Thread-Sensitive Shared Instruction Cache Analysis for Precise WCET Estimation of Multithreaded Programs

Naveeta Rani, P Padma Priya Dharishini, PVR Murthy
Department of Computer Science and Engineering, MS Ramaiah University of Applied Sciences, Bangalore, India

***Abstract*—Real-time systems require strict adherence to task deadlines, making Worst-Case Execution Time (WCET) analysis essential. WCET estimation typically involves static analysis of instructions in a program, making use of models of hardware and architectural units such as shared instruction caches that are as precise as possible. Shared instruction caches pose a challenge because cache behaviour depends on access history and inter-core interference in multicore systems. Existing approaches do not fully exploit thread lifecycle and synchronization semantics when modeling shared instruction cache behavior. In contrast, the proposed TP-WCIP model explicitly incorporates these semantics to eliminate infeasible interference scenarios. By confining interference placement to feasible concurrent execution regions, it achieves more precise WCET estimation. Worst-case latency, due to shared instruction cache accesses in the presence of inter-core interferences, is estimated for each thread in a multithreaded program with a focus on start, join, and synchronization (wait and notify) primitives. A highly precise model, termed Threaded Program Worst-Case Interference Placement (TP-WCIP), is proposed for the static analysis of multithreaded programs to estimate Worst-Case Execution Time (WCET). The proposed model is evaluated in comparison with Cache Block Conflict Number (CCN), Worst-Case Interference Placement (WCIP), and Interference Partitioning (IP). TP-WCIP exploits *concurrency* and *happens-before* relationships induced by start, join, and synchronization primitives to accurately characterize inter-core interferences. The TP-WCIP model for shared instruction cache analysis is validated using benchmark programs and compared against approaches reported in the literature. It is established both theoretically and experimentally that the proposed model TP-WCIP leads to more precise worst-case latency measurements. Result analysis of benchmark programs Papabench and extended Mälardalen benchmarks shows that the TP-WCIP model reduces interferences by up to 27% over IP, 53% over WCIP, and 75% over CCN, while preserving up to 16% more Shared Instruction Cache Hits than IP, 48% than WCIP, and 84% than CCN, thereby delivering more precise static WCET estimates for multi-threaded programs on multicore architectures.**

Keywords*—Synchronization; concurrency; multicore systems; multithreaded program; shared instruction cache analysis; worst-case execution time

I. Introduction

Real-time systems must satisfy strict timing constraints, ensuring that every task completes within its deadline. Missing a deadline can lead to severe consequences, particularly in safety-critical domains. To guarantee deadline adherence under all circumstances, static analysis [1] is employed to verify compliance with timing requirements, even in the worst case. As a result, analyzing the Worst-Case Execution Time (WCET) of each task becomes indispensable. Over time, a well-defined framework for WCET estimation has been established [2]. The process typically begins by constructing the program's control-flow graph (CFG) from its binary executable, followed by determining loop bounds. Next, micro-architectural analysis is carried out to account for the effects of hardware components, such as caches and pipelines, on the execution time of individual instructions. The worst-case execution path is then identified to determine the program's WCET. A central objective of WCET research is to achieve high precision by minimizing the gap between the estimated WCET and the actual WCET. To attain this level of precision, accurate modeling of hardware resources, including memory hierarchy, processor pipeline, shared buses, and other shared resources, is essential.

Cache memory plays a vital role in bridging the performance gap between high-speed processors and comparatively slower main memory. In multicore architectures, caches are often organized into multiple levels, with the last-level cache typically shared among all cores. Given the substantial impact of caches on execution time, the precision of cache analysis fundamentally determines the accuracy of the estimated WCET. Accurate prediction of cache behavior is therefore crucial for precise WCET estimation. However, this task is inherently complex, as cache behavior depends heavily on access history. The problem becomes even more challenging due to inter-core interference, where one core may evict the instructions or data of another at any time. Such unpredictable access patterns in multicore systems often lead to overestimation of WCET.

Although both instruction and data caches influence WCET, their behavior differs significantly. Instruction caches exhibit structured and predictable access patterns driven by control flow, whereas data caches depend heavily on input-dependent memory accesses, making their static analysis less precise. Instruction cache behavior is therefore largely deterministic, while data cache behavior is inherently more indeterministic due to imprecise address analysis and the possibility of a single reference accessing multiple memory blocks. Motivated by this distinction, this study focuses on statically modeling shared instruction cache behavior.

A straightforward approach proposed in the literature is the Cache Block Conflict Number (CCN) method [3], which assumes that interferences from other cores occur continuously at all program points. Although simple, this assumption is overly

pessimistic, as in reality, a single interference can cause at most one additional cache miss. Interference Partitioning (IP)–based techniques [4, 5] enhance the analysis by incorporating the knowledge of the happens-before relationship introduced due to synchronization primitives wait() and notify(). Nevertheless, within each partition, they continue to pessimistically assume that interferences may occur at every program point, indicating the need for further refinement in modeling multicore shared instruction cache. The Worst-Case Interference Placement (WCIP) technique was introduced in [6], formulated as an Integer Linear Programming (ILP) problem. However, ILP-based WCIP suffers from poor scalability when applied to large applications. To address this issue, algorithmic approximations of WCIP [7, 8] were later proposed. These algorithms strategically place interferences from other cores such that instruction accesses corresponding to shared instruction cache hits are prioritized when they are already near eviction. The primary limitation of the WCIP algorithm is that it assumes independent threads execute on separate cores without any dependencies. In reality, multi-threaded programs rarely behave this way. They often contain inter-thread dependencies. In multithreaded programs, certain regions of different threads can never execute concurrently due to happens-before relationships induced by thread lifecycle operations and synchronization primitives such as start(), join(), wait(), and notify(). For example, in Fig. 1, the instructions in thread T1 from begin(T1) to start(T2) can never execute in parallel with any instructions of thread T2. Therefore, considering interference from thread T2 on shared instruction cache (SIC) accesses within the region T1[begin(T1) to start(T2)] leads to false interference scenarios that cannot occur during actual program execution. The WCIP algorithm does not consider concurrency or happens-before relationships, which can otherwise be exploited to determine a more precise set of potential interferences during the execution of the thread under analysis. To address this limitation, this study proposes a novel approach called Threaded Program Worst Case Interference Placement (TP-WCIP). TP-WCIP integrates concurrency and happens-before analysis by incorporating thread lifecycle operations (start() and join()) and synchronization primitives (wait() and notify()) to identify sequential, concurrent, and competing regions, as well as the resulting concurrent and competing partitions in multithreaded programs. By leveraging this information, TP-WCIP enables a more accurate computation of the interference budget [6–8] by accounting for potential evictions of SIC hits.

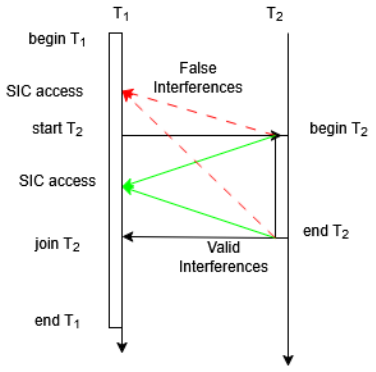

Fig. 1. Illustration of valid and false interferences.

