Cache Behavior Analysis with SP-Relative Addressing for WCET Estimation

Shangshang Xiao[1,2], Mengxia Sun[1], Wei Zhang[1,2(✉)], Naijun Zhan[3], and Lei Ju[2]

[1] School of Cyber Science and Technology, Shandong University, Qingdao, China
[2] Quan Cheng Laboratory, Jinan, China
sduzhangwei@sdu.edu.cn
[3] School of Computer Science, Peking University, Beijing, China

Abstract. Accurately bounding the worst-case execution time (WCET) is crucial for efficient real-time system design. Precisely analyzing whether a memory reference results in a cache miss or a cache hit significantly impacts the accuracy of the estimated WCET bound, as the latency of a cache miss is typically two orders of magnitude higher than that of a cache hit access. SP-relative addressing enables dynamic memory allocation on the stack, enhances thread safety, an therefore is widely adopted in embedded real-time systems. However, existing cache behavior analysis requires obtaining the exact address of each memory reference and pessimistically assumes that SP-relative addressing could access any memory address, resulting in an overestimated WCET bound. In this paper, we propose a new WCET analysis framework that first identifies the program segments in which the SP value remains constant for a accurate cache behavior analysis, and then comprehensively compute the WCET of the whole program based on the constraints between the SP value defined by the program's control flow. Experimental results show that, our method can tighten the estimated WCET bound by up to 36.31% and achieve an average improvement of 11.85%.

Keywords: WCET · Abstract interpretation · Cache analysis · SP-relative addressing

1 Introduction

Real-time systems [1,14] are extensively utilized in safety-critical fields such as autonomous driving, medical devices, and nuclear power plants, where tasks must operate within strict time constraints to guarantee reliability and safety. Therefore, bounding the worst-case execution time (WCET) during compile time is essential for the real-time verification. To ensure secure and precise timing

S. Xiao and M. Sun—The first two authors are Contributed equally to this work.

T. Bourke et al. (Eds.): SETTA 2024, LNCS 15469, pp. 256–274, 2025.
https://doi.org/10.1007/978-981-96-0602-3_14

verification, the estimated WCET bounds must be both tight (as close as possible to the actual worst-case execution time) and sound (no lower than any possible execution time).

Cache behavior analysis is crucial for accurate WCET estimation, given that the latency of a cache miss is typically two orders of magnitude greater than that of a cache hit. Current methods [3,10] often rely on abstract interpretation-based cache behavior analysis to predict the cache behavior of memory references, specifically assessing whether a memory block is evicted from the cache before its next access.

The memory address accessed by SP-relative addressing deepens on the SP register's value, which undergoes frequent changes during runtime, especially across function transitions. Moreover, the value of SP is path-dependent, which makes obtaining the exact value of it at compile time computationally challenging. Conventional methods struggle to precisely analyze the SP register values, and have to pessimistically assume that SP-relative addressing instructions might access any stack address, resulting in a substantial overestimation of cache interference.

Despite the unpredictability of the SP value at compile time, we observe that within specific program segments, such as loops without function calls, the SP value remains constant. In scenarios where the SP value in unchanged, even if its exact value is unknown, a more accurate cache interference analysis can be conducted based on the relative offsets among different SP-relative addressing instructions. To facilitate this, we introduce a novel analytical unit, called Hyper-Block, to identify program segments where the SP value remains consistent. By leveraging existing WCET analysis techniques, we can precisely calculate a tight WCET for each Hyper-Block and the corresponding SP value.

However, the change in the SP values between different Hyper-Blocks follows the control flow of the program, and the individually analyzed WCET bound for each Hyper-Block may result in an overestimated WCET bound. For example, the SP value in a caller Hyper-Block is never lower than that in the callee Hyper-Block. We formally model such constraints on SP values across distinct Hyper-Blocks and formulate the program's WCET as an Integer Linear Programming (ILP) problem to accurately compute the overall program's WCET. Experimental results conducted with the widely used MRTC benchmark program reveal that our method can tighten the estimated WCET bound by up to 36.31% and achieve an average improvement of 11.85%.

The subsequent sections of this paper are organized as follows: Sect. 2 delves into the related work, Sect. 3 provides an overview of cache analysis, Sect. 4 outlines the motivation behind this study, Sect. 5 introduces our proposed approach in detail, Sect. 6 evaluates the effectiveness of the proposed approach, and finally, Sect. 7 presents the concluding remarks for this paper.

2 Related Work

Research on cache analysis for WCET computation has been investigated for several decades. Ferdinand and Wilhelm proposed the abstract interpretation-

based technique [5] to statically predict the cache behavior of programs. At present, abstract interpretation method has been successfully applied to instruction cache analysis [2,23], data cache analysis [11] and multi-level caches analysis [8] of WCET estimation, and becomes a dominating approach for cache analysis with LRU replacement policy. Building on this cache analysis technique, Huynh et al. introduced a scope-aware persistence analysis [10] to capture the dynamic behavior of memory access, Zhang et al. further applied it to simulate dynamic cache behavior in preemptive multi-tasking systems [27], several works [17,26,28] considered the effect of interference from multi-core processors on cache analysis.

Existing cache behavior analysis methods generally require the exact address of each memory reference [15] to determine if it will result in a cache hit or miss. For instruction caches, executing an instruction leads to access to a concrete memory block. Since program access patterns to instructions are regular and predictable, current techniques can analyze instruction caches with high accuracy. However, data access patterns are significantly more complex [11]. To analyze data caches, it is crucial to determine all possible memory blocks that each instruction might access during execution [16]. This task is challenging because static data references often require extending invariant registers [18], which are generated by instructions that can be moved out of loops. For loop-affine array accesses, several methods are proposed [10,19] to analyze array access patterns for data cache analysis.

