HSP}^3_{\mathrm{out}}) = \begin{bmatrix} (\mathrm{HP}^2,\ \{\mathrm{m}_2,\ \mathrm{m}_1\},\ C^2) \\ (\mathrm{HP}^3,\ \{\mathrm{m}_3,\ \mathrm{m}_1\},\ C^3) \end{bmatrix}$$

The $\mathrm{HSP}_{\mathrm{out}}$ of each basic block can be computed by applying the update function on each PS-ACS. For example:

$$\mathrm{HSP}^4_{\mathrm{out}} = \mathit{H\text{-}Update}(\mathrm{HSP}^4_{\mathrm{in}}, \mathrm{Gen}^4) = \begin{bmatrix} (\mathrm{HP}^2,\ \{\mathrm{m}_3,\ \mathrm{m}_2\},\ C^2) \\ (\mathrm{HP}^3,\ \{\mathrm{m}_3,\ \mathrm{m}_2\},\ C^3) \end{bmatrix}$$




(a) Less than S-Thres (b) More than S-Thres

Fig. 6. Example of *H-Update* and *H-Join* with S-Thres = 3

## 5.2 Similarity-Based Merging Policy

In this section, we provide a detailed explanation of how the *Shrink* function works to balance the scalability and accuracy. Among a set of PS-ACSs, merging which of them into one new PS-ACS stands as pivotal in enhancing the precision of the estimated WCET bound. The key insight of our method is that merging two "similar" PS-ACSs into a single PS-ACS preserves a larger portion of the information originally encapsulated in those "similar" PS-ACSs. Consequently, we argue that the merged PS-ACS from "similar" PS-ACSs is more effective at articulating the precise behavior of the system. To quantitatively measure the differences between different cache behaviors, we introduce a metric termed Relative-Difference, which serves as the criterion for the subsequent *Shrink* algorithm.

*Definition 5.4 (Relative-Difference).* The Relative-Difference (R-DIFF) between two PS-ACSs is defined as the sum of the squared differences in the relative ages of all memory blocks contained in either PS-ACS. For an abstract cache state ACS, let $\mathcal{B}(\text{ACS})$ denote the set of memory blocks appearing in ACS. Then, the Relative-Difference between two PS-ACSs is defined as

$$R\text{-}Diff(\text{PS-ACS}^x, \text{PS-ACS}^y) = \sum_{m \in \mathcal{B}(\text{ACS}^x) \cup \mathcal{B}(\text{ACS}^y)} (\text{ra}_m^x - \text{ra}_m^y)^2.$$

If a memory block is not present in one of the PS-ACSs, its relative age in the PS-ACS is considered to be the cache associativity.

We adopt R-DIFF because age difference directly reflects divergence in reuse distance between paths, and memory blocks with similar ages in different cache states are therefore likely to exhibit similar future behavior. Based on this observation, we consider blocks with smaller age differences to be more "similar" and use the squared age difference R-DIFF as a merge metric. Its quadratic nature imposes a stronger penalty on larger positional discrepancies, aligning with our goal of merging only states with sufficiently aligned cache behavior. It is worth noting that the proposed framework is not tied to this specific metric; R-DIFF can be seamlessly replaced with alternative similarity measures or generalized to accommodate broader notions of state similarity.

In case the cardinality of $\text{PS-ACS}_{\text{in}}$ exceeds S-Thres, we introduce the function *Shrink* that iteratively identifies the pair of PS-ACSs with the smallest R-DIFF to merge until the cardinality of $\text{PS-ACS}_{\text{in}}$ is no larger than S-Thres. The overall procedure of the function *Shrink* is defined in Algorithm 3.

When a pair of PS-ACSs with the smallest R-DIFF are identified (line 3-9 in Algorithm 3), we first adopt the *Join* function from the conventional method to merge them (line 11 in Algorithm 3).

**Algorithm 3:** *Shrink*()

**Input:** $\text{HSP}^i_{\text{temp}}$

1 **while** $|\text{HSP}^i_{\text{temp}}| > \text{S-Thres}$ **do**
2  R-DIFF $\leftarrow \infty$
3  **for** each PS-ACS$^m \in \text{HSP}^i_{\text{temp}}$ **do**
4  **for** each PS-ACS$^n \in \text{HSP}^i_{\text{temp}}$, $n \neq m$ **do**
5  **if** $\sum (\text{ra}^m - \text{ra}^n)^2 <$ R-DIFF **then**
6  R-DIFF $\leftarrow \sum (\text{ra}^m - \text{ra}^n)^2$, PS-ACS$^x \leftarrow$ PS-ACS$^m$, PS-ACS$^y \leftarrow$ PS-ACS$^n$
7  **end**
8  **end**
9  **end**
10  $\text{HSP}^i_{\text{temp}} \leftarrow \text{HSP}^i_{\text{temp}} \setminus \{\text{PS-ACS}^x, \text{PS-ACS}^y\}$
11  $\text{ACS}^h \leftarrow \mathit{Join}(\text{ACS}^x, \text{ACS}^y)$
12  $L_x \leftarrow$ length of HP$^x$, $L_y \leftarrow$ length of HP$^y$
13  $k \leftarrow 0$
14  **while** $k < L_x$ **and** $k < L_y$ **and** $\text{BB}^x_{L_x-k} = \text{BB}^y_{L_y-k}$ **do**
15  $k \leftarrow k + 1$
16  **end**
17  **if** $k = 0$ **then**
18  $\text{HP}^h \leftarrow$ empty sequence
19  **end**
20  **else**
21  $\text{HP}^h \leftarrow [\text{BB}^x_{L_x-k+1}, \ldots, \text{BB}^x_{L_x}]$
22  **end**
23  $C_h \leftarrow C_x + C_y$, PS-ACS$^h \leftarrow (\text{ACS}^h, \text{HP}^h, C^h_i)$, $\text{HSP}^i_{\text{temp}} \leftarrow \text{HSP}^i_{\text{temp}} \cup \{\text{PS-ACS}^h\}$
24 **end**
25 **return** $\text{HSP}^i_{\text{temp}}$

The merged state, computed via the traditional *Join* function, provides a safe upper bound of the original states, although it may trade some precision for scalability. A formal proof that the resulting analysis remains sound for WCET estimation is given in Section 5.4. Subsequently, we compute the associated HP by retaining the longest common suffix of the basic block sequences from HP$^x$ and HP$^y$ (lines 13–22 in Algorithm 3), as this suffix is sufficient to capture the aggregated execution counts of all merged program paths. The execution counts are computed by summing the execution counts of each PS-ACS as shwon in Algorithm 3 (line 23).