The key research gaps addressed in this work are as follows:

1) Construction of a precise set of interferences from competing threads or cores for shared instruction cache accesses in a thread by extracting precise concurrency information from the Message Sequence Chart (MSC) representation of a multithreaded program with calls to start(), join(), wait(), and notify().

2) Design, verification, and experimental validation of the TP-WCIP model, which uses extracted concurrency information and interference sets to estimate additional shared instruction cache (SIC) misses caused by inter-thread interference for WCET computation of multithreaded programs.

To evaluate TP-WCIP against existing approaches-Interference Partitioning (IP), Worst-Case Interference Placement (WCIP), and Cache Block Conflict Number (CCN), on real-world applications, namely *Papabench* and *extended* Mälardalen, are considered to execute on 4-cores. Experimental results show that TP-WCIP model significantly improves interference analysis accuracy by considering only valid inter-thread interferences. Specifically, it accounts for about 73% of the interferences identified by IP, 47% of those identified by WCIP, and 25% of those identified by CCN, indicating that the IP, WCIP, and CCN approaches include a substantial number of invalid interferences. These correspond to false positives of up to 27%, 53%, and 75%, respectively, in the estimation of worst-case latency for shared instruction cache (SIC) accesses. In addition, TP-WCIP preserves more shared instruction cache hits (SICHs), retaining up to 16% more than IP, 48% more than WCIP, and 84% more than CCN. By reducing false interference assumptions and maintaining more valid cache hits, TP-WCIP enables more precise static WCET estimation for multithreaded programs on multicore architectures.

The rest of this study is structured as follows. Section II presents the related work. Section III introduces the relevant terminology and key concepts. Section IV presents the design of the TP-WCIP algorithm and provides a formal proof of its safety and correctness. Section V reports the experimental results. Finally, Section VI concludes the study.

II. Related Work

In the literature, a variety of hardware modeling techniques have been investigated. Among hardware resources, the shared bus is one of the critical resources, as it introduces significant unpredictability in timing analysis. Extensive research has therefore been devoted to the analysis of shared bus behavior [9]. Cache modeling techniques have also received considerable attention. One notable approach is the must and may analysis [10], which determines whether data is guaranteed or potentially present in the cache. Another line of research applies abstract interpretation to approximate concrete cache states, focusing on the maximal and minimal ages of cache blocks [11]. The work in [12] surveys key challenges and static analysis methods for independent programs under different cache configurations, as well as existing tools that implement static cache analysis techniques. The work in [13] introduces a framework that integrates abstract interpretation with program verification,

enabling a wide range of cache analyses spanning both single-core and multicore architectures.

An integrated timing analysis was proposed that accounts for the combined timing effects of the shared bus and shared cache in [14]. This work also represents the first analysis of cache-related preemption delay in multi-level cache hierarchies, with a focus on non-inclusive caches. Subsequently, cache analysis techniques have been extended to multi-level inclusive caches [15]. Studies such as [16, 17] analyze direct-mapped shared instruction caches in multicore systems, while [3] introduces a technique for set-associative shared caches by estimating the number of potentially interfering cache blocks, defined as the cache block conflict number (CCN). In this context, a cache access is classified as a hit if the number of conflicting blocks is less than the associativity minus the block's age. The Worst-Case Interference Placement (WCIP) approach [6-8] refines interference modeling further. Specifically, when core 1 accesses the shared instruction cache, interferences from other cores that have already been accounted for are excluded from further consideration. This is achieved through the concept of an interference budget, where each interference is deducted from the budget once considered, ensuring that it is not double-counted. More recently, [18] incorporates timing information to establish tighter bounds on worst-case interference, while also classifying individual memory accesses as either guaranteed cache hits or potential cache misses. Existing literature largely addresses cache analysis in non-preemptive scheduling contexts, whereas [19] models interference in a preemptive system to categorize accesses as cache hits or possible misses. In [20], a fine-grained shared-cache interference analysis uses path-based basic-block execution times to model memory-access lifecycles and exclude infeasible interferences.

Building on these approaches, integrating thread lifecycle semantics enables more precise interference placement. This is the primary challenge addressed in this study.

III. Key Concepts and Terminologies

This section outlines the essential definitions and concepts necessary to understand the TP-WCIP approach.

In Fig. 2, the WCET of the thread T_1 is computed. Therefore, thread T1 is referred to as the reference thread. Threads T1, T2, T3, and T4 access the shared instruction cache. Therefore, T2, T3, and T4 are competing threads relative to thread T1.

1) Interference: A Shared Instruction Cache (SIC) access issued by a competing thread is termed an interference [6-8] with the instruction in the reference thread.

2) Sequential region: A sequential region denotes a contiguous set of instructions in the reference thread during which no other threads are executing concurrently. It captures intervals of sequential execution within an otherwise multithreaded program. For example, in Fig. 2, regions T_1[begin(T_1) : start(T_2)], T_1(join(T_2) : start(T_4)] and T_1(join(T_4) : end(T_1)] are the sequential regions of threads T_1 and are denoted as SR_{11}, SR_{12} and SR_{13} respectively.

3) Concurrent region: A concurrent region refers to the instructions between the corresponding start() and join() calls of a reference thread, during which multiple threads execute simultaneously. This region captures the period of concurrent execution within a multithreaded program. For example, in Fig. 2, regions T_1(start(T_2):join(T_2)], T_1(start(T_4):join(T_4)] are the concurrent regions of reference thread T_1, denoted as R_{11} and R_{12} respectively.

4) Competing region: A competing region refers to the set of instructions in competing threads that may execute in parallel with the instructions of a concurrent region in a reference thread. These regions represent portions of competing threads that execute simultaneously, potentially accessing SIC during concurrent execution. For example, in Fig. 2, regions T_2(begin(T_2):end(T_2)], T_3(begin(T_3):end(T_3)] are the competing regions of concurrent region CR_{11}, denoted as CR_{21} and CR_{31} and region T_4(begin(T_4):end(T_4)] is the competing region of CR_{12} and denoted as CR_{41}.

In real multithreaded programs, synchronization is not limited to the start() and join() primitives. The wait() and notify() synchronization primitives also induce happens-before relationships. These relationships can be leveraged to further minimize concurrent executions. If concurrent and competing regions contain matching pairs of wait() and notify(), these constructs can be used to further partition execution into concurrent and competing partitions.

5) Concurrent partitions: A concurrent partition is a fragment of a concurrent region that interacts with other parallel fragments from competing regions. The concurrent region R_{11} is partitioned in two concurrent partitions, P_{111} : T_1(start(T_2): notify(T_2)) and P_{112} : T_1[notify(T_2) : join(T_2)] while concurrent region R_{12}, is partitioned in two concurrent partitions P_{121} : (start(T_4): wait(T_4)], and P_{122} : (wait(T_4): join(T_4)], based on the synchronization primitives.