Different from the loop-affine array accesses, the cache behavior analysis for SP-relative addressing is more complicated. R. T. White et al. [24] have noted that for stack data, the location of local variables for each function call can be determined accurately using address calculators and compiler analysis. By analyzing the program's control and data flow, the compiler and address calculator can predict stack data behavior even with multiple function calls and recursion. However, applying this approach to WCET calculations can lead to high computational loads and path explosion problems. Existing method [4] assumes that a SP-relative addressing may access any stack address and therefore leads to pessimistic cache interference analysis result, and WCET bound.

After obtaining cache behavior, integrating program constraints further enhances the accuracy of WCET predictions and helps address issues such as path explosion. Li, Malik, and Wolfe [13] proposed a method that combines cache models with Implicit Path Enumeration Technique (IPET) to simulate the impact of cache behavior on program execution time. This method considers dynamic changes in cache states, such as whether the cache contains the required data when accessing a specific memory address. Wilhelm et al. [25] focused on the precise modeling of loop constraints. Vivi et al. [21] proposed an infeasible path detection algorithm that finds pairwise conflicts between branches and assignments. Schoeberl, Brandner, and Puffitsch [20] discussed the necessity of considering specific program constraints in WCET analysis, such as function calls, task scheduling, and stack operations. Therefore, by designing functional constraints specific to a particular analysis method, combined with detailed information on the behaviour of the program, it is possible to exclude infeasible paths and avoid unreliable WCET estimates.

3 Background

This sections briefly presents the background knowledge of existing Abstract Interpretation-based WCET analysis method.

3.1 Control Flow Analysis

```
0000000000400604 <my_fabs>
  400604:   d10043ff    sub sp, sp, #0x10
  400608:   bd000fe0    str s0, [sp, #12]
  ...// Some assembly code omitted here
  400614:   54000044    b.mi  40061c <my_fabs+0x18>
  400618:   14000004    b   400628 <my_fabs+0x24>
  40061c:   bd400fe0    ldr s0, [sp, #12]
  400620:   1e214000    fneg    s0, s0
  400624:   14000002    b   40062c <my_fabs+0x28>
  400628:   bd400fe0    ldr s0, [sp, #12]
  40062c:   910043ff    add sp, sp, #0x10
  400630:   d65f03c0    ret

0000000000400634 <main>
  400634:   a9bd7bfd    stp x29, x30, [sp, #-48]!
  400638:   910003fd    mov x29, sp
  40063c:   90000000    adrp    x0, 400000 <__abi_tag-0x278>
  ...// Some assembly code omitted here
  400660:   1400000a    b   400688 <main+0x54>
  400664:   b9802fe0    ldrsw   x0, [sp, #44]
  400674:   97ffffe4    bl  400604 <my_fabs>
  ...// Some assembly code omitted here
  400688:   b9402fe0    ldr w0, [sp, #44]
  40068c:   7100101f    cmp w0, #0x4
  400690:   54fffead    b.le    400664 <main+0x30>
  400694:   52800000    mov w0, #0x0
  400698:   a8c37bfd    ldp x29, x30, [sp], #48
  40069c:   d65f03c0    ret
```

Listing 1. Assembly Code of the Program fabs

The worst-case execution Time (WCET) analysis is conducted utilizing the Control Flow Graph (CFG) of each program. The CFG is generally constructed based on the binary code of each program rather than the high-level languages to mitigate the effects of compiler optimizations. In specific, it is created through the decomposition of the program into a series of basic blocks, in which instructions must be executed sequentially without branch. The constructed basic blocks are subsequently linked based on their sequential execution order, forming the CFG. An illustrative example of a CFG is presented in Fig. 1. We first construct basic blocks by identifying the branch instructions in the disassembled code in Listing 1. Function my_fabs has three branch instructions: line4 points to line6, line5 points to line9, and line8 points to line10. Therefore the function are divided by these branch instructions into 4 basic blocks, and the CFG of my_fabs is constructed as the gray parts shown in Fig. 1. Similarly, by identifying all the branch instructions including function call and return instructions, the CFG of the program is constructed as shown in Fig. 1.

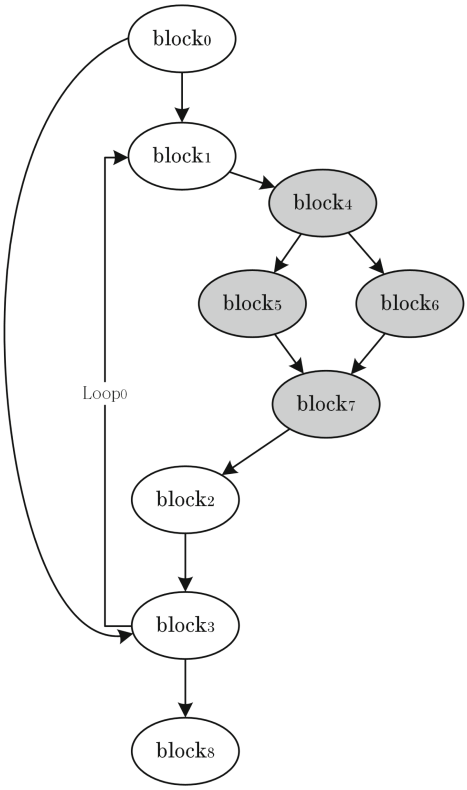

Fig. 1. CFG of the Program of Listing 1

3.2 Abstract Interpretation Based Cache Behavior Analysis

Based on the CFG, the analysis of cache behavior is conducted. Whether a memory reference results in a cache hit or miss significantly depends on the basic blocks (i.e., paths) the program traverses. To avoid exploring all possible program paths, existing methods typically employ abstract interpretation to statically analyze cache behavior. In essence, the abstract interpretation method merges concrete cache states from various paths into a singular abstract cache state, unveiling the worst-case scenario. The cache analysis rooted in abstract interpretation generally comprises three components: the Must analysis, which keeps only the memory blocks that are present in the cache on every path and maintains the maximum age of the accessed memory block, to predict whether a memory block is always in the cache; the May analysis, which retains all memory blocks across distinct paths and maintains the minimum age of the accessed memory block, to predict whether a memory block is likely to be in the cache; and the Persistence analysis, which maintains the maximum age of the accessed memory block and retains memory blocks that appear on at least one path, to predict whether a memory block is persistent. Then all the memory references are classified into four categories, as described in Table 1.