Table 3. Results of R-DIFF between different PS-ACSs.

| **R-DIFF** | PS-ACS$^1$ | PS-ACS$^2$ | PS-ACS$^3$ | PS-ACS$^4$ |
|---|---|---|---|---|
| PS-ACS$^1$ | – | 28 | 44 | 36 |
| PS-ACS$^2$ | 28 | – | 6 | 36 |
| PS-ACS$^3$ | 44 | 6 | – | 28 |
| PS-ACS$^4$ | 36 | 36 | 28 | – |

We use Figure 7 as an example to illustrate how the *Shrink* function with R-DIFF works. We assume an associativity of 4 and set S-Thres to 3. For simplicity, only the must analysis is shown.

Initially, the $HSP_{in}$ of $BB_1$ and $BB_2$ both contain two PS-ACSs, simplified as $PS\text{-}ACS^1$ and $PS\text{-}ACS^2$ for $BB_1$, and $PS\text{-}ACS^3$ and $PS\text{-}ACS^4$ for $BB_2$. For $BB_3$, we first aggregate all predecessors' $HSP_{out}$ to form its $HSP^3_{in}$. At this stage, since the cardinality of $HSP^3_{in}$ exceeds the threshold S-Thres, the *Shrink* function is invoked to prevent state explosion.

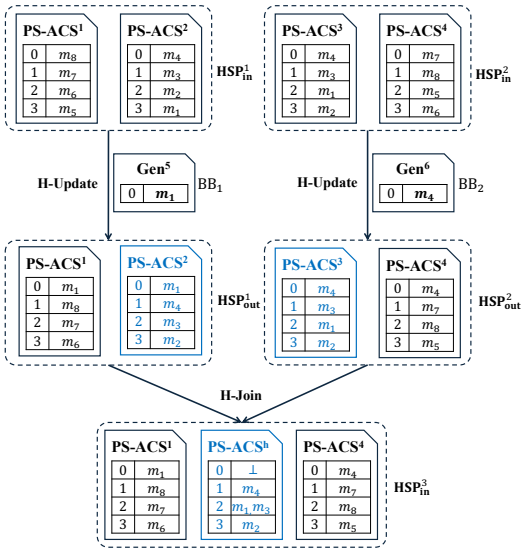



Fig. 7. Example of *H-Update* and *H-Join* with *Shrink*

The R-DIFF between each pair of PS-ACS states can be computed as shown in Table 3. According to the table, $PS\text{-}ACS^2$ and $PS\text{-}ACS^3$ exhibit the minimal R-DIFF, indicating maximal similarity. Therefore, we merge these two cache states:

$$\begin{aligned} \text{PS-ACS}^h &= \textit{H-Join}\left(\text{PS-ACS}^2,\ \text{PS-ACS}^3\right) \\ &= \textit{H-Join}\begin{pmatrix}(\text{HP}^2,\ \{\text{m}_1,\ \text{m}_4,\ \text{m}_3,\ \text{m}_2\},\ C^2),\\ (\text{HP}^3,\ \{\text{m}_4,\ \text{m}_3,\ \text{m}_1,\ \text{m}_2\},\ C^3)\end{pmatrix} \\ &= (\text{HP}^h,\ \{\perp,\ \text{m}_4,\ \{\text{m}_1,\ \text{m}_3\},\ \text{m}_2\},\ C^{\text{h}}) \end{aligned}$$

Thus, the updated set of PS-ACSs in $HSP^3_{in}$ is expressed as:

$$\text{HSP}^3_{\text{in}} = \begin{bmatrix} (\text{HP}^h,\ \{\perp, \text{m}_4, \{\text{m}_1,\ \text{m}_3\}, \text{m}_2\},\ C^{\text{h}}) \\ (\text{HP}^1,\ \{\text{m}_1, \text{m}_8, \text{m}_7, \text{m}_6\},\ C^1) \\ (\text{HP}^4,\ \{\text{m}_4, \text{m}_7, \text{m}_8, \text{m}_6\},\ C^4) \end{bmatrix}$$

### 5.3 WCET Computation with HSP

By iteratively updating $HSP_{out}$ and $HSP_{in}$ for each basic block until a fixed-point is reached, we obtain the cache behavior of each memory reference on different abstract paths. If the accessed memory block is in PS-ACS of $HSP_{in}$ by the must analysis, the memory reference is classified as always hit on the corresponding path, while if the accessed memory block is not in PS-ACS of $HSP_{in}$ by the may analysis, the memory reference is classified as always miss on the corresponding path. Then, with the pipeline model of the target processor, the WCET of each basic block on

different paths can be simulated. Subsequently, these results are substituted into our enhanced IPET formulation to compute the overall WCET using the generated Hyper-Path and refined IPET constraints.

The refined IPET consists of two types of constraints: (i) standard constraints that describe the conventional control-flow relations, and (ii) path-sensitive constraints that capture the relationships among the execution counts of hyper-paths. The standard constraints are derived from the branching and control-transfer relations in the control flow, as in the IPET formulation used in abstract-interpretation–based analysis. The path-sensitive constraints, emitted by *Shrink*, preserve the execution-count relations among merged Hyper-Paths.

