Real-Time Systems
https://doi.org/10.1007/s11241-024-09430-8

Timing-aware analysis of shared cache interference for non-preemptive scheduling

Thilo L. Fischer[1] · Heiko Falk[1]

Accepted: 13 September 2024

Abstract
In multi-core architectures, the last-level cache (LLC) is often shared between cores. Sharing the LLC leads to inter-core interference, which impacts system performance and predictability. This means that tasks running in parallel on different cores may experience additional LLC misses as they compete for cache space. To compute a task's worst-case execution time (WCET), a safe bound on the inter-core cache interference has to be determined. We propose an interference analysis for set-associative shared least-recently-used caches. The analysis leverages timing information to establish tight bounds on the worst-case interference and classifies individual accesses as either cache hits or potential cache misses. We evaluated the analysis performance for systems containing 2 and 4 cores using shared caches up to 64 KB. The evaluation shows an average WCET reduction of up to 23.3% for dual-core systems and 8.5% for quad-core systems.

Keywords Shared cache · WCET analysis · Multi-core · Event-arrival curve

1 Introduction

Real-time systems are subject to timing requirements, meaning that the tasks in a real-time system must be completed before their associated deadline. Otherwise, the system can fail with catastrophic consequences. In order to verify that the timing requirements are met even in the worst-case, static analyses need to be performed on the system. An important property to analyze is the worst-case execution time (WCET) of each task, which describes the longest duration required to complete an instance of that task.

✉ Thilo L. Fischer
thilo.leon.fischer@tuhh.de

Heiko Falk
heiko.falk@tuhh.de

[1] Institute of Embedded Systems, Hamburg University of Technology, Am Schwarzenberg-Campus 1, 21073 Hamburg, Germany

Modern architectures use caches to bridge the performance gap between the fast processor core and the slower memory. Although caches improve the average case performance of a system, careful analysis is required to determine their worst-case behavior. When analyzing the behavior of a cache, *cache-hit-miss-classifications* (CHMC) are used to describe whether an access will be a hit, a miss, or result in unknown behavior. These classifications can be used to derive a safe upper bound on the WCET. Analyses (Ferdinand and Wilhelm 1999; Touzeau et al. 2019) based on the well established framework of abstract interpretation (Cousot and Cousot 1977) have been developed to analyze the contents of caches. These analyses perform data-flow analyses to determine the maximal and minimal intra-task cache interference for each access to a particular cache block. The result of these data-flow analyses directly induces the CHMC classifications when considering the associativity of the cache.

In modern systems, the trend is shifting from single-core to multi-core systems, where the memory hierarchy typically consists of multiple cache levels, with the last level cache shared between cores. The parallel execution of tasks on different cores influences the state of the shared cache, invalidating the cache hit classifications if the additional cache conflicts due to cache sharing are not considered. For this reason, the effects cache sharing have to be taken into account in the system analysis. However, inter-core interference on shared caches is notoriously difficult to quantify. This unpredictability leads to potential overestimation of the worst-case timing behavior. To avoid this overestimation, it is essential to establish a tight bound on the effects of inter-core cache interference.

In this paper, we propose a novel analysis approach to derive cache hit classifications for individual accesses, considering inter-core interference. The approach is applicable to set-associative shared caches using the *least-recently-used* (LRU) replacement policy. To quantify inter-core interference, we model cache accesses issued by each task as an event stream. These event streams are characterized using the concept of event-arrival curves, which have been applied previously in *network calculus* (Le Boudec and Thiran 2001) and *real-time calculus* (Thiele et al. 2000). By applying the concept of event-arrival curves to shared caches, we can examine the inter-arrival time between multiple cache access events and quantify the impact on the shared cache.

This perspective on inter-core cache interference essentially induces a *time-to-live* (TTL) for information stored in the shared cache. The TTL of a cache block corresponds to the time frame during which interfering tasks cannot issue a sufficient number of conflicting accesses to evict the cache block. Consequently, if a block is accessed before its *time-to-live* has expired, the access will definitely result in a cache hit. We use the term *interference curve* for an event-arrival curve describing shared cache interference in this paper.