Table 1. Classification of Cache Access Behaviors

Classification	Cache Access Behaviors Described	Analysis
AH (Always Hit)	Every time the memory block is accessed, it is guaranteed to hit in the cache	Must
AM (Always Miss)	Every time the memory block is accessed, it is guaranteed to miss in the cache	May
PS (Persistent)	The first time the memory block is accessed, it misses; then all subsequent accesses hit in the cache	Persistent
NC (No classified)	The memory access cannot be classified into any of the above three categories	/

In the following we take the Must analysis as an example to illustrate the abstract interpretation based method. Firstly, three abstract cache states are defined for each basic block:

- **$\mathbf{ACS_{in}}$**: It represents the abstract state (ACS) of the base block before execution begins. It is obtained by merging the post-execution abstract state (ACS_{out}) of all the predecessor nodes of the basic block. This merge operation requires the Join function of the Must analysis method. If no predecessor node exists, the ACS_{in} is set to none.
- **Gen**: It includes the memory blocks accessed by the basic block.
- **$\mathbf{ACS_{out}}$**: It represents the abstract state of the base block after performing a series of memory access operations. First it needs to get access to the memory block through the Gen function, and update based on ACS_{in} status, which needs to apply the Update function of the Must analysis method.

Then, To describe the changes in cache state during program execution, two functions are defined over the abstract cache state to describe the changes in cache state during the program's execution.

- **Update**: The Update function for Must analysis is used to represent changes of the memory access over the ACS. Similar to actual cache replacement policies, each time a new memory access occurs, it is placed in the most "recent" position. The remaining memory blocks in the cache then follow the LRU (Least Recently Used) strategy.
- **Join**: The Join function is used to combined the abstract cache states of multiple basic blocks. It is similar to set intersection, and always has an upper bound of the position of the memory block. A memory block only stays in the abstract set state, if it is in both operand abstract set states. It gets the oldest age, if it has two different ages. For program in Listing 1, in the $block_6$, the memory block at address `"sp+12"` and `"0x400028"` is accessed, while in the $block_5$, only `"0x400028"` is accessed. Therefore in the $block_7$, the memory block at address `"sp+12"` will not left after the Join function while `"0x400028"` does.

Because programs contain loops or recursive structures, this process requires multiple iterations to explore all possible execution paths. Each iteration updates

the abstract state based on the program's control flow and data flow until the abstract state stabilizes. The final ACS are obtained, the memory blocks that reside in each basic block $\mathrm{ACS}_{\mathrm{in}}$ are treated as AH, which means that by the time execution reaches this basic block, the cache already contains these memory blocks and they will be hit without having to be reloaded. The analysis process of the other methods is similar to the Must method, but they differ in the Join function.

3.3 Implicit Path Enumeration

With the predicted cache behaviors of each memory reference and the pipeline architecture of the CPU, the WCET of each basic block can be computed. Then Implicit Path Enumeration Technique (IPET) is used for calculating the WCET of the program. Computing the WCET of the program is essentially equals to find the longest path within its CFG. IPET models the problem of finding the longest path as an Integer Linear Programming (ILP) problem.

We take the example in Fig. 1 to illustrate IPET. The objective function can be expressed as:

$$\mathrm{WCET} = \mathrm{MAX} \sum_{i=1}^{N} T_i \cdot X_i \tag{1}$$

where N is the count of basic blocks of the program, T_i is the WCET of the basic block i, and X_i is the execution count of the basic block i. The constraints is derived from the CFG of the program. For any basic block, its execution count is equal to the sum of the execution counts of all predecessor blocks, and also equal to the sum of the execution counts of all successor blocks:

$$X_i = \sum_{j \in \text{ all predecessors}} X_j = \sum_{k \in \text{ all successors}} X_k \tag{2}$$

For example, the execution count of block_7 (denoted as X_7) is the sum of X_5 and X_6, and it is also equal to X_2. The execution count of block_1 (represented as X_1) is the sum of X_0 and the loop bound of L_0, and equal to X_4, we denote the Loop bound of the loop i by L_i. Some of the constraints of the example program are expressed as followed:

$$\begin{aligned} X_7 - X_5 - X_6 &= 0 \\ X_7 - X_2 &= 0 \\ X_1 - X_0 - L_0 &= 0 \\ X_1 - X_4 &= 0 \end{aligned}$$

4 Motivation

For data cache analysis, executing a single instruction can involve accessing multiple memory blocks. Therefore, before performing data cache analysis, it is essential to pre-process and determine the set of memory blocks that each

instruction might access. A significant portion of these memory accesses are related to stack operations.

At runtime, global variables, static variables, and constants are stored in the global static area, allocated and freed by the system. The addresses of global variables are typically obtainable through static analysis, as they are determined at compile time and can be retrieved from the symbol table. In contrast, local variables and function parameters are stored in the stack area, with addresses dynamically allocated and released by the compiler based on the stack layout. This dynamic nature makes it challenging to precisely determine the addresses of local variables through static analysis.

Stack frame management is closely tied to the stack pointer, a register used to point to the current top of the stack, e.g., SP register in AArch64, %rsp and %rbp register in x86-64, and the B15 register in C66x DSPs. In this paper, we uniformly refer to it as the SP. SP-relative addressing is a method for accessing memory based on an offset from the SP register. This technique is commonly used to access local variables, function parameters, or return addresses on the stack. Since SP points to the top of the stack, SP-relative addressing allows for convenient reading or writing of data within the stack. For example, local variables can be accessed by adding an offset to SP, such as `"mov r0, [sp, #-4]"`. Additionally, the value of SP can be manipulated to manage stack space. For instance, the instruction `"sub sp, sp, #8"` decrements SP by 8, allocating extra stack space for local variables. Conversely, `"add sp, sp, #8"` increments SP by 8, releasing the previously allocated stack space. These operations ensure proper positioning and management of local variables during function calls. SP-relative addressing simplifies and enhances the efficiency of managing function calls and returns, as adjusting SP is sufficient to handle new local variables and parameters.