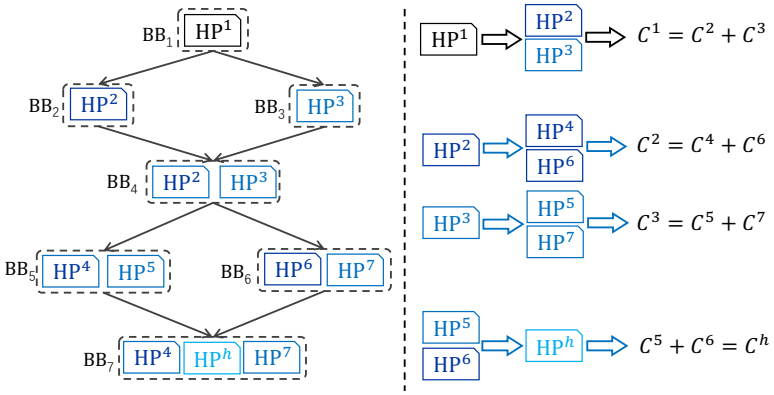



Fig. 8. An example to illustrate the constraint generation

We use the example in Figure 8 to illustrate how to compute the WCET bound with the path-sensitive cache behavior. In this example, we set the threshold S-Thres to 3. Similar to traditional approaches, the objective function for the IPET formulation is expressed as the sum of the accumulated execution time of each basic block. The accumulated execution time of each basic block is the sum of the products between the WCET analyzed by each PS-ACS and the corresponding execution counts.

$$\begin{aligned}\textbf{maximize} \quad \text{WCET} = & C^1 \cdot T_1^1 + \\ & C^2 \cdot T_2^2 + C^3 \cdot T_3^3 + \cdots + \\ & C^4 \cdot T_7^4 + C^h \cdot T_7^h + C^7 \cdot T_7^7\end{aligned}$$

The key to solving the objective function lies in establishing the constraints based on the CFG and the analyzed PS-ACSs. We first generate the constraint derived from the CFG of the target program based on the following principle: for abstract paths traversing a *Branch*, the sum of the execution counts along the outgoing branch paths equals the execution count of the original path. For the example in Figure 8, we have:

$$\begin{aligned}C^1 &= C^2 + C^3, \\ C^2 &= C^4 + C^6, \\ C^3 &= C^5 + C^7.\end{aligned}$$

For paths that traverse a *Shrink* function, we automatically generate the constraint for HPs each time the *Shrink* function is called. Specifically, when two PS-ACSs are merged into one, the execution count associated with the generated PS-ACS equals the sum of the two original PS-ACSs.

Considering the example in Figure 8, the PS-ACS corresponding to $HP^5$ and $HP^6$ exhibit the smallest R-DIFF values and are therefore selected for merging. Consequently, we obtain the relationship:

$$C^h = C^5 + C^6.$$

By integrating the aforementioned new constraints into the IPET constraints, the ILP problem for computing the WCET is refined. Finally, the WCET of the program can be derived by solving the refined ILP constraints.

### 5.4 Termination and Soundness

In this section, we prove the termination of the iterative cache behavior analysis and the soundness of the estimated WCET bound.

Lemma 5.1. *The cache behavior will eventually end with a fixed point.*

Proof. This result can be established via the Kleene fixed-point theorem:

- There are only a finite number of sets and set lines, and for each program, a finite number of memory blocks.
- Therefore, the domain of abstract cache states is finite, and every ascending chain is finite.
- The abstract cache update functions and the join functions are monotonic.

□

For the final WCET computation, the objective function of the IPET calculates the maximized sum of the accumulated execution time of all basic blocks. We establish soundness by showing that the accumulated execution time attributed to each basic block in the objective function is safe. Specifically, the per-block bounds used in the aggregation are conservative, ensuring that the resulting WCET is an upper bound on all feasible executions.

Lemma 5.2. *The accumulated execution time estimated by the proposed method safely upper-bounds the actual accumulated execution time.*

Proof. We first establish the safety of the estimated execution time for an individual basic block. The execution time of $BB_i$ can be expressed as:

$$ET_i = C_i^1 \cdot T_i^1 + C_i^2 \cdot T_i^2 + ...C_i^h \cdot T_i^h \tag{7}$$

Each element in Equation 7 corresponds to a PS-ACS in $HSP_{in}$ once the iterative computation has converged. The PS-ACSs fall into two scenarios:

- If a PS-ACS never experiences a *Shrink* operation, the analyzed cache state belongs to a concrete path. Since the *Update* function is safe, the estimated cache behavior is safe.
- If a PS-ACS experiences a *Shrink* operation, since we adopt the *Join* function from the conventional method, the estimated cache behavior is safe. Moreover, the execution count is computed by summing the execution counts of all the merged possible paths each time the *Shrink* function is called. Therefore the product is safe for computing the accumulated execution time on the corresponding paths.

Therefore, for every basic block, the estimated execution time of each path is no less than the actual execution time along the corresponding concrete path. Consequently, the sum of these estimates is guaranteed to be no smaller than the actual total execution time. Hence, the WCET computed under the above constraints is sound. □

## 6 Evaluation

In this section, we evaluate both the accuracy and efficiency of the proposed method by comparing it against the traditional abstract interpretation–based approach [3], which supports must, may, and persistent analyses. We give a comprehensive evaluation on the tightness of the estimated bound and the analysis time with different values of S-Thres. In addition, a recommendation on how to select an appropriate value for S-Thres is provided.

### 6.1 Evaluation Setup

We integrated the proposed method into the existing WCET analysis tool Chronos [19]. Our implementation [44] does not introduce new value or pointer/alias analysis. It is orthogonal to Chronos's scope-aware analysis [16] and can benefit from replacing it with other, more precise value analyses.