For a single interfering task, the interference curve can be derived from the control-flow graph of that task. This was first demonstrated in Fischer and Falk (2023). However, when multiple tasks run on a single interfering core, they can collaborate to evict data from the shared cache. In this paper, we extend the concept of interference curves from a single task to multiple tasks executing on the same core using the *max-plus algebra*. This paper focuses on non-preemptive scheduling, as this application has not been explored previously. Interference curves have been applied to preemptively scheduled systems in Fischer and Falk (2024). Non-preemptive scheduling is of interest for real-time systems as it avoids context switching costs. In the presence of caches, each context switch creates *cache-related preemption delay* (Altmeyer and Maiza Burguière 2011). In particular for the multi-level cache hierarchy considered in this paper, there exist additional *indirect effects of preemptions* that increase the context switching costs (Chattopadhyay and Roychoudhury 2014). Non-preemptive scheduling avoids these delays.

The key contributions of this paper are:

- Formulation of an ILP model to derive event-arrival curves for inter-core cache interference by a single task.
- Modelling of non-preemptive scheduling using interference curves.
- Design of a data-flow analysis that leverages timing information to classify shared cache accesses as cache hits or potential misses using interference curves.
- Correctness proofs for all components of the presented analysis.
- An evaluation demonstrating the scalablility and significant improvements over previous methods (Hardy et al. 2009; Nagar and Srikant 2016).

The remainder of this paper is structured as follows: in Sect. 2, related research is discussed. Section 3 discusses the system model and explores existing analyses for shared caches. Section 4 describes the general workflow of the presented analysis. We introduce the event-arrival perspective on cache interference in Sect. 5. In Sect. 6 accesses to the shared cache are classified as cache hits or potential misses by applying the interference curves in a data-flow analysis. An evaluation using realistic workloads is presented in Sect. 7. Section 8 concludes the paper.

2 Related work

For over two decades, many different facets of cache behavior and cache-related delays have been analyzed. Ferdinand and Wilhelm (1999) introduced *may* and *must* analyses that determine whether data will potentially or definitely be present in the cache. The concrete state of the system is abstracted using *abstract interpretation* (Cousot and Cousot 1977) to the minimal and maximal age of a cache block. Twenty years later, Touzeau et al. (2019) presented an abstraction based on zero-suppressed binary decision diagrams that allows for a precise analysis of may/must information.

The may and must analyses are *qualitative* analyses. Each individual access to the cache is analyzed and classified as either a hit, a miss, or unknown. In contrast, *quantitative* analyses provide an upper bound on the total delay due to some cache behavior. One such quantitative analysis is the *persistence* analysis, which determines whether a set of accesses can result in at most one cache miss (Ferdinand and Wilhelm 1999; Reineke 2018).

A survey of multi-core analysis techniques was published by Maiza et al. (2019), whereas cache specific analyses were surveyed by Lv et al. (2016).

Cache-related preemption delay (CRPD) occurs when multiple tasks are assigned to a single core and preemptions are allowed. CRPD for single-level caches has been studied extensively (Lee et al. 1998; Altmeyer and Maiza Burguière 2011). Chattopadhyay and Roychoudhury (2014) presented the first CRPD analysis for a two-level cache hierarchy, focusing on non-inclusive caches. Zhang and Koutsoukos (2016) developed an analysis for inclusive cache hierarchies. The more recent work by Rashid et al. (2022) introduced the concept of L1-UCBs and L2-UCBs to compute CRPD in a two level non-inclusive cache hierarchy.

Xiao et al. (2017, 2022) integrated the effects of inter-core interference from shared caches directly into the schedulability analysis for non-preemptive systems. They compute an interference penalty by multiplying the number of accesses to potentially evicted cache blocks by a cache miss penalty.

Yan and Zhang (2008) and Zhang and Yan (2009) analyzed direct-mapped shared instruction caches in multi-core systems. Yan and Zhang (2008) differentiate between accesses which are contained in loops and those which are not in loops. An L2 cache hit is degraded to a cache miss if an interfering task accesses the same cache set. Zhang and Yan (2009) models inter-core interference in an ILP to determine the worst-case execution time.

Hardy et al. (2009) analyzed set-associative shared caches in multi-core systems by counting the number of potentially interfering cache blocks. This value is called the *cache block conflict number* (CCN). An access to a cache block is classified as a hit if the number of conflicting blocks is less than the associativity minus the block age. Liang et al. (2012) extended this approach by incorporating a lifetime analysis. The lifetime analysis determines which tasks are running concurrently, as tasks with a disjoint lifetime do not create any mutual interference on the cache. While this approach yields safe results, it is pessimistic as it assumes that an interfering task can potentially issue all interfering accesses concurrently at any point in time.

Kelter (2014) analysed shared caches by evaluating all possible state combinations of tasks executing on different cores. Due to the high analysis overhead, this approach proved to be feasible only for small systems.