Although providing advantages in memory management, SP-relative addressing poses challenges in WCET estimation, as **the value of the stack pointer (SP) register is typically not available at compile time.** The SP register dynamically changes as the program executes, its exact value is often dependent on the current state of the program stack during runtime, making it challenging to predict or determine beforehand at compile time. Especially in case of recursive or nested function calls, the stack pointer varies with each call, creating multiple layers of stack frames. Static analysis tools struggle to accurately predict SP positions across different function call frames without runtime information, making it difficult to determine the potential data blocks accessed by these instructions.

We show an example in Listing 1 to show the pessimism of the existing method. This example includes the *main* function and the *my_fabs* function, where the *main* function is responsible for initializing variables and call *my_fabs* to calculate the absolute value of a float.

Upon entering *main*, the stack pointer (SP) references the current stack top. The initial instruction in *main*, `"stp x29, x30, [sp, #-48]!"`, stores x29 (the frame pointer) and x30 (the return address) onto the stack, moving the SP down-

wards by 48 bytes to reserve space for the new stack frame. Subsequently, the program transitions to *my_fabs*, causing the SP to descend once more to establish the stack frame for *my_fabs* (i.e., `"sub sp, sp, #0x10"`). In this example, the instructions `"ldr s0, [sp, #12]"` (line 7) and `"ldr w0, [sp, #44]"` (line 23) both reference memory addresses relative to the SP value. However, the SP values at the time of executing these instructions differ.

The stack pointer's value is path-dependent, making precise SP value determination typically needs to enumerate all possible execution paths, which is computational intractable. Existing methods [10,11] adopt a pessimistic approach by assuming that SP-relative addressing operations could potentially access any stack addresses, and therefore may consider that SP-relative addressing might contend with each other in the cache, resulting in a pessimistic analysis result.

5 Methodology

This section introduces a novel analysis framework that leverages a novel structure termed Hyper-Block to estimate the worst-case execution time. Conventional methods [10,11,28] struggle to accurately forecast SP-relative addressing, leading to a pessimistic estimation of the WCET bound. The proposed approach initially identifies the program segment where the SP address remains constant, termed Hyper-Block, aiming for a more precise WCET bound within each Hyper-Block, as detailed in Sect. 5.1. Subsequently, a novel technique is introduced to conduct a precise WCET analysis for the whole program by accurately determining the offset on SP values of different Hyper-Blocks, based on the call hierarchy among various procedures, as shown in Sect. 5.2.

5.1 Hyper-Block

Existing WCET analysis frameworks encounter challenges in accurately evaluating the stack pointer (SP) value, and have to assume a worst-case scenario where each SP-relative addressing operation could potentially access any stack address, leading to an estimated WCET bound, as discussed in Sect. 4. However, the SP value does not change randomly. Changes in the SP value generally occur during function calls or returns, remaining constant over prolonged duration. In cases where the SP value remains unchanged, despite its specific value being unknown, the offset concerning memory addresses accessed by SP-relative instructions can be statically determined. This capability facilitates a more precise cache behavior analysis.

Our analysis relies on the Global-Control Flow Graph (G-CFG) of the program, constructed from the Control Flow Graph (CFG) of each function and their inter-function call relationships as shown in Fig. 2. More specifically, we establish an G-CFG by analyzing the caller-callee relationships between functions using assembly code, which avoiding the impact of compiler optimizations. We establish connections between the basic block containing the function's call instruction and the entry basic block of the called function. Similarly, the exit

basic block of the called function is linked to the return block in the calling function. This caller-callee relationship can typically be directly obtained through the jump instructions in the assembly code. However, for function pointers and indirect function calls where the target function cannot be determined explicitly, we adopt symbolic execution to identify all the potential functions they point to. In addition, we accept the user input to identify the caller-callee relationship for building a more accurate G-CFG. In order to perform a precise cache behavior analysis for SP-relative addressing memory references, we first identify program fragments where the SP value in constant, dented by Hyper-Block. A Hyper-Block is a sub-graph of the G-CFG, which contains multiple basic blocks that share the same function stack. In order to obtain a control flow between different Hyper-Blocks without violating the program structure, we construct Hyper-Block based on the G-CFG by the following steps:

1. from the inner-most loop to the outer loop of the G-CFG, group neighbouring basic blocks that reside in the same loop and same function as a Hyper-Block.
2. if all the basic blocks in a loop are grouped as a Hyper-Block, the Hyper-Block is considered as a basic block for the following Hyper-Block construction; otherwise, if the loop is divide into several Hyper-Blocks, the Hyper-Blocks are finally obtained.
3. iteratively perform the above steps until a fixed-point.

The algorithm for building Hyper-Blocks is detailed in Algorithm 1.

Algorithm 1: Hyper-Block construct

```
1: for each basic block i does not belong to any HyperBlock do
2:    Initialize HyperBlock_i for i and Add HyperBlock_i to HyperBlocks
3:    for each successor j of i do
4:       if j in same procudure AND no other path between i AND in same loop
         level then
5:          Add j to HyperBlock_i.blocks
6:       end if
7:    end for
8: end for
9: for each HyperBlock hb do
10:   for each head block b_h and tail block in b_t in hb do
11:      Initialize dummy_in AND dummy_out
12:      dummy_in.out = b.in AND dummy_in.in = b.out
13:   end for
14: end for
```

Figure 2 shows a typical example of how to construct the Hyper-Block. In the example, basic blocks with the same color belongs to the same function. Following the instructions, we firstly construct 6 Hyper-Blocks, {block_1, block_2, block_3}, {block_6, block_7, block_8, block_9}, {block_{12}, block_{13}, block_{14}}, {block_{10}},