In the experiment, we adopt the processor architecture of the TMS320C66x DSP CPU [38], a high-performance digital signal processor from Texas Instruments whose cache configuration are summarized in Table 4:

Table 4. Cache configurations used in the experiment

| Cache Level | Capacity | Cache line | Associativity | Hit latency | Miss latency |
|---|---|---|---|---|---|
| L1 Inst | 32KB | 32B | 1 | 1 cycle | 12.5 cycles |
| L1 Data | 32KB | 64B | 2 | 1 cycle | 6 cycles |

We selected benchmark programs from two widely used embedded benchmark suites, i.e., the Mälardalen WCET Benchmark Suite [12] and MiBench [13]. The largest program has 2123 LOC, 1008 blocks, and control-flow depth 20; on average, benchmarks have 327 LOC, 190 blocks, and depth 5.3. All programs were compiled using the TI C6000-CGT compiler version 7.4.4.

### 6.2 Improvement Across Different Benchmarks

In this section, we demonstrate the accuracy and soundness of our method by comparing the estimated WCET bound with those of the existing methods and measured execution time.

Table 5. WCET Overestimation, Analysis Time, and Memory Overhead Compared to Baseline

| Bench | Baseline | S-Thres = 2 | | | S-Thres = 4 | | | S-Thres = 8 | | | S-Thres = 16 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | WCET | WCET | T. | Mem. | WCET | T. | Mem. | WCET | T. | Mem. | WCET | T. | Mem. |
| qsort | 144.73% | 85.59% | 3.30 | 3.02 | 78.42% | 10.30 | 9.96 | 70.76% | 29.57 | 29.22 | 68.67% | 72.20 | 71.86 |
| ud | 124.81% | 74.13% | 2.77 | 2.49 | 74.13% | 4.07 | 3.70 | 74.13% | 8.42 | 8.01 | 74.13% | 15.00 | 14.63 |
| insertsort | 66.32% | 40.50% | 2.83 | 2.56 | 38.10% | 4.63 | 4.31 | 35.52% | 7.63 | 7.38 | 35.52% | 12.48 | 12.13 |
| qurt | 90.22% | 61.13% | 2.21 | 1.96 | 57.13% | 4.43 | 4.10 | 56.86% | 12.08 | 11.67 | 56.86% | 25.82 | 25.39 |
| cnt | 35.05% | 22.54% | 3.15 | 2.87 | 20.44% | 6.99 | 6.62 | 17.21% | 22.15 | 21.72 | 14.91% | 47.79 | 47.46 |
| jfdctint | 116.82% | 88.34% | 1.65 | 1.43 | 84.56% | 3.73 | 3.44 | 84.56% | 7.93 | 7.58 | 84.56% | 16.85 | 16.45 |
| susan | 79.99% | 56.57% | 1.15 | 0.92 | 56.57% | 1.74 | 1.48 | 56.57% | 2.61 | 2.34 | 56.57% | 4.65 | 4.41 |
| lms | 57.83% | 47.11% | 1.80 | 1.53 | 45.99% | 2.90 | 2.63 | 45.99% | 4.80 | 4.46 | 45.99% | 9.82 | 9.49 |
| crc | 11.45% | 6.01% | 2.51 | 2.27 | 5.36% | 4.92 | 4.57 | 5.11% | 14.75 | 14.43 | 4.69% | 30.49 | 30.10 |
| basicmath | 49.96% | 43.12% | 2.01 | 1.76 | 43.07% | 4.80 | 4.51 | 42.77% | 8.80 | 8.52 | 42.77% | 14.80 | 14.42 |
| fft | 67.01% | 63.22% | 1.54 | 1.29 | 63.22% | 2.71 | 2.48 | 63.22% | 4.53 | 4.24 | 63.21% | 8.94 | 8.69 |
| prime | 14.64% | 12.96% | 1.02 | 0.83 | 12.96% | 2.04 | 1.74 | 12.96% | 4.63 | 4.27 | 12.95% | 8.07 | 7.69 |
| **Average** | 71.57% | 50.10% | 2.16 | 1.91 | 48.33% | 4.44 | 4.13 | 47.14% | 10.66 | 10.32 | 46.74% | 22.24 | 21.89 |

We report the WCET overestimation percentages by the existing method and our approach relative to the measured WCET under different threshold values S-Thres in Table 5. We have evaluated the proposed method with 4 different S-Thres, i.e., 2, 4, 8, and 16, to investigate the scalability and the accuracy of the proposed method. For simplicity, we use the value of different S-Thres to denote the experimental results produced by the proposed method.

The experimental results show that our methodology consistently generates more precise WCET bounds across all benchmark programs while upholding soundness. With a S-Thres value of 16, our method produces results that are merely 46.7% higher than the measured execution time, marking a notable precision enhancement of 24.83% compared to the existing method. Even with a S-Thres value of 2, our approach maintains a high level of accuracy, with estimates only 50.1% above the measured execution time, exhibiting a significant 21.47% improvement over the existing method.

According to the results, for benchmark tasks such as *qsort*, *ud*, *insertsort*, *qurt*, *cnt*, and *jfdctint*, our method significantly tightens the estimated WCET bounds, with improvements exceeding 40%. These benchmarks not only contain a large number of branches, but also exhibit significant differences in memory access behavior across different branches. The existing method merges different cache states from various paths into a single abstract cache state and therefore results in a pessimistic analysis result. By employing the proposed method, a substantial amount of critical cache behaviors are preserved, consequently leading to tighter WCET bounds.