6) Competing partitions: A partition p denotes a parallel fragment whose corresponding fragments in interacting or competing regions are identified through the method Competing_Partitions(p). Concurrent partition P_{111} may execute in parallel to CP_{211} and CP_{311}, but can never execute in parallel to CP_{212} since wait() always happens-before notify(). Similarly, Concurrent partition P_{112} may execute in parallel to CP_{212} but can never execute in parallel to CP_{211}. Therefore, CP_{211} and CP_{311} are competing partitions of P_{111}, and CP_{212} is a competing partition of P_{112}.

7) Concurrency and interferences in threads: A multithreaded program P say consists of threads T_1, T_2, T_3, and T_4. The corresponding Message Sequence Chart [21] is shown in Fig. 2. Assume that T_1, T_2, T_3, and T_4 run on core$_1$, core$_2$, core$_3$, and core$_4$, respectively. T_1(start(T_2) : join(T_2)] || T_2 || T_3. SIC accesses in T_1[begin(T_1): start(T_2)] are not subject to interferences, whereas SIC accesses in T_1(start(T_2): join(T_2)] are subject to SIC interferences from thread T_2 and T_3. The SIC accesses T_1(join(T_4): end(T_1)] again, not subject to any interferences.

Ideally, a worst-case interference placement (WCIP) algorithm should be aware of both the sequential and concurrent execution regions of the reference thread in a multithreaded program, as well as the competing regions of other concurrently

executing threads. Such awareness allows the algorithm to model interferences only within concurrent regions while excluding them from sequential regions, thereby enabling more precise computation of the worst-case execution time (WCET) of multithreaded programs. However, when a WCIP algorithm does not accurately identify concurrent regions, as in the approaches presented in [6–8], imprecision arises in mapping interferences from competing threads to shared instruction cache (SIC) accesses of the reference thread, as illustrated in Fig. 1.

To analyze and infer information about concurrent regions, competing regions, concurrent partitions, and competing partitions, a Message Sequence Chart (MSC) is used to represent the multithreaded program. In this representation, the program—consisting of the main (reference) thread and competing threads—is modeled as an MSC [21]. The start(), join(), wait(), and notify() method calls are explicitly shown in the MSC to illustrate concurrency. These calls represent thread creation, synchronization, and termination, thereby revealing the concurrent structure of the program. Each corresponding pair of start() and join() calls defines concurrent regions across the involved threads, as illustrated in Fig. 2. The sequential, concurrent, and competing regions of the multithreaded program with respect to the reference thread T_1 in Fig. 2 are summarized in Table I.

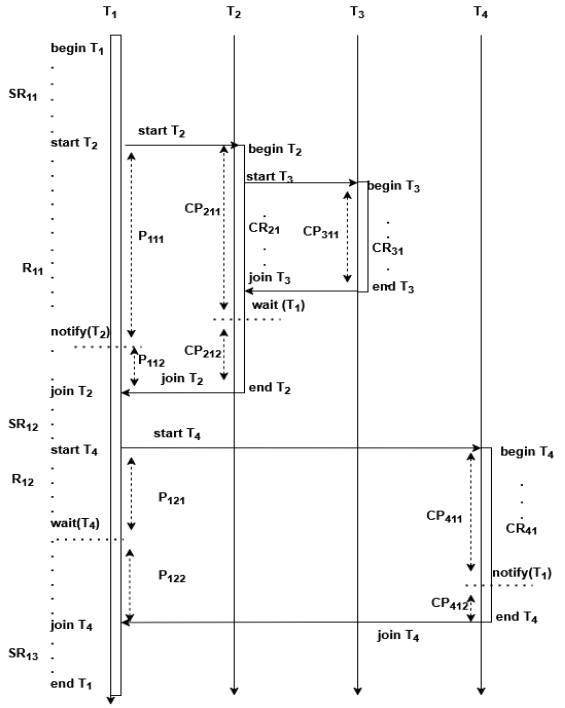

Fig. 2. MSC representation of a multithreaded program showing concurrency and happens-before relationships induced by thread lifecycle and synchronization primitives.

TABLE I. SEQUENTIAL, CONCURRENT, AND COMPETING REGIONS RELATIVE TO THE REFERENCE THREAD (T_1)

Sequential Regions	Concurrent Regions and Respective Competing Regions
S_{11}, S_{12}, S_{13}	R_{11}: CR_{21}, CR_{31} R_{12}: C_{41}

IV. DESIGN

The computation of the worst-case execution time (WCET) of multithreaded programs using Integer Linear Programming (ILP) is an NP-hard problem [6, 8]. Algorithm-based approaches estimate the WCET of multithreaded programs by introducing simplifying assumptions, such as assuming that all shared instruction cache (SIC) hits occur along the worst-case execution path. Compared with ILP-based approaches, algorithm-based methods compute WCET more efficiently while maintaining comparable precision. However, the methods reported in the literature do not sufficiently consider the properties specific to multithreaded programs. To overcome this limitation, this section presents a novel algorithm for WCET computation that explicitly incorporates multithreading-specific properties to reduce imprecision.

A. Threaded Program Worst Case Interference Placement

Fig. 3 presents an overview of the TP-WCIP approach. The input to the TP-WCIP consists of a multithreaded program together with information identifying the reference thread. From the source code of the multithreaded program, the Control Flow Graph (CFG) of the reference thread is constructed. Subsequently, a must analysis for multi-level caches is performed on the CFG using abstract-interpretation-based techniques [11]. The must analysis of the CFG of the reference thread yields information about SIC hits and their corresponding eviction distances [4-8].

In parallel, the source code of the multithreaded program is used to derive a Message Sequence Chart (MSC) [21], which captures thread creation(start()), thread termination (join()), and inter-thread synchronization events (wait() and notify()). This information is used to generate both the threaded event trace and the synchronization event trace. Algorithm 1 analyzes the event trace to determine the sequential and concurrent regions of the reference thread, along with the competing regions in other threads that may run in parallel with the reference thread's concurrent regions. The synchronization event trace is subsequently used to map these concurrent and competing regions to their respective concurrent and competing partitions.

Interference experienced by each concurrent partition of the reference thread is determined via may-analysis of the competing threads and subsequently mapped to the corresponding competing partitions using the synchronization event trace. Next, SIC hit path analysis is performed to identify shared instruction cache hit paths (SICHPs). A SICHP may be entirely contained within a single partition or may span multiple partitions of the reference thread. The SICHPs and concurrent partitions information are then used to map SICHPs to the identified concurrent partitions.

For each concurrent partition, the Overlapping Factor [6, 8] is computed using the paths or subpaths associated with it. The Threaded Interference Placement Algorithm distributes/places interferences across these paths or subpaths within each concurrent partition and determines the resulting worst-case number of cache misses. The aggregate worst-case misses over all partitions are finally multiplied by the SIC miss penalty to obtain the maximum WCET increase due to interferences.