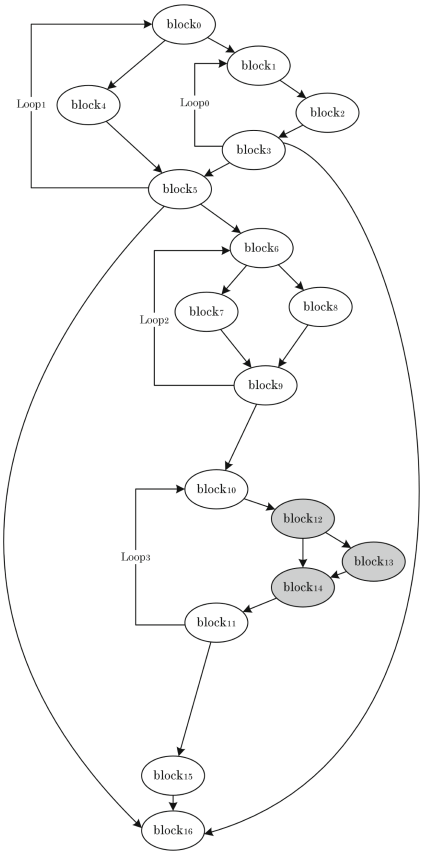

Fig. 2. G-CFG of the Program

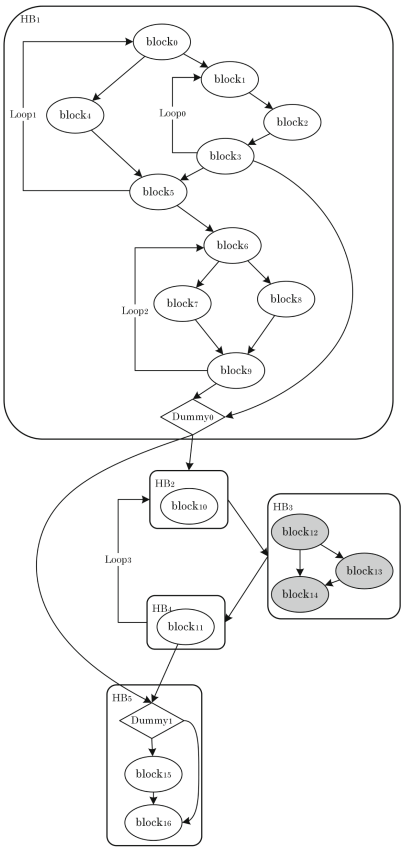

Fig. 3. Control flow constructed by Hyper-Block

{$block_{11}$}, and {$block_{15}$, $block_{16}$}. Since all the basic blocks in loop0 and loop2 are all constructed as a Hyper-Block, respectively, {$block_1$, $block_2$, $block_3$} and {$block_6$, $block_7$, $block_8$, $block_9$} are considered a basic block and further to construct a new Hyper-Block, i.e., HB_1.

After dividing the G-CFG into several Hyper-Blocks, each Hyper-Block can be constructed as a smaller CFG to facilitate further analysis. If irregular branch structure occurs between Hyper-Blocks, special block called dummy node is required to add in the Hyper-Block. In the example of Fig. 3, $block_3$ and $block_4$ in the Hyper-Block1 both connect to $block_7$ in Hyper-Block5. To build a CFG with Hyper-Blocks, we introduce dummy node to create a dedicated entry and exit nodes of each Hyper-Block. If there are multiple exit blocks, a dummy node is created, and all outgoing edges from the exit nodes are connected to this node. This dummy node is then connected to the entry nodes of other Hyper-Block, with the edge types consistent with those of the original outgoing edges. Similarly,

if there are multiple entry blocks, a dummy node will be created. This makes the dummy node the only entry node, and it connects to the exit nodes of other Hyper-Block, maintaining consistent edge types. Therefore, redundant edges of the same type between two Hyper-Block should be removed. For example, if there are two *nottaken* type edges between Hyper-Block 1 and Hyper-Block 3, only one should be retained.

5.2 WCET Computation

Once all the Hyper-Block are constructed, all the SP-relative addressing instructions inside a Hyper-Block share the same SP value. Then, we can simply and safely compute the WCET of each Hyper-Block with the existing WCET analysis method by enumerating the SP value from 0 to $set - 1$, where set denote the counts of cache set. Note that, whether two memory references cause cache contentions depends on the cache set they mapped rather than the exact memory address they accessed. Therefore, it is unnecessary to perform enumeration across the entire memory address space, while simply enumerating the SP value from 0 to $set - 1$ can significantly reduce analysis overhead. Moreover, to accurately determine whether two SP-relative addressing are mapped to the same cache set, we further enumerate such addresses by calculating the specific cache set of it in different SP value. Since step is carried out at the cache set granularity, its overhead is acceptable. Such enumeration yields a sound WCET estimation for each Hyper-Block. The proof is straightforward and is thus omitted.

Moreover, our method only leads to a linearly increased computational complexity, which is acceptable. Finally, we can generate a table that correlates various SP values with their respective estimated WCET bounds. We define function $\mathrm{WCET}(\mathrm{HB_i}, \mathrm{SP_j})$ to return the WCET of Hyper-Block $\mathrm{HB_i}$, when the SP equals $\mathrm{SP_j}$. Compared to Transformed-Control Flow Graph (T-CFG), which unfolds all the function calls and struggles to represent recursions, the G-CFG we use is essentially a function call graph, making it well-suited to represent recursive functions. We compute the WCET of each instance of a recursion by enumerating as mentioned above. Although different instances have different SP value, since we always seek the worst-case SP value across all instances, this approach remains safe in WCET estimation.

Although the WCET of each Hyper-Block can be computed by enumerating all the possible SP values, simply adding them together would yield a conservative estimation of the WCET, as the SP values that result the WCET of different Hyper-Block may not be simultaneously satisfied. For instance, for Hyper-Blocks that belongs to the same function, their SP value are same. So if the estimated WCET of these Hyper-Blocks corresponding to different SP values, the combination of the SP value are invalided, resulting in a overestimated WCET bound.