For example, *qsort*, follows a divide-and-conquer structure: it partitions the input around a pivot and then recursively processes the resulting subarrays. This control-flow pattern creates highly path-dependent cache behavior, which the conventional path-insensitive analysis merges too aggressively. In contrast, our approach effectively addresses this issue by accurately tracking branch information. Similarly, in *cnt*, two tags are assigned to record the result of each comparison. As long as there are two cache misses, both tags will reside in the cache. Moreover, such shallow branching divergences can be handled efficiently by our method. However, existing methods still pessimistically assume that every access to a tag results in a cache miss. This, in turn, yields a higher estimate.

When handling tasks such as *susan* and *lms*, the gains achieved by our method are smaller compared to earlier scenarios. Nevertheless, it maintains a substantial level of precision in refining the estimates, showing an improvement of more than 10%. These benchmarks still encompass a significant number of branches. However, the memory access patterns among different branches do not exhibit substantial variations, and the cache behaviors across a certain number of execution paths remain relatively similar. In this case, recording distinct path information yields diminishing returns in reducing the WCET overestimation.

For example, in the *susan* task, the algorithm's core comprises two main components: computing gray-level variances between each pixel and its neighboring pixels, and determining edge pixels by calculating their horizontal and vertical gradients. The former, being computationally intensive, is less influenced by execution paths, while the latter involves branches with differing memory access behaviors. Consequently, our proposed method achieves a more accurate cache behavior analysis for the latter segment and a marginal enhancement for the former segment. Thus, benchmarks with this property can still benefit from the path-sensitive cache state analysis, even though the improvements may not be as pronounced as observed with the previous benchmarks.

For benchmarks such as *crc*, *prime*, *fft*, and *basicmath*, the improvement brought by our method is relatively slight, amounting to less than a 10% improvement in precision over the existing method. These benchmarks primarily focus on computations without extensive branching. For instance, the *crc* task involves iterative XOR operations between input data and a polynomial divisor to generate a checksum, thereby featuring fewer branches. As a result, our proposed path-sensitive cache behavior analysis method yields a smaller improvement in such scenarios. Even in cases

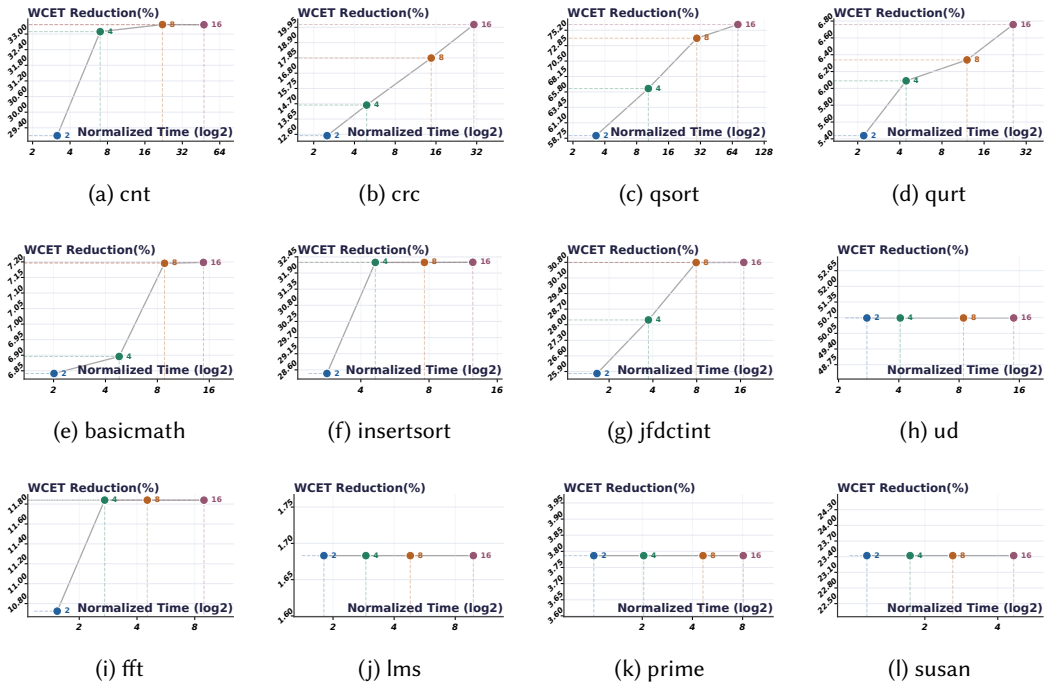



Fig. 9. Growth in analysis time as S-Thres increases.

where branches exist, they typically entail similar memory access patterns across different branches. In tasks like *fft*, the operations predominantly revolve around computations based on the Cooley-Tukey algorithm, where a butterfly operation is iteratively applied to process an array. During this process, the butterfly operation performs distinct operations on various branches but with similar memory block accesses. We expect the benefit of our approach to grow with control-flow complexity, so larger programs with more path-dependent cache divergence are likely to benefit more.

### 6.3 Sensitivity to S-Thres and Selection Guidelines

This section shows the changes in the estimated WCET bound and the analysis time with different S-Thres. The analysis time and the reduced estimated WCET bound compared with the existing method are shown in Figure 9. Note that all analysis results are normalized to the existing method.

On average, the analysis time grows approximately proportionally with increasing S-Thres. However, there are significant differences among individual benchmarks, which can be roughly categorized into three groups: those in which the analysis time increases sharply with larger S-Thres, as shown in the first row of Figure 9; those exhibiting a linear increase in analysis time, as shown in the second row of the figure; and those with only a slight increase, as shown in the third row of the figure.

For the benchmarks in which the analysis time increases sharply with larger S-Thres, such as *cnt*, *crc*, *qurt*, and *qsort*, larger S-Thres can further improve the analysis accuracy to a certain extent. For these benchmarks, the disproportionate increase in analysis time indicates that the analysis tool not only needs to maintain a large S-Thres-sized state space throughout the entire analysis, but also frequently invokes the *Shrink* function to select and merge similar PS-ACSs. This suggests

that these benchmarks have not only a large number of branches, but also significant differences in cache behaviors among them. For these benchmarks, increasing S-Thres allows more branches to be preserved, thereby improving the analysis accuracy—up to 60% at best, with an average improvement of 32%.