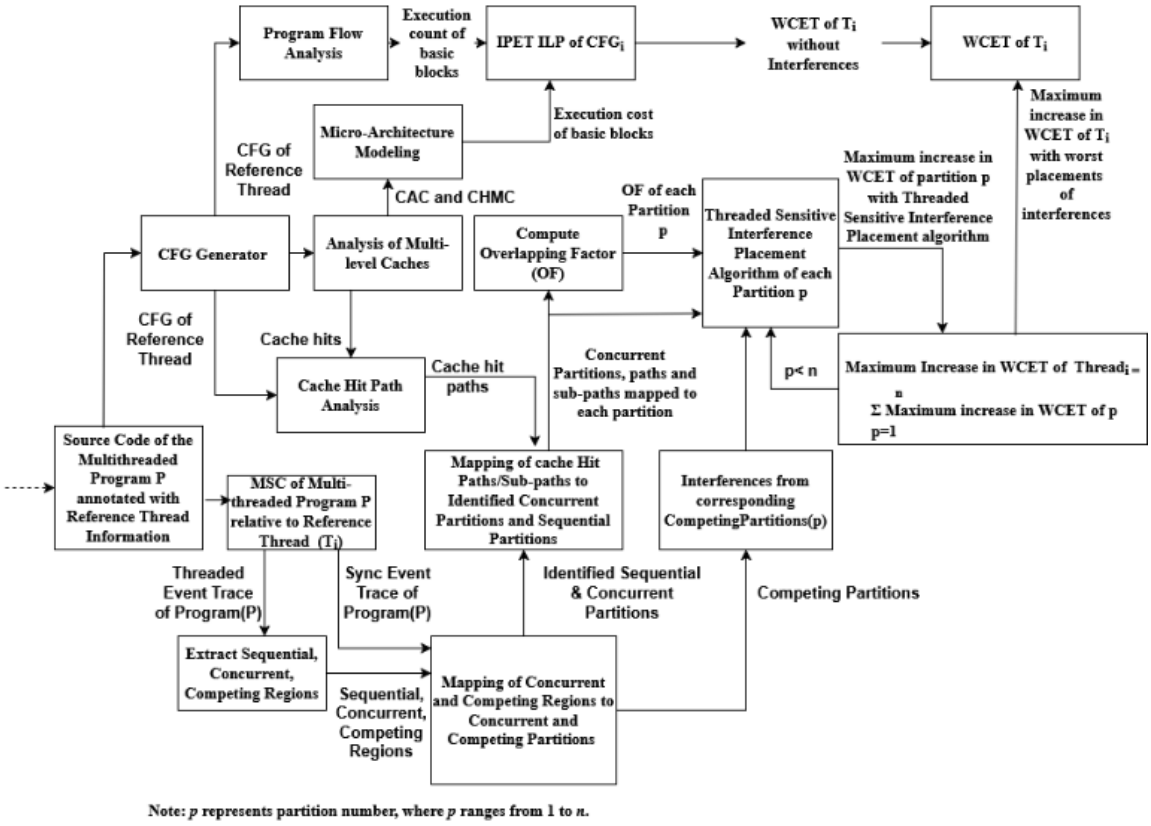

Fig. 3. Overview of the threaded program worst-case interference placement method.

B. Setup

The TP-WCIP methodology begins with a multi-level must-cache analysis of the instruction cache hierarchy on the control flow graph (CFG) of the reference thread T_i. This phase derives both the Cache Access Classification (CAC) and Cache Hit–Miss Classification (CHMC) for all memory accesses at the SIC level. The analysis focuses on SIC accesses whose CAC is classified as Always or Uncertain, and whose CHMC at the shared cache level is Always Hit.

For a reference thread T_i, let SAcc denote the set of instructions that may access the shared instruction cache, and $SAcc_H$ the subset guaranteed to hit in the SIC when no inter-thread interference occurs. Let m_s represent the number of SIC hits mapped to cache set s from $SAcc_H$. SIC hit-path analysis is then performed to compute the list of SIC hit paths, denoted $paths_{sm}$, where each set contains m_s paths mapped to cache set s. For each cache hit path, the eviction distance (ED) is calculated, represented as ED_{si} for i = 1 … m_s. The definition of eviction distance for a cache hit path follows the same principles as the eviction distance described in [6-8].

The MSC is extracted from the program source code to generate two traces: threaded_event_trace and synchronisation event trace. The threaded_event_trace records thread lifecycle events, including creation, start(t_i), and termination join(t_i). Algorithm 1 uses this trace to derive sequential_regions, concurrent_regions, and competing_regions, namely, ordered lists describing the sequential and concurrent regions of the reference thread and the competing regions that may interfere with concurrent regions. The sync_event_trace captures ordered synchronization events such as wait(t_i) and notify(t_i).

Each concurrent partition p in the reference thread is identified using both the concurrent_regions and sync_event_trace of the multithreaded program P, while respecting the happens-before relationship wait < notify. Let $mapPathsToSubpaths_{sm}$ denote the mapping that associates every path in $paths_{sm}$ with its corresponding subpaths across different partitions. The interference budget for the cache set s within concurrent partition p, denoted IB_{sp}, represents the number of SIC accesses to set s originating from CompetingPartitions(p). Let A_{sp} be the set of shared instruction-cache accesses within these competing partitions; then

$$IB_{sp} = \{A \in A_{sp} \mid A \text{ maps to cache set } s \wedge CAC(A) \in \{Always, Uncertain\}\} \quad (1)$$

C. Extract Regions from the Threaded Event Trace

The threaded event trace (threaded_event_trace) of a multithreaded program records thread lifecycle events, specifically creation (start()) and termination (join()). Algorithm 1 takes the threaded_event_trace and a reference thread (t_i) as inputs and uses this information to determine the sequential and concurrent regions of the reference thread, denoted as sequential_regions and concurrent_regions, respectively, as well as the corresponding competing regions in other threads, denoted as competing_regions, that may execute in parallel with the reference thread's concurrent regions. The algorithm

identifies concurrent regions as segments of the reference thread enclosed between start(t_j) and the corresponding join(t_j), while the execution segment preceding each concurrent region is classified as a sequential region. It also detects nested thread creation. When multiple threads are spawned in a nested manner, the concurrent region is extended to cover the full overlapping lifetimes of the involved threads, and all overlapping execution segments are added to the competing regions. Furthermore, the algorithm recursively analyzes threads spawned within competing regions to identify additional competing execution regions. Finally, any remaining portion of the reference thread is treated as the last sequential region, and the algorithm returns sequential_regions, concurrent_regions, and competing_regions.