For the example shown in Fig. 3, we divide the G-CFG of the example into 5 Hyper-Blocks: $\mathrm{HB_1}$, $\mathrm{HB_2}$, $\mathrm{HB_3}$, $\mathrm{HB_4}$ and $\mathrm{HB_5}$. We assume that the longest path of the program is:

$$\mathrm{HB_1} \mapsto \mathrm{HB_2} \mapsto \mathrm{HB_3} \mapsto \mathrm{HB_4} \mapsto \mathrm{HB_5}$$

and the WCET of each Hyper-Block and the effected SP values are:

$$\mathrm{WCET}(\mathrm{HB}_1, 0), \mathrm{WCET}(\mathrm{HB}_2, 0), \mathrm{WCET}(\mathrm{HB}_3,\ 2)$$

$$\mathrm{WCET}(\mathrm{HB}_4, 1), \mathrm{WCET}(\mathrm{HB}_5,\ 0)$$

Since HB_1, HB_2, HB_4, HB_5 belong to the same function, their SP values should be same. However, the computed WCET of these corresponds to different SP values, i.e., 0, 0, 2, 0, respectively. So the combination of the SP values and the WCET are infeasible.

In order to safely compute the WCET of the program and avoid involving invalid combinations of SP values from different Hyper-Blocks, we formulate the constraints between SP values based on the control flow among Hyper-Blocks. Firstly, if Hyper-Block belong to the same function, they share the same stack and their stack pointers should be identical. Therefore, if $\mathrm{HB_i}$ and $\mathrm{HB_j}$ belong to the same function, we obtain the relationship:

$$\mathrm{SP}_{\mathrm{HB_i}} = \mathrm{SP}_{\mathrm{HB_j}} \tag{3}$$

Additionally, during task transitions, the SP value changes. In specific, the SP values increased when entering a new function for saving the state of the currently executing task (including the stack pointer), and decreases when returning to the caller function for restoring the state of the next task. Therefore, equations are established to relate the SP value between the caller Hyper-Block HB_i and the called Hyper-Block HB_j, as well as between the called Hyper-Block HB_m and the Hyper-BlockHB_n it returns to:

$$\mathrm{address}(\mathrm{SP}_{\mathrm{HB_i}}) + \mathrm{Size}_{\mathrm{HB_j^i}} = \mathrm{address}(\mathrm{SP}_{\mathrm{HB_j}}) \tag{4}$$

$$\mathrm{address}(\mathrm{SP}_{\mathrm{HB_m}}) - \mathrm{Size}_{\mathrm{HB_n^m}} = \mathrm{address}(\mathrm{SP}_{\mathrm{HB_n}}) \tag{5}$$

where $\mathrm{Size}_{\mathrm{HB_j^i}}$ denotes the change on SP value during Hyper-Block transition from $\mathrm{HB_i}$ to $\mathrm{HB_j}$, $\mathrm{address}(\mathrm{SP}_{\mathrm{HB_i}})$ denotes the exact value of the SP in $\mathrm{HB_i}$.

Since we only care the cache set that a SP-relative addressing mapped to, rather than the exact memory address. The above relationships can be transferred into the following SP constraint:

$$\mathrm{abs}(\mathrm{SP}_{\mathrm{HB_i}} - \mathrm{SP}_{\mathrm{HB_j}})\%set = \frac{\mathrm{Size}_{\mathrm{HB_j^i}}}{\mathrm{Size}_{\mathrm{line}}}\%set \tag{6}$$

$$\mathrm{abs}(\mathrm{SP}_{\mathrm{HB_m}} - \mathrm{SP}_{\mathrm{HB_n}})\%set = \frac{\mathrm{Size}_{\mathrm{HB_n^m}}}{\mathrm{Size}_{\mathrm{line}}}\%set \tag{7}$$

where $\mathrm{Size}_{\mathrm{line}}$ returns the cache line size.

With the above analysis on the SP values, we can derive the SP constraints and combine them with the program IPET constraints (i.e., as shown in Sect. 3.3). For the example in Fig. 3 the WCET can be formulated as:

$$\mathrm{Maximize} = \mathbf{X}(\mathrm{HB_i}) * \mathrm{WCET}(\mathrm{HB_i}, \mathrm{SP_i}) \tag{8}$$

Subject to:

ipet constraints :

$$X(\mathrm{HB}_1) - X(\mathrm{HB}_2) - X(\mathrm{HB}_5) = 0$$
$$X(\mathrm{HB}_2) - X(\mathrm{HB}_1) - X(\mathrm{Loop}_3) = 0$$
$$X(\mathrm{HB}_5) - X(\mathrm{HB}_4) - X(\mathrm{HB}_1) = 0$$
$$X(\mathrm{HB}_2) - X(\mathrm{HB}_3) = 0$$
$$X(\mathrm{HB}_3) - X(\mathrm{HB}_4) = 0$$

SP constraints :

$$\mathrm{SP}_{\mathrm{HB}_1} = \mathrm{SP}_{\mathrm{HB}_2} = \mathrm{SP}_{\mathrm{HB}_4} = \mathrm{SP}_{\mathrm{HB}_5}$$
$$\mathrm{abs}(\mathrm{SP}_{\mathrm{HB}_2} - \mathrm{SP}_{\mathrm{HB}_5})\%set = \frac{\mathrm{Size}_{\mathrm{HB}_5^2}}{\mathrm{Size}_{\mathrm{line}}}\%set$$
$$\mathrm{abs}(\mathrm{SP}_{\mathrm{HB}_5} - \mathrm{SP}_{\mathrm{HB}_4})\%set = \frac{\mathrm{Size}_{\mathrm{HB}_4^5}}{\mathrm{Size}_{\mathrm{line}}}\%set$$

where $\mathbf{X}(\mathrm{HB_i})$ returns the execution counts of HB_i.

6 Evaluation

In this section, we evaluate the proposed method and compare it with the existing method [10]. In this method, which precisely model the access behavior of global variable. However, this method does not analyze SP-relative addressing accesses, pessimistically assuming that such accesses could target any memory address in the stack.