For those benchmarks where the analysis time increases nearly linearly with S-Thres, such as *basicmath*, *insertsort*, *jfdctint*, and *ud*, a larger S-Thres may still improve the accuracy, but the improvement is rather limited. For these benchmarks, although they have some branch structures, the differences among the branches are not significant. As a result, it is usually sufficient to record the timing information for only one or two branches to cover the different execution paths. Setting S-Thres = 4 is generally sufficient to achieve a significant improvement in analysis precision. When S-Thres is set to 16, the analysis accuracy can be further improved by an average of 9.5%; however, this comes at the cost of increasing the analysis time by up to 13 times.

For the programs in which the analysis time increases only slightly with larger S-Thres, such as *fft*, *lms*, *prime*, and *susan*, a larger S-Thres brings almost no improvement in analysis accuracy. Most of these benchmarks also exhibit either a small number of branches—with only minor benefits gained from our method—or relatively long joint paths, where the effects of different branches are largely overshadowed by the main execution path. Therefore, setting S-Thres = 2 is usually sufficient to achieve the desired benefits for these programs.

Overall, our results indicate that for the majority of benchmarks, setting S-Thres to 2—i.e., recording just one additional cache state—is adequate to achieve satisfactory results. When S-Thres is 2, the average improvement in precision achieved by our method amounts to 21.47%. However, increasing S-Thres to 16 results in only a marginal 3.36% improvement in average precision, while simultaneously increasing the average analysis time by a factor of 10.3. The above analysis shows that for most benchmark programs, using a small S-Thres is sufficient to achieve a significant improvement, demonstrating the scalability and efficiency of the proposed method.

Furthermore, the growth in analysis time can be used as an indicator of whether a larger S-Thres is necessary. For most benchmarks, analysis time grows approximately linearly with S-Thres size, since the operations required to maintain the corresponding number of cache states also scale linearly, and for benchmarks where our method offers limited benefits, the increase in runtime is actually sublinear relative to the growth in size. For programs with a moderate but not overly complex branch structure, it is worthwhile to increase S-Thres to 4 or 8. In contrast, for benchmarks where the analysis time increases disproportionately, further increasing S-Thres may be beneficial for achieving higher analysis precision.

Our results suggest that S-Thres should be selected as a power of two, since the number of distinguishable cache states may grow rapidly with successive binary branches. In practice, we recommend starting from S-Thres = 2 and increasing it by powers of two until the estimated WCET bound shows little further improvement. One may then decrease S-Thres to the smallest value that still preserves the desired accuracy-efficiency trade-off. This strategy is particularly suitable because WCET analysis is performed offline, and a larger S-Thres reduces the frequency of *Shrink* and therefore retains more path-sensitive cache states.

Overall, our approach can significantly tighten the estimated WCET bound compared with state-of-the-art methods, particularly for programs with complex control and data flows. Empirically, the estimated WCET bounds remain reasonably tight relative to measured executions: the average overestimation is 48%, and the minimum observed overestimation is 4.68%. Considering that it is nearly impossible to measure the actual WCET, the estimated WCET bound is relatively tight.

## 7 Related Work

Cache analysis for static WCET estimation is a central topic in static program analysis and has been extensively studied due to its importance for real-time systems. Mainstream approaches [9] adopt the mathematical framework of abstract interpretation, introduced by Patrick Cousot [4], to model and abstract concrete cache state spaces, and combine cache modeling with the IPET [20] to capture the dynamic influence of cache states on execution time. This integration transforms explicit path enumeration into an ILP-based formulation, markedly improving efficiency and scalability. This line of work has proven effective and has seen practical deployment [7].

Early methods relied on exact memory reference addresses [22] to classify hits and misses, and therefore targeted simple, single-level instruction caches [2, 39]. For instruction caches, memory access patterns are typically predictable, enabling highly accurate analyses. By contrast, data cache analysis is more challenging [18] because it requires identifying all memory blocks that an instruction may access [23], and better abstract modeling and improved value or alias precision can significantly tighten the resulting WCET bound.. Puasuareanu et al.[28] provide a comprehensive survey of symbolic execution techniques that systematically explore program paths using symbolic inputs to enable path-sensitive software testing and analysis. In addition, Suhendra et al.[36] propose an algorithm that detects infeasible paths via branch-assignment conflicts, while Wilhelm et al.[42] underscore the critical role of precise loop-constraint modeling. Furthermore, Schoeberl et al.[34] highlights the necessity of incorporating program constraints—such as function calls, task scheduling, and stack operations—into the analysis. Huynh [16] introduced the notion of scope within loops to enhance the precision of persistent analysis in nested-loop structures.

Although abstract interpretation primarily reduces analysis complexity by discarding temporal information of different execution paths, several studies have shown that selectively reintroducing limited temporal information can improve precision without incurring prohibitive costs. Early on, persistent analyses for loops were proposed and became a cornerstone of WCET cache analysis. Building on this, Ramaprasad's work argues that loop-affine array accesses require specialized techniques to accurately analyze their access patterns [29]. More recently, Stock [35] addressed the challenge of persistent analysis by effectively leveraging temporal information in conjunction with binary decision diagrams, and Xiao [43] achieved precise handling of previously hard-to-analyze stack data references by constructing local scopes at the function granularity. Meanwhile, a number of studies incorporate explicit path information. Touzeau et al.[40] introduce a partial order over cache states and perform reductions with respect to this order, thereby effectively bridging the gap between accuracy and efficiency in persistent analysis. Nagar's work [26, 27] focuses on identifying cache-miss paths for each memory reference to qualitatively bound the maximum number of misses. Zhang's work [46] refines persistent analysis to identify additional persistent memory references.