Algorithm 1: ExtractRegionsFromPartialOrderInformation (threaded_event_trace, t_i)

```
/*threaded_event_trace: Ordered list of events from a multithreaded program (events include start(t_i), join(t_i))*/
    sequential_regions ← empty list
    concurrent_regions ← empty list
    competing_regions ← empty list of lists
    competing_threads ← empty list
    seq_region_start = begin(t_i)
    concur_region_count = 0
    while not end of threaded_event_trace do                /* Step 1: Search for thread creation */
        | if event is start(t_j) then                       /* Sequential region before t_j starts */
        |  | sequential_regions.append(region from seq_region_start to start(t_j))
        |  join(t_j) ←  find_matching_join(t_j)             /* Step 2: Identify lifespan of t_j */
        | nested_thread_found ← false                       /* assume there are no nested threads spawned*/
    |   | for each event e_k between start(t_j) and join(t_j) do    /* Step 3: Check for nested thread spawning */
        |  | if e_k is start(t_k) and t_k ≠ t_i then
        |  |          | nested_thread_found ← true
        |  |          | join(t_k) ← find_matching_join(t_k)
        |  |          | temp_region = region from start(t_j) to max(join(t_j), join(t_k))
    |   |  | concurrent_regions.insert(concur_region_count, temp_region)
    |   |  | seq_region_start = max(join(t_j), join(t_k))
        |  |          | temp_region1 = region from begin(t_j) to end(t_j)
        |  |          | temp_region2 = region from begin(t_k) to end(t_k)
        |  |          | competing_regions.insert( concur_region_count,[ temp_region1, temp_region2])
    |   |  | competing_threads.append(thread t_i & thread t_k)
        |  |          | concur_region_count = concur_region_count + 1
        |  | end if
        | end for
    | if nested_thread_found = false then                 /* Step 4: No nested thread within t_j */
    |   | temp_region = region from start(t_j) to join(t_j)
        |  | concurrent_regions.insert(concur_region_count, temp_region)
        |  | temp_region1 = region from begin(t_i) to end(t_i)
    |   | competing_regions.insert(concur_region_count, [temp_region1])
        |  | seq_region_start = end(t_i)
    |   | competing_threads.append(thread t_j)
    |   | concur_region_count = concur_region_count + 1
    | end if
    | move to next event
    end while
    /* Step 5: Detect cascading concurrency when a competing thread creates an additional thread */
    for each competing_thread in competing_threads
    | seq, con, comp = extract_Regions_From_Partial_OrderInformation( threaded_event_trace, competing_thread
        | insert_returnedInfo_to_Respective_Data_Strucutres(seq, con, comp)
    End for
    sequential_regions.append(region from seq_region_start to end(ti)) /* Step 6: Append last sequential region of reference thread */end
    return sequential_regions, concurrent_regions, competing_regions
    End
```

D. Threaded Sensitive Interference Placement Algorithm

Algorithm 2, Threaded Sensitive Interference Placement algorithm, places the interferences on the SIC subpaths to compute the maximum increase in WCET of reference T_i caused by interferences from threads executing on the other cores. The algorithm takes the following inputs:

- n: number of concurrent partitions in the reference thread T_i
- $listOfSubPaths_{sp}$: subpaths mapped to SIC set s in partition p
- ED_{sp}: eviction distances of subpaths mapped to SIC set s in partition p
- IB_{sp}: interference budget of SIC set $s$in partition p
- BLK_{sp}: number of interfering blocks mapped to SIC set s in partition p
- A: SIC associativity
- $SIC\_Miss\_Penalty$: latency of an SIC miss in cycles

The Threaded Sensitive Interference Placement algorithm places interferences partition by partition. It adopts a greedy strategy in which interferences are inserted in increasing order of eviction distance, and once an interference is considered, it is not revisited. However, when overlap exists among SIC subpaths, a single interference may affect multiple SIC hit blocks already present in the cache, making a greedy strategy unsuitable. To address this, for each SIC set s in partition p, the function compute_OF_partition() is invoked to calculate the Overlapping Factor (OF). The OF measures the extent of overlap among subpaths. This factor is multiplied by the interference budget to obtain a new interference budget. Since overlapping SIC subpaths allow one interference to affect multiple cache hits, the adjusted interference budget captures this effect, enabling the greedy strategy to be applied for distributing interferences.

Next, the subpaths in the set are grouped according to their eviction distance (ED), for example, the number of subpaths with ED = 1, ED = 2, and so on up to associativity A.

A cache hit can only be converted into a miss if the number of interfering blocks is greater than or equal to its eviction distance. If the number of interfering blocks is less than the eviction distance, the corresponding cache hit cannot be evicted by interference.

Algorithm 2: Threaded Sensitive Placement Algorithm

Input: n: number of partitions in reference thread t, $listOfSubPath_{sp}$: list of SIC hit subpaths mapped to SIC set s of partition p, ED_{sp} : list of eviction distances of subpaths mapped to SIC set s of partition p, IB_{sp} : list of Interference Budget of SIC set s of partition p, BLK_{sp}: number of interfering blocks of SIC set c in partition p, A: SIC associativity, SIC_Miss_Penalty : SIC miss cycles

Output: Maximum increase in WCET of thread t_i

```
p ← 1
SIC_Misses ← 0
while p ≤ n do:
| for each SIC set s do:
    | | OF_sp = compute_OF_partition( subPaths_sp )
    | | NEW_IB_sp = OF_sp * IB_sp
|   | subPathsCountByED_sp[1…A] = computeSubpathCountByED( listOfSubPath_sp, ED_sp)
    | | for i ← 1 to A do
    | |         | if BLK_sp ≥ i then
    | |         |         | if NEW_IB_sp ≤ i * subPathsCountByED_sp[i] then
    | |         |         |         | SIC_Misses ← SIC_Misses + ⌈ NEW_IB_sp / i ⌉
    | |         |         |         | NEW_IB_sp ← 0
|   | | |       |         | break
    | |         |         | else
    | |         |         |         |         | SIC_Misses ← SIC_Misses + subPathsCountByED_sp[i]
|   | | |       |         |         | NEW_IB_sp ← NEW_IB_sp - subPathsCountByED_sp[i] * i
    | |         |         | end if
    | |         | end if
    | | end for
    | end for
| p ← p + 1 /* Move to the next partition*/
end while
return SIC_Misses * SIC_Miss_Penalty
End
```

For each eviction distance level i:

- If the available interference budget is insufficient to convert all eligible hits into misses, only a subset of the hits is converted to misses according to the remaining interference budget, after which the budget becomes zero.
- Otherwise, if the interference budget is sufficient, all eligible hits are converted into misses, and the interference budget is reduced accordingly.

This procedure is repeated for all eviction distance levels, for every SIC set in every partition. The total number of additional misses incurred due to interference is accumulated and multiplied by the SIC miss penalty to compute the maximum WCET increase. This increase is then added to the WCET of the reference thread without interference.

The time complexity of the interference distribution algorithm is $O(ns(M + A)$, where M denotes the total number of SIC-hit subpaths, n represents the number of concurrent partitions, s denotes the number of SIC sets, and A is the SIC associativity. Since A and s are fixed architectural constants, the overall complexity reduces to $O(n * M)$ linear in the total number of SIC hit subpaths and partitions in the reference thread.

E. Proof of Safety

Theorem: Let P be a multithreaded program that uses start(), join(), wait(), and notify() primitives for thread creation, termination, and synchronization. For each thread in P, the number of Shared Instruction Cache (SIC) misses estimated by TP-WCIP is greater than or equal to the number of SIC misses that can occur during any concrete execution of P.