6.1 Evaluation Setup

We implement the proposed method based on the existing WCET analysis tool, Chronos [12]. For simplicity, we only focus on the single-core scenario without consider the shared cache contention. Note that, the proposed method focuses on a precise cache behavior analysis for SP-relative addressing and can be straightforwardly extended to deal with multi-core scenario by integrating it with the existing method [28]. In the experiment, we adopt the processor architecture of the TMS320C66x DSP CPU [22], a high-performance digital signal processor from Texas Instruments. The detailed cache configuration are summarized as follows (Table 2):

Table 2. Cache configurations in the experiment

Cache Level	Capacity	Block Size	Associativity	Hit Latency	Miss Latency
L1 Instruction	32 KB	32B	1	1 cycle	12.5 cycles
L1 Data	32 KB	64B	2	1 cycle	6 cycles

Similar to most relative works [7,9–11], we selected benchmark programs from the Mälardalen WCET Benchmark Suite [6]. All programs were compiled using the TI C6000-CGT compiler v7.4.4. The program uses the default stack start address (compilers usually assign a fixed base address to the stack), and resizes the stack size to 4 KB, which is the default stack size commonly used in embedded systems, which is also same as the GNU Compiler Collection for ARM.

6.2 Experimental Results

Table 3 shows the experimental results. we refer to the estimated WCET bounds derived from our proposed method and the traditional approach as WCET-Our and WCET-Con, respectively. We compare the WCET-Our and WCET-Con and use $\frac{\text{WCET-Con}}{\text{WCET-Our}} - 1$ as a metric to demonstrate the tightness of WCET bound produced by the proposed method. Experimental results in Table 3 reveals that, in comparison to the conventional method, our approach produces a tighter estimated WCET bound for all the benchmark programs. On average, our method achieves a 11.85% improvement in tightness.

Table 3. Estimated WCET bound in cycles from our method and the existing method

Benchmark	WCET-Con	Con-Timr	WCET-Our	Our-Time	$\frac{\text{WCET-Con}}{\text{WCET-Our}} - 1$
bsort100	658159	9.37 s	482852	401.50 s	36.31%
fdct	14624	84.76 s	11420	3511.47 s	28.06%
edn	3669718	48.38 s	2883263	2499.95 s	27.28%
insertsort	5676	7.74 s	4728	389.17 s	20.05%
matmult	550293	25.10 s	459668	794.64 s	19.72%
fir	161006	18.85 s	137000.5	317.20 s	17.52%
bs	642	6.78 s	546.5	76.16 s	17.47%
lms	3735553	52.86 s	3299397	3618.88 s	13.22%
cnt	1649631	11.71 s	1496112	5208.48 s	10.26%
djikstra	12186	276.96 s	11516	6513.51 s	5.82%
jfdctint	53509	75.21 s	50929.5	3158.24 s	5.06%
fibcall	1411	4.76 s	1363.5	72.34 s	3.48%
ud	23866	138.77 s	23080.5	16234.53 s	3.40%
crc	227066	69.84 s	220334.5	781.84 s	3.06%
floyd	10374	25.93 s	10158.5	1013.38 s	2.12%
ns	38355	25.60 s	38235	798.53 s	0.31%
expint	296891	40.81 s	296677	233.99 s	0.07%
prime	245877	8.16 s	245824	178.04 s	0.02%
Average					11.85%

Our method achieves a significant improvement for *bsort100* (36.3%), *fdct* (28.05%), *edn* (27.3%), *insertsort* (20.05%), and *matmult* (19.7%). This is due to the fact that these benchmarks contain a certain number of SP-relative addressing operations, which introduce significant cache interference between SP-relative addressing and global variables. For instance, in *insertsort*, which involves iterative array traversals and element insertions, out of its total of 32 data memory references, 12 are SP-relative addressing. Existing method can not determine the exact memory address of SP-relative addressing, and pessimistically classify all the data memory references as either AM or NC. Our approach identifies Hyper-Block and is capable to justify which SP-relative addressing instructions share the same SP value, enabling more accurate cache interference analysis.

The following List 2 is the source code for *insertsort*. All local variables (i.e., variable *i*, *j*, *temp*) and temporary data are stored in `B15[1]`, `B15[2]` and `B15[3]`. With the execution of `"i = 2"` (line 8), the memory block associated with `B15[1]` to be loaded into the corresponding cache set, since `B15[1]`, `B15[2]` and `B15[3]` belong to the same memory block, all the subsequent instructions related SP address are AH.

```
1  unsigned int a[11];
2  int main()
3  {
4    int  i,j, temp;
5    a[0] = 0;   /* assume all data is positive */
6    a[1] = 11; a[2]=10;a[3]=9; a[4]=8; a[5]=7; a[6]=6; a[7]=5;
7    a[8] =4; a[9]=3; a[10]=2;
8    i = 2;
9    while(i <= 10){
10       j = i;
11       while (a[j] < a[j-1])
12       {
13         temp = a[j];
14         a[j] = a[j-1];
15         a[j-1] = temp;
16         j--;
17       }
18       i++;
19     }
20     return 1;
21 }
22
```

Listing 2. Source Code of insertsort

Consequently, in *insertsort*, 11 data memory references are classified as AH. Similarly, in *bsort100*, with a total of 34 data memory references, 27 being SP-relative addressing, our method categorizes 21 data memory references as AH, a significant improvement over the existing method, which yields none.

However, for benchmarks like ns, expint, and prime, which have less inherently cache conflict for local variables, our method still performs better but the precision improvement is reduced. Our improvement in accuracy is 11.85%, while the analysis time increased by an average of 48.16 times. Considering that this analysis is conducted statically and performed only a limited number of times, the additional time overhead incurred for improved accuracy is worthwhile.