More broadly, disjunctive abstract interpretation and trace partitioning provide a general foundation for path-sensitive analysis. Rather than merging all executions eagerly, trace partitioning improves precision by maintaining separate abstractions for different classes of traces [25]. ESP further shows that useful path sensitivity can be achieved in a bounded and scalable way [6]. Like disjunctive analysis, we maintain multiple abstract states at program points. However, unlike general disjunctive approaches, our method specifically captures cache-state divergence within the cache lattice through a similarity-guided Shrink operator. Moreover, the path-sensitive analysis also generates refined IPET constraints, allowing the resulting path-sensitive cache behavior to be integrated with the IPET framework to derive tighter WCET bounds.

In recent years, as processor architectures have grown increasingly complex, WCET analysis has also begun to consider more sophisticated cache structures, such as different replacement policies, cache hierarchies, and even multicore shared caches. Previous work has shown that

abstract interpretation can be extended to FIFO [10], PLRU [11], and MRU [15]. Then, Damien Hardy proposed a comprehensive multi-level cache analysis accommodating a variety of cache policies [14, 15], laying a solid foundation for subsequent work. Furthermore, Reineke et al.[30–33] conducted a comprehensive series of investigations into WCET analysis under various cache replacement policies, establishing a robust theoretical foundation for extending prevailing WCET analysis frameworks to new architectural paradigms. In the multi-core context, Yan Li introduced a state-of-the-art conflict model for shared caches [21], later refined by Zhang [45] through a conflict matrix that more accurately predicts interference among program segments. Jeanmougin et al. [17] modeled the SIMT architecture of modern GPUs via a warp-level control-flow graph (CFG), extending WCET techniques from CPUs to modern GPUs. Similarly, Dagli [5] developed an analysis framework for heterogeneous SoCs, providing a theoretical basis for WCET analysis in domains such as autonomous driving and embodied intelligence. Our method is orthogonal to the specific replacement policy or cache hierarchy, as it builds on the underlying abstract-interpretation update and join operators. When a cache structure can be modeled soundly within an abstract domain and a domain-appropriate similarity metric can be defined, our framework can potentially be instantiated on top of it by defining an appropriate similarity metric over the corresponding abstract states.

## 8 Conclusion

This paper points out that the existing abstract interpretation-based cache behavior analysis fails to articulate different cache behaviors on different program paths, consequently producing pessimistic estimated WCET bounds. To solve the issue, this paper introduces a path-sensitive cache behavior analysis approach that preserves different cache behaviors originating from different program paths. To uphold the efficiency of this approach, a novel merging policy that selectively consolidates "similar" cache behaviors from a potentially exponentially large number of incoming program paths is proposed, ensuring that the preserved cache states consistently remain below a predefined threshold. The proposed path-sensitive cache behavior analysis can be seamlessly integrated with the WCET analysis framework by generating new constraints. Experimental results demonstrate that the proposed method can significantly tighten the estimated WCET bounds without sacrificing scalability.

## 9 Data Availability Statement

The artifact of this paper is publicly available at: https://doi.org/10.5281/zenodo.19045049.

## Acknowledgments

This work is supported by National Natural Science Foundation of China (Grant No.62432005, 62302270, 62372272), Shandong Provincial Natural Science Foundation (Grant ZR2024MF099).

## References

[1] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. 1996. Cache behavior prediction by abstract interpretation. In *Static Analysis: Third International Symposium, SAS'96 Aachen, Germany, September 24–26, 1996 Proceedings 3*. Springer, 52–66. doi:10.1007/3-540-61739-6_33

[2] Clément Ballabriga and Hugues Cassé. 2008. Improving the first-miss computation in set-associative instruction caches. In *2008 Euromicro Conference on Real-Time Systems*. IEEE, 341–350. doi:10.1109/ECRTS.2008.34

[3] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. 2014. A unified WCET analysis framework for multicore platforms. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014), 1–29. doi:10.1145/2584654

[4] Patrick Cousot. 1996. Abstract interpretation. *ACM Computing Surveys (CSUR)* 28, 2 (1996), 324–328. doi:10.1145/234528.234740

[5] Ismet Dagli and Mehmet E. Belviranli. 2024. Shared Memory-Contention-Aware Concurrent DNN Execution for Diversely Heterogeneous System-on-Chips. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles*

*and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 243–256. doi:10.1145/3627535.3638502

[6] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 57–68. doi:10.1145/512529.512538

[7] Christian Ferdinand and Reinhold Heckmann. 2004. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France*. Springer, 377–383. doi:10.1007/978-1-4020-8157-6_29

[8] Christian Ferdinand and Reinhard Wilhelm. 1998. On predicting data cache behavior for real-time systems. In *Languages, Compilers, and Tools for Embedded Systems: ACM SIGPLAN Workshop LCTES'98 Montreal, Canada, June 19–20, 1998 Proceedings*. Springer, 16–30. doi:10.1007/BFb0057777

[9] Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and precise cache behavior prediction for real-time systems. *Real-time systems* 17 (1999), 131–181. doi:10.1023/A:1008186323068

[10] Daniel Grund and Jan Reineke. 2009. Abstract interpretation of FIFO replacement. In *International Static Analysis Symposium*. Springer, 120–136. doi:10.1007/978-3-642-03237-0_10

[11] Daniel Grund and Jan Reineke. 2010. Toward precise PLRU cache analysis. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 23–35. doi:10.4230/OASIcs.WCET.2010.23

[12] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2010.136

[13] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 3–14. doi:10.1109/WWC.2001.990739