Zhang et al. (2022) aimed to reduce the pessimism of the CCN analysis by excluding infeasible interferences. Memory accesses are grouped based on their location in the control-flow graph. This creates a *happens-before* partial order on all accesses contained in a task. Using this ordering, infeasible combinations of interfering accesses are excluded from the interference estimation. The additional execution time caused by cache misses due to interference is computed using the cumulative execution count of all accesses to potentially evicted cache blocks. This upper bound for the additional execution time is named *worst-case-extra-execution-time* (WCEET). They evaluate the approach for dual-core systems and compare it to the CCN approach. The approach is evaluated for a single instance of a single co-running task. Using the MRTC benchmark suite (Gustafsson et al. 2010), an average WCET reduction of 13% is reported. However, in the median only a 1% improvement is achieved.

A similar approach was used by Dharishini and Murthy (2021) to analyze multi-threaded programs running on multi-core systems. Synchronization points between threads are used to determine which sections of the program may run in parallel in different threads. This information is then used to reduce the number of potentially occurring conflicts.

Nagar and Srikant (2014, 2016); Nagar (2016) approach the cache interference problem from a different perspective. They developed a shared cache analysis by capturing inter-core interference in an ILP model. The total amount of possible interfering accesses originating from competing tasks is statically determined and used to limit the interference experienced by the analysed task. The worst-case distribution of interfering accesses along the control-flow graph (CFG), which results in the largest increase in execution time, is then determined by solving the ILP. The analysis thus determines the *worst case interference placement* (WCIP). This approach does not depend on the exact interleaving of memory accesses issued by different tasks and yielded more precise WCET estimations than the CCN classification method. In addition to the precise ILP model, an approximate algorithm is presented, which exhibits similar precision but lower analysis overhead compared to the ILP model. In this paper, we pursue an orthogonal approach. Instead of limiting the total number of interfering accesses, we examine how quickly a sequence of multiple accesses may be issued.

The two techniques presented by Zhang et al. (2022) and Nagar (2016) do not produce a hit or miss classification for any single cache access but only bound the overall increase in execution time for the complete program. That is, they are quantitative analyses. The problem of classifying each individual access is more complex than determining the WCEET in the sense that given a classification of individual accesses, a safe WCEET value can be determined, but the inverse is not possible. In this paper, we tackle the harder problem of classifying each individual access as a cache hit or potential miss.

Oehlert et al. (2018); Oehlert (2021) presented a method to quantify memory accesses issued by a task using event-arrival curves. Memory access events are derived from the program code at the level of basic blocks. To compute an upper bound on the number of events arriving in a given time frame, an ILP model using *implicit path enumeration* is developed. The event-arrival curves are used to analyze, and subsequently improve, the bus contention in multi-core systems. Based on this work, we develop an ILP model to quantify inter-core cache interference. Instead of quantifying the total number of memory accesses, we analyze how much time is required for a task to access a given number of interfering cache blocks.

3 System model & background

In this section, we discuss the assumptions made by the analysis on the system architecture and describe existing analysis approaches to establish the current landscape of shared cache analysis.

3.1 System model

The system architecture considered in this paper consists of multiple cores with private L1 caches, which are connected to a shared L2 cache via a shared bus. The shared cache uses the least-recently-used (LRU) replacement policy. The analysis is applicable to set-associative shared caches employing the LRU replacement policy with associativity $\mathcal{A}$. As cache sets operate independently of one another, we may consider them in isolation during the analysis. We concentrate on instruction caches in this paper because the memory layout of the program code is known at compile time. However, the analysis approach is general and can be applied to data, instruction, and unified caches.

The analysis requires that the minimal and maximal blocking time for an access over the shared bus is known and finite, e.g. due to round-robin arbitration.

We denote the set of all cores by $\mathcal{C}$. In the remainder of the paper, we use the convention that $C \in \mathcal{C}$ represents the core of the task currently under analysis, while $C' \in \mathcal{C} \setminus \{C\}$ represents an interfering core. The set of tasks executing on a core C are denoted by T_C. We use the convention that $\tau \in T_C$ refers to the task under analysis and $\varphi \in T_{C'}$ refers to an interfering task. An instance of a task is called a job.

In our formalization, we differentiate between the sets $T_{C'}, C' \in \mathcal{C} \setminus \{C\}$ to support partitioned scheduling. Partitioned scheduling refers to the fact that the mapping of a task to the executing core does not change. However, the analysis is applicable both to partitioned and non-partitioned scheduling of tasks. When applying the analysis to non-partitioned scheduling, all tasks $\varphi \neq \tau$ can cause shared cache interference for τ. This needs to be considered when determining the interference curves, which means that the set of tasks potentially executing on a core C' needs to be updated to $\bigcup_{C \in \mathcal{C}} T_C \setminus \{\tau\}$. We note that, due to its higher predictability, partitioned scheduling will likely lead to more precise analysis results.