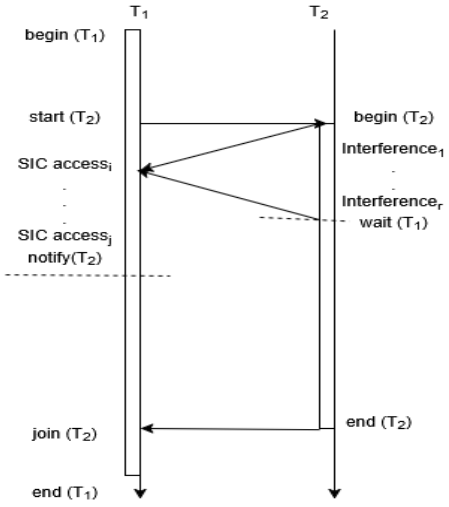

Fig. 4. MSC representation of a multithreaded program with maximal interference set.

Proof: Consider two threads T_1 and T_2 in program P, as illustrated in Fig. 4. Concurrent partitions and subsequently their competing partitions are identified. One of the identified concurrent partitions, p: $T_1[\text{start}(T_2): \text{notify}(T_2)]$ and the corresponding competing partition is: $T_2[\text{begin}(T_2): \text{wait}(T_2)]$. Both threads may execute concurrently within these two partitions.

Let SIC access_i be an instruction access to the SIC mapped to set s in thread T_1 occurring within the concurrent partition p. The TP-WCIP algorithm constructs the interference budget of the cache set s in partition p.

$$IB_{sp} = \{interferences_1 \dots interferences_r\} \quad (2)$$

where each interferences_l:

- belongs to the thread T_2,
- maps to the same cache set as a_i, and
- can execute concurrently within the identified partition.

Because the interference set is derived from the concurrent execution partitions, any interfering access that may occur during a concrete execution must belong to this set. Therefore, TP-WCIP includes all feasible interfering accesses and thus forms an over-approximation of the interference that may occur during a concrete execution.

Next, TP-WCIP processes instruction accesses in non-decreasing order of their eviction distances. By analyzing accesses with smaller eviction distances first, TP-WCIP constructs a scenario that maximizes the number of potential SIC misses. This ordering increases the likelihood of eviction and, therefore, models the worst-case scenario. As a result, TP-WCIP produces the maximum possible number of potential SIC misses. In contrast, during an actual concrete execution, the order of accesses is constrained by the program control flow and synchronization. If a subsequent instruction has a smaller eviction distance, the execution cannot reorder accesses to prioritize that instruction as TP-WCIP does. Therefore, the concrete execution cannot always realize the worst-case ordering assumed by TP-WCIP. Therefore,

$$Misses_{TP-WCIP}(p) \geq Misses_{concrete}(p) \quad (3)$$

The above reasoning applies to every concurrent partition induced by the thread life cycle and synchronization primitives. Since a thread's execution can be viewed as a sequence of sequential regions interleaved with concurrent partitions, the safety property holds independently for each concurrent partition, while the sequential regions remain unaffected by interferences. Therefore, the property holds for the entire thread as well.

V. Results and Discussion

Experiments are conducted using the multicore Chronos WCET analysis tool [22, 23], which supports multicore shared instruction cache (SIC) analysis. Our extension to multicore Chronos implements four SIC analysis methods: WCIP, Interference Partitioning (IP), Cache Block Conflict Number (CCN), and the proposed TP-WCIP technique. WCIP, IP, and CCN are widely discussed in the literature, so TP-WCIP is evaluated against them. Experiments run on a quad-core, timing anomaly-free [24] architecture with a 512-byte, 4-way private instruction cache (32-byte block) and an 8-KB, 8-way shared instruction cache (32-byte block). Private cache hits take 1 cycle, L2 shared instruction cache hits take 6 cycles, and

memory accesses take 30 cycles. The reference thread runs on core 1, with worst-case adversaries on cores 2-4. Benchmarks include real-world applications Papabench and extended Mälardalen, featuring input-dependent loops, nested conditionals, recursive calls, and auto-generated code.

A. Shared Instruction Cache Interference Reduction

The primary metric used to assess the performance of the TP-WCIP approach is Shared Instruction Cache Interference (SICI) reduction. It is computed as:

$$SICI\ Reduction(\%) = \frac{(SICI_{existing} - SICI_{TP-WCIP})}{SICI_{existing}} * 100 \quad (4)$$

The results presented in Table II demonstrate the effectiveness of TP-WCIP in reducing SICI across a diverse set of benchmark programs. Overall, TP-WCIP consistently reduces SICI compared with the baseline approaches. On average, TP-WCIP achieves a 27% reduction relative to IP and a 53% reduction relative to WCIP. When compared with CCN, TP-WCIP attains an average reduction of 75%, highlighting its strong improvement over existing approaches. These gains arise because TP-WCIP considers only feasible interferences originating from competing regions and competing partitions.

TABLE II. SIC INTERFERENCE REDUCTION USING TP-WCIP, COMPARED AGAINST WCIP AND CCN

Program	SICI Reduction (%) of TP-WCIP Relative to IP	SICI Reduction (%) of TP-WCIP Relative to WCIP	SICI Reduction (%) of TP-WCIP Relative to CCN
HeavyCompute	27.99	49.29	92.53
CascadeCall	28.8	51.94	91.72
LoopNest	39.43	69.71	89.58
Papabench	4.34	8.33	20.28
HighOverlap	38.51	85.71	85.38

The improvement is particularly notable when compared with CCN, where the reduction ranges from 20.28% to 92.53%. This trend indicates that incorporating thread-aware analysis substantially eliminates infeasible interference that arises when thread interactions are not precisely modelled. Similarly, reductions relative to WCIP range from 8.34% to 85.71%. While WCIP avoids repeatedly accounting for the same interference, it still overestimates feasible concurrency. The most significant improvement of TP-WCIP over WCIP appears in HighOverlap, where a single concurrent partition exhibits substantial overlap. WCIP computes the overlapping factor and interference at the thread level, which causes the interference budget to grow dramatically. In contrast, TP-WCIP restricts this growth to the specific partition where the high overlap occurs, instead of propagating it across the entire thread. As a result, the interferences are localized to the affected partition rather than being attributed to cache hit paths that would not actually be executed during program execution.

The reductions relative to IP vary between 4.34% and 39.43%. This variation indicates that IP still introduces notable pessimism, which TP-WCIP reduces by leveraging additional multithreading semantics.

In contrast, TP-WCIP demonstrates only modest improvement over the other approaches for the Papabench program. This is primarily because the competing threads and the reference threads in Papabench perform very few shared cache accesses and cache hits, resulting in relatively low interference and therefore limited opportunity for further reduction. Overall, these results highlight the advantage of TP-WCIP in accurately constraining interference by incorporating thread-aware execution semantics. In particular, TP-WCIP is particularly effective for programs with complex concurrency patterns or with highly overlapping shared instruction cache hit paths that are confined to specific regions of the program rather than spread across the entire program.

B. Retention of Shared Instruction Cache Hits

The second metric used to evaluate the performance of the TP-WCIP approach is Shared Instruction Cache Hits (SICH) retention. This retention is calculated as:

$$SICH\ Retention(\%) = \frac{(SICH_{proposed} - SICH_{existing})}{SICH_{proposed}} * 100 \quad (5)$$

Fig. 5 illustrates the improvement in retaining Shared Instruction Cache Hits (SICHs) achieved by the TP-WCIP algorithm over three baseline schemes: IP, WCIP, and CCN, across all benchmark programs. The results demonstrate that TP-WCIP consistently outperforms all baseline algorithms, although the magnitude of improvement varies with all the baseline and benchmark characteristics.