[14] Damien Hardy and Isabelle Puaut. 2008. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *2008 Real-Time Systems Symposium*. IEEE, 456–466. doi:10.1109/RTSS.2008.10

[15] Damien Hardy and Isabelle Puaut. 2011. WCET analysis of instruction cache hierarchies. *Journal of Systems Architecture* 57, 7 (2011), 677–694. doi:10.1016/j.sysarc.2010.08.007

[16] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. 2011. Scope-aware data cache analysis for WCET estimation. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 203–212. doi:10.1109/RTAS.2011.27

[17] Louison Jeanmougin, Thomas Carle, Pascal Sotin, and Christine Rochange. 2023. Warp-level CFG construction for GPU kernel WCET analysis. In *21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023)*, Vol. 114. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 1–1. doi:10.4230/OASIcs.WCET.2023.1

[18] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. 2009. WCET analysis of multi-level set-associative data caches. In *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)(2009)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2009.2283

[19] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2007. Chronos: A timing analyzer for embedded software. *Science of Computer Programming* 69, 1-3 (2007), 56–67. doi:10.1016/j.scico.2007.01.014

[20] Y-TS Li, Sharad Malik, and Andrew Wolfe. 1995. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings 16th IEEE Real-Time Systems Symposium*. IEEE, 298–307. doi:10.1109/REAL.1995.495219

[21] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. 2012. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems* 48 (2012), 638–680. doi:10.1007/s11241-012-9160-2

[22] Thomas Lundqvist and Per Stenström. 1999. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems* 17 (1999), 183–207. doi:10.1023/A:1008138407139

[23] Thomas Lundqvist and Per Stenstrom. 1999. A method to improve the estimated worst-case performance of data caching. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No. PR00306)*. IEEE, 255–262. doi:10.1109/RTCSA.1999.811244

[24] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. 2016. A Survey on Static Cache Analysis for Real-Time Systems. *Leibniz Transactions on Embedded Systems* 3, 1 (2016), 05:1–05:48. doi:10.4230/LITES-v003-i001-a005

[25] Laurent Mauborgne and Xavier Rival. 2005. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming*. Springer, 5–20. doi:10.1007/978-3-540-31987-0_2

[26] Kartik Nagar and YN Srikant. 2015. Path sensitive cache analysis using cache miss paths. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 43–60. doi:10.1007/978-3-662-46081-8_3

[27] Kartik Nagar and YN Srikant. 2017. Refining cache behavior prediction using cache miss paths. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 4 (2017), 1–26. doi:10.1145/3035541

[28] Corina S Păsăreanu and Willem Visser. 2009. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer* 11 (2009), 339–353. doi:10.1007/s10009-009-0118-1

[29] Harini Ramaprasad and Frank Mueller. 2005. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *11th ieee real time and embedded technology and applications symposium*. IEEE, 148–157. doi:10.1109/RTAS.2005.12

[30] Jan Reineke and Daniel Grund. 2008. Relative Competitive Analysis of Cache Replacement Policies. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Tucson, AZ, USA). ACM, New York, NY, USA, 51–60. doi:10.1145/1375657.1375665

[31] Jan Reineke and Daniel Grund. 2008. Relative Competitiveness of Cache Replacement Policies. In *SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Annapolis, MD, USA). ACM, New York, NY, USA, 431–432. doi:10.1145/1375457.1375506

[32] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. 2006. *Predictability of Cache Replacement Policies*. Reports of SFB/TR 14 AVACS 9. SFB/TR 14 AVACS. ISSN: 1860-9821.

[33] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. 2007. Timing predictability of cache replacement policies. *Real-Time Systems* 37 (2007), 99–122. doi:10.1007/s11241-007-9032-3

[34] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, and Christian W Probst. 2011. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems (2011)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. doi:10.4230/OASIcs.PPES.2011.11

[35] Gregory Stock, Sebastian Hahn, and Jan Reineke. 2019. Cache persistence analysis: Finally exact. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 481–494. doi:10.1109/RTSS46320.2019.00049

[36] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. 2006. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the 43rd annual Design Automation Conference*. 358–363. doi:10.1145/1146909.1147002

[37] Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. (1955). doi:10.2140/pjm.1955.5.285

[38] Texas Instruments. 2010. *C66x CPU and Instruction Set Reference Guide*. Texas Instruments. https://www.ti.com/lit/pdf/sprugh7 No. SPRUGH7.

[39] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. 2000. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems* 18 (2000), 157–179. doi:10.1023/A:1008141130870

[40] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2019. Fast and exact analysis for LRU caches. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29. doi:10.1145/3290367

[41] Reinhard Wilhelm. 2004. Why AI+ ILP is good for WCET, but MC is not, nor ILP alone. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 309–322. doi:10.1007/978-3-540-24622-0_25

[42] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 1–53. doi:10.1145/1347375.1347389

[43] Shangshang Xiao, Mengxia Sun, Wei Zhang, Naijun Zhan, and Lei Ju. 2024. Cache Behavior Analysis with SP-Relative Addressing for WCET Estimation. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 256–274. doi:10.1007/978-981-96-0602-3_14

[44] Shangshang Xiao, Mengxia Sun, Wei Zhang, Naijun Zhan, and Lei Ju. 2026. Artifact for Path-Sensitive Abstract Interpretation for WCET Estimation. doi:10.5281/zenodo.19045049

[45] Wei Zhang, Mingsong Lv, Wanli Chang, and Lei Ju. 2022. Precise and scalable shared cache contention analysis for WCET estimation. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1267–1272. doi:10.1145/3489517.3530613

[46] Zhenkai Zhang and Xenofon Koutsoukos. 2015. Improving the precision of abstract interpretation based cache persistence analysis. *ACM SIGPLAN Notices* 50, 5 (2015), 1–10. doi:10.1145/2808704.2754967

Received 2025-11-07; accepted 2026-04-03