7 Conclusion

This paper introduces a novel cache behavior analysis approach for SP-relative addressing, leveraging the well-established abstract-interpretation method. By introducing a novel analysis unit called Hyper-Block, this method accurately identify the program segments where the SP value remains constant, enabling a more refined cache interference analysis for each Hyper-Block. Across various Hyper-Blocks, we systematically investigate the SP value transitions and encapsulate the program's Worst-Case Execution Time (WCET) calculation as an Integer Linear Programming (ILP) problem. Empirical results underscore the efficacy of the proposed technique in significantly refining the estimated WCET bounds.

Acknowledgments. This work is supported by National Natural Science Foundation of China (Grant No.62432005, 62302270), Shandong Provincial Natural Science Foundation (Grant No. ZR20220F003), Department of Science & Technology of Shandong Province (Grant No. SYS202201), Quan Cheng Laboratory (Grant No. QCLZD202302), Taishan Scholars Program (No. tsqn202211281).

References

1. Audsley, N.C., Burns, A., Davis, R.I., Tindell, K.W., Wellings, A.J.: Fixed priority pre-emptive scheduling: an historical perspective. Real-Time Syst. **8**(2–3), 173–198 (1995)
2. Ballabriga, C., Cassé, H.: Improving the first-miss computation in set-associative instruction caches. In: 2008 Euromicro Conference on Real-Time Systems, pp. 341–350. IEEE (2008)
3. Cullmann, C.: Cache persistence analysis: theory and practice. ACM Trans. Embed. Comput. Syst. **12**(1s) (2013). https://doi.org/10.1145/2435227.2435236
4. Ferdinand, C., Heckmann, R.: aiT: worst-case execution time prediction by static program analysis. In: Jacquart, R. (ed.) Building the Information Society. IIFIP, vol. 156, pp. 377–383. Springer, Boston, MA (2004). https://doi.org/10.1007/978-1-4020-8157-6_29
5. Ferdinand, C., Wilhelm, R.: Efficient and precise cache behavior prediction for real-time systems. Real-Time Syst. **17**, 131–181 (1999)
6. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: 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 (2010)
7. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In: 2006 27th IEEE International Real-Time Systems Symposium (RTSS 2006), pp. 57–66 (2006). https://doi.org/10.1109/RTSS.2006.12

8. Hardy, D., Puaut, I.: WCET analysis of multi-level non-inclusive set-associative instruction caches. In: 2008 Real-Time Systems Symposium, pp. 456–466. IEEE (2008)
9. Hardy, D., Rouxel, B., Puaut, I.: The heptane static worst-case execution time estimation tool. In: Worst-Case Execution Time Analysis (2017). https://api.semanticscholar.org/CorpusID:20377524
10. Huynh, B.K., Ju, L., Roychoudhury, A.: Scope-aware data cache analysis for WCET estimation. In: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 203–212. IEEE (2011)
11. Lesage, B., Hardy, D., Puaut, I.: WCET analysis of multi-level set-associative data caches. In: 9th International Workshop on Worst-Case Execution Time Analysis (WCET 2009). Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2009)
12. Li, X., Liang, Y., Mitra, T., Roychoudhury, A.: Chronos: a timing analyzer for embedded software. Sci. Comput. Program. **69**(1–3), 56–67 (2007)
13. Li, Y.T., Malik, S., Wolfe, A.: Efficient microarchitecture modeling and path analysis for real-time software. In: Proceedings 16th IEEE Real-Time Systems Symposium, pp. 298–307. IEEE (1995)
14. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard real-time environment, pp. 174–189. IEEE Computer Society Press, Washington, DC (1989)
15. Lundqvist, T., Stenström, P.: An integrated path and timing analysis method based on cycle-level symbolic execution. Real-Time Syst. **17**, 183–207 (1999)
16. Lundqvist, T., Stenstrom, P.: 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 1999 (Cat. No. PR00306), pp. 255–262. IEEE (1999)
17. Mancuso, R., Pellizzoni, R., Caccamo, M., Sha, L., Yun, H.: WCET (m) estimation in multi-core systems using single core equivalence. In: 2015 27th Euromicro Conference on Real-Time Systems, pp. 174–183. IEEE (2015)
18. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. Int. J. Softw. Tools Technol. Transfer **11**, 339–353 (2009)
19. Ramaprasad, H., Mueller, F.: Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In: 11th IEEE Real Time and Embedded Technology and Applications Symposium, pp. 148–157. IEEE (2005)
20. Schoeberl, M., Schleuniger, P., Puffitsch, W., Brandner, F., Probst, C.W.: Towards a time-predictable dual-issue microprocessor: the patmos approach. In: Bringing Theory to Practice: Predictability and Performance in Embedded Systems. Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2011)
21. Suhendra, V., Mitra, T., Roychoudhury, A., Chen, T.: Efficient detection and exploitation of infeasible paths for software timing analysis. In: Proceedings of the 43rd Annual Design Automation Conference, pp. 358–363 (2006)
22. Texas Instruments: C66x CPU and Instruction Set Reference Guide. Texas Instruments (2010). https://www.ti.com/lit/pdf/sprugh7. no. SPRUGH7
23. Theiling, H., Ferdinand, C., Wilhelm, R.: Fast and precise WCET prediction by separated cache and path analyses. Real-Time Syst. **18**, 157–179 (2000)
24. White, R.T., Mueller, F., Healy, C., Whalley, D., Harmon, M.: Timing analysis for data and wrap-around fill caches. Real-Time Syst. **17**, 209–233 (1999)
25. Wilhelm, R., et al.: The worst-case execution-time problem–overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. (TECS) **7**(3), 1–53 (2008)

26. Yan, J., Zhang, W.: WCET analysis for multi-core processors with shared L2 instruction caches. In: 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 80–89. IEEE (2008)
27. Zhang, W., Gong, F., Ju, L., Guan, N., Jia, Z.: Scope-aware useful cache block analysis for data cache related preemption delay. In: 2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 63–74 (2017). https://doi.org/10.1109/RTAS.2017.35
28. Zhang, W., Lv, M., Chang, W., Ju, L.: Precise and scalable shared cache contention analysis for WCET estimation. In: Proceedings of the 59th ACM/IEEE Design Automation Conference, pp. 1267–1272 (2022)