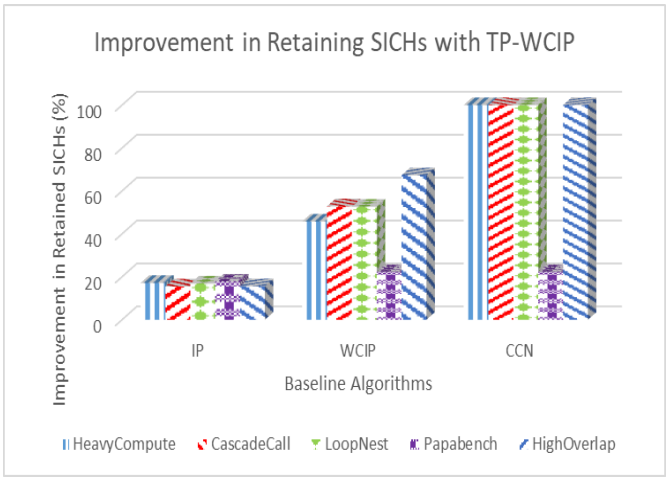

Fig. 5. Improvement of TP-WCIP algorithm in retaining shared instruction cache hits over IP, WCIP, and CCN.

TP-WCIP achieves improvements of up to 25%, with an average gain of 16% over the IP baseline across all benchmarks. This performance comes from its interference-placement strategy, which avoids repeatedly accounting for the same interference. Compared with WCIP, the performance improvement ranges from 30% to more than 70%, with an average improvement of up to 48%. This advantage arises because TP-WCIP retains a greater number of SICHs by allocating interferences to SICHs where interferences are expected during execution. The most significant improvements are observed when compared against CCN, where TP-WCIP achieves 100% improvements for most benchmarks, with an

average improvement of 84%, highlighting its scalability and superior cache modeling capability.

C. Shared Instruction Cache Hit Ratio

The next metric used to evaluate the performance of the proposed approach is the shared instruction cache hit ratio (SICHR). The SICHR is calculated as:

$$SICHR = \frac{SICH_M}{SICH_{without\_Interference}} \quad (6)$$

$SICH_M$ is the number of shared instruction cache hits with interferences, where M is the shared instruction cache modeling approach (TP-WCIP, IP, WCIP, CCN).

$SICH_{without\_Interference}$ is the number of shared instruction cache hits without interferences.

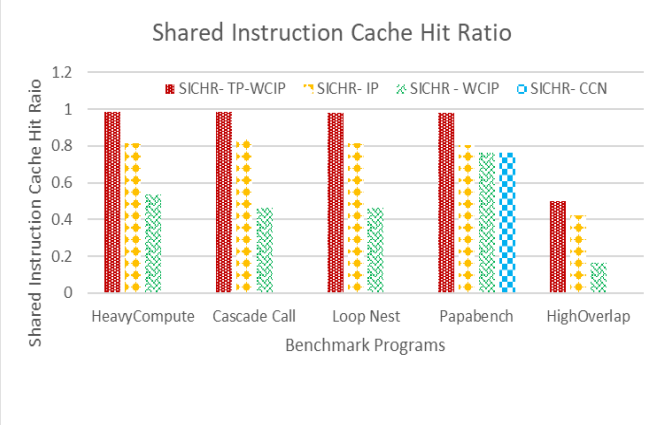

Fig. 6. SICHR of benchmark programs under different interference placement algorithms, TP-WCIP, IP, WCIP, and CCN.

Fig. 6 reports the Shared Instruction Cache Hit Ratio (SICHR) for all benchmark programs under four cache modeling schemes: TP-WCIP, IP, WCIP, and CCN. A value of 1.0 indicates complete preservation of shared instruction cache hits in the presence of interference; lower values indicate hit loss due to contention and evictions. Two principal trends are evident. First, TP-WCIP consistently attains SICHR values close to 1 across HeavyCompute, CascadeCall, and LoopNest (≈0.95–1.0), indicating that TP-WCIP effectively preserves shared instruction cache hits even when the thread is subject to interferences. The baseline schemes show substantially lower SICHRs: IP retains roughly ~0.8 (≈80% of isolated hits), while WCIP shows greater variability (≈0.45–0.77 across workloads). CCN fails to preserve any SICHs for the benchmark programs HeavyCompute, CascadeCall, LoopNest, and HighOverlap. This indicates that CCN cannot retain Shared Instruction Cache hits mapped to a cache set s when the eviction distance of those hits is less than or equal to the total number of interfering blocks mapped to the same cache set s across the entire multithreaded execution. Consequently, CCN becomes overly pessimistic for high-contention benchmarks such as HeavyCompute, CascadeCall, LoopNest, and HighOverlap. In contrast, Papabench exhibits lower contention, allowing CCN to preserve some Shared Instruction Cache hits in that program.

D. False Interference Placement Ratio

The next metric used to evaluate the performance of the cache modeling approaches is the false interference placement ratio (FIPR). The FIPR is calculated as:

$$FIPR = \frac{Number\ of\ False\ Interferences\ Placed}{Total\ Interferences\ Placed} \quad (7)$$

TABLE III. FIPR OF IP, WCIP, AND CCN ALGORITHMS

Program	Heavy Compute	Cascade Call	Loop Nest	Papa Bench	High Overlap
FIPR-IP	0.27	0.28	0.39	0.04	0.38
FIPR-WCIP	0.49	0.51	0.69	0.08	0.85
FIPR-CCN	0.92	0.91	0.89	0.2	0.85

Table III shows the FIPR of the IP, WCIP, and CCN approach. Notably, TP-WCIP achieves an FIPR of zero across all benchmark programs. In comparison, CCN exhibits the highest level of false interference placement, with an average FIPR of approximately 75%, followed by WCIP with about 53%, and IP with around 27%. These false interferences directly contribute to the imprecision in the computation of the WCET. A higher FIPR indicates that the analysis introduces a significant number of infeasible interference scenarios that cannot occur during actual program execution. As a result, the number of predicted Shared Instruction Cache (SIC) interferences increases, leading to a more imprecise WCET estimate. Consequently, reducing FIPR is essential for achieving tighter and more realistic WCET bounds. The results demonstrate that TP-WCIP significantly reduces false interference scenarios compared to existing approaches. This improvement directly translates to tighter WCET bounds, enabling more efficient utilization of computational resources in real-time systems.

E. Implications for System-Level Optimization and Modern Architectures

The improvements achieved by TP-WCIP have important implications for system-level design in multicore real-time systems. From an architectural standpoint, the insights from this work apply to modern processors featuring multi-level cache hierarchies and shared last-level caches (LLCs). Although the current model focuses on a two-level instruction cache, the underlying principle of restricting interference to feasible concurrency regions naturally extends to the analysis of interference propagation across multiple cache levels. These observations indicate that incorporating concurrency-aware interference modeling, as realized in TP-WCIP, can enhance both timing predictability and cache utilization efficiency in contemporary multicore architectures.

VI. CONCLUSION AND FUTURE WORK

Precise static Worst-Case Execution Time (WCET) analysis for multithreaded programs on multicore architectures critically depends on accurate shared instruction cache (SIC) modeling. Existing approaches address inter-core interference in a conservative manner. In particular, the Cache Block Conflict Number (CCN) technique assumes interference at all program points, effectively classifying most shared instruction cache accesses as misses. Although Worst-Case Interference

Placement (WCIP) reduces some pessimism by ensuring that already-accounted interference is not repeatedly considered, it does not exploit multithreading semantics to eliminate infeasible interferences implied by happens-before relationships. Interference Partitioning (IP) further reduces pessimism by incorporating synchronization primitives, but it still does not fully leverage multithreading semantics, particularly parallelism and happens-before relations introduced by thread lifecycle primitives.

This work introduced the Threaded Program Worst-Case Interference Placement (TP-WCIP) approach, which explicitly incorporates thread lifecycle primitives (start(), join()) and synchronization primitives (wait(), notify()) to derive precise sequential regions, concurrent regions, competing regions, and corresponding partitions. By leveraging a concurrency structure extracted from the Message Sequence Chart (MSC) representation, TP-WCIP restricts interference placement to only those accesses that can execute in parallel with the reference thread. Consequently, false interference scenarios are eliminated while safety is preserved.

Experimental evaluation on Papabench and extended Mälardalen benchmark programs demonstrates that TP-WCIP substantially improves precision. The TP-WCIP reduces shared instruction cache interferences by up to 27% over IP, 53% over WCIP, and 75% over CCN. In addition, it preserves up to 16% more shared instruction cache hits than IP, 48% more than WCIP, and 84% more than CCN. These results confirm that TP-WCIP delivers tighter and safer static shared instruction cache analysis for multithreaded programs executing on multicore systems, without sacrificing computational efficiency.

Despite its advantages, TP-WCIP has certain limitations. The model assumes non-preemptive execution within cores, and its evaluation is restricted to selected benchmark suites and a specific cache configuration, which may limit the generalizability of the results.

As part of future work, the model will be extended to incorporate preemptive scheduling, including the analysis of cache-related preemption delays (CRPD), enabling more accurate WCET estimation in fully preemptive real-time systems.

REFERENCES

[1] J. Park, H. Lee, and S. Ryu, "A survey of parametric static analysis," ACM Computing Surveys (CSUR), vol. 54, no. 7, pp. 1–37, Jul. 2021.

[2] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm, "Static timing analysis for hard real-time systems," in *Proc. International Workshop on Verification, Model Checking, and Abstract Interpretation*, Jan. 17, 2010, pp. 3–22, Berlin, Heidelberg: Springer.

[3] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multicore processors with shared instruction caches," in Proc. 30th IEEE Real-Time Systems Symp. (RTSS), Dec. 2009, pp. 68–77.

[4] P. P. Dharishini and P. V. Murthy, "Precise shared instruction cache analysis to estimate WCET of multi-threaded programs," in Proc. 2021 IEEE 18th India Council Int. Conf. (INDICON), pp. 1–7, Dec. 2021.

[5] P. P. Priya Dharishini and P. Ramana Murthy, "Static analyzer for computing WCET of multithreaded programs using Hoare's CSP," in Proc. 15th Innovations in Software Engineering Conf., Feb. 2022, pp. 1–12.

[6] K. Nagar and Y. N. Srikant, "Precise shared cache analysis using optimal interference placement," in Proc. 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), Apr. 2014, pp. 125–134.

[7] K. Nagar and Y. N. Srikant, "Fast and precise worst-case interference placement for shared cache analysis," ACM Trans. Embed. Comput. Syst., vol. 15, no. 3, pp. 1–26, Mar. 2016.

[8] K. Nagar, "Precise analysis of private and shared caches for tight WCET estimates," Ph.D. dissertation, Indian Institute of Science Bangalore, Bengaluru, India, May 2016.

[9] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified WCET analysis framework for multicore platforms," ACM Transactions on Embedded Computing Systems (TECS), vol. 13, no. 4s, pp. 1–29, Apr. 2014.

[10] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," Real-Time Systems, vol. 17, no. 2, pp. 131–181, Nov. 1999.

[11] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in Proc. 4th ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages (POPL), Jan. 1977, pp. 238–252.

[12] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," Leibniz Transactions on Embedded Systems, vol. 3, no. 1, pp. 05-1, Jun. 2016.

[13] S. Chattopadhyay and A. Roychoudhury, "Scalable and precise refinement of cache timing analysis via path-sensitive verification," Real-Time Systems, vol. 49, no. 4, pp. 517–562, Jul. 2013.

[14] M. Jacobs, S. Hahn, and S. Hack, "WCET analysis for multi-core processors with shared buses and event-driven bus arbitration," in Proceedings of the International Conference on Real Time and Networks Systems (RTNS), Nov. 2015, pp. 193–202.

[15] Z. Zhang and X. Koutsoukos, "Cache-related preemption delay analysis for multi-level inclusive caches," in Proc. 13th Int. Conf. Embedded Software (EMSOFT), Pittsburgh, PA, USA, Oct. 2016, pp. 1–10.

[16] Z. Zhang, X. Koutsoukos, J. Yan, and W. Zhang, "WCET analysis for multicore processors with shared L2 instruction caches," in Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Apr. 2008, pp. 80–89.

[17] Z. Zhang and X. Koutsoukos, "Accurately estimating worst-case execution time for multicore processors with shared direct-mapped instruction caches," in Proc. 15th IEEE Int. Conf. Embedded and Real-Time Computing Systems and Applications, Aug. 2009, pp. 455–463.

[18] T. L. Fischer and H. Falk, "Timing-aware analysis of shared cache interference for non-preemptive scheduling," Real-Time Systems, vol. 60, no. 4, pp. 570–624, Dec. 2024.

[19] T. L. Fischer and H. Falk, "Shared cache analysis under preemptive scheduling," in Proc. 2024 Design, Automation & Test in Europe Conf. & Exhibition (DATE), Mar. 2024, pp. 1–6.

[20] Y. Zhu, W. Lou, Y. Gao, B. Jiang, X. Gong, and X. Li, "Fine-Grained Shared Cache Interference Analysis Using Basic Block's Execution Time," in Proc. 2024 IEEE 42nd Int. Conf. on Computer Design (ICCD), Nov. 2024, pp. 320–323.

[21] A. Roychoudhury, "Depiction and playout of multi-threaded program executions," in Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Oct. 2003, pp. 331–336.

[22] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," ACM Computing Surveys (CSUR), vol. 52, no. 3, pp. 1–38, Jun. 2019.

[23] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," Science of Computer Programming, vol. 69, no. 1–3, pp. 56–67, 2007.

[24] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in Proceedings of the 20th IEEE Real-Time Systems Symposium, Dec. 1999, pp. 12–21.