We impose no restrictions on the order in which tasks are executed, i.e., there are no execution order dependencies between tasks. Each task τ may be executed repeatedly, with a minimal inter-arrival time of $Period(\tau)$ cycles.

The pair (V_τ, E_τ) represents the control-flow graph (CFG) of τ. The set V_τ contains the nodes in the graph, whereas the edges between nodes are contained in the set E_τ.

The set of accesses performed by a task τ to the shared cache are denoted by Acc_τ, whereas the set of accessed cache blocks is denoted by $\mathcal{B}_\tau$. Each access targets a particular cache block, which is given by the function $cb : Acc_\tau \rightarrow \mathcal{B}_\tau$.

Multiple cache accesses may be associated to each node $v \in V_\tau$. However, it is required that the accesses associated to a node are ordered, i.e., they will be issued in a specific order when executing v. This is necessary, as the conflicts occurring between these accesses has to be known during the analysis. This restriction may prohibit speculative execution and out-of-order execution. These features are problematic for a cache analysis, because it is no longer possible to statically predict which accesses are performed or in which order they happen. This is a problem for WCET analyses in general and not just limited to the presented analysis.

The notation introduced above is summarized in Table 1.

Table 1 General notation

Symbol	Meaning
$\mathcal{C}$	Set of all cores
T_C	Set of tasks assigned to core C
$\tau \in T_C$	The task for which to analyze the shared cache
$\varphi \in T_{C'},\ C \neq C'$	A task causing interference at the shared cache
$\mathcal{A}$	Associativity of the shared cache
Acc_τ	Set of all shared cache accesses from τ
$\mathcal{B}_\tau$	Set of all cache blocks of τ
$cb : Acc_\tau \rightarrow \mathcal{B}_\tau$	Mapping of an access to the targeted cache block
$Period(\tau)$	Period of task τ
(V_τ, E_τ)	Nodes and edges in the control-flow graph of task τ

3.2 Background on shared cache analysis

In this section, we will give an introduction into cache analysis and analysis of inter-core interference for shared caches.

Ferdinand and Wilhelm (1999) introduced the *May* and *Must* analyses for LRU caches. These analyses perform a data-flow analysis on the control-flow graph of the analyzed task τ to determine the minimal and maximal age of every cache block at each program location.

The *Must* analysis computes the maximal number of conflicting cache blocks accessed since the last access to a given cache block. As we focus on a single cache set at a time, the *Must* information at particular program location can be represented by a mapping $Must : \mathcal{B}_\tau \rightarrow \{0, \ldots, \mathcal{A} - 1\} \cup \{\infty\}$, where ∞ indicates that the block is definitely not contained in the cache. A mapping $Must(b) = n < \mathcal{A}$ shows that at most n conflicting cache blocks have been accessed by the analyzed task since the last access to b. Consequently, an access to the cache block b would result in a cache hit.

The difference between the maximal age of a cache block and the associativity of the cache is called the *resilience*, which corresponds to the minimal amount of additional interference for which a cache miss may occur.

The *May* analysis is the analog to the *Must* analysis in that it computes the minimal age of cache blocks and can be used to deduce definite cache misses. We denote the minimal age of a cache block by the mapping $May : \mathcal{B} \rightarrow \{0, \ldots, \mathcal{A} - 1\} \cup \{\infty\}$. For an in depth description of the *Must* and *May* analyses see Ferdinand and Wilhelm (1999). To differentiate between the information regarding the first and second level cache, we use a superscript, i.e., $May^l, Must^l$ for $l \in \{1, 2\}$.

The *May* and *Must* information of the L1 cache can be used to determine the *cache access classification* (CAC) for the second level cache. The CAC indicates whether the access will reach the shared cache *always* (A), *never* (N), or whether the behavior is *uncertain* (U) (Hardy and Puaut 2008). This is represented by the function: CAC $: Acc_\tau \rightarrow \{A, N, U\}$. Assuming the access is always performed on the L1 cache, the CAC for the L2 cache is defined as follows:

$$\mathrm{CAC}(a) = \begin{cases} A & \text{if } May^1(cb(a)) = \infty \\ N & \text{else if } Must^1(cb(a)) < \infty \\ U & \text{otherwise} \end{cases}. \tag{1}$$

As noted above, without any additional interference, an access will definitely result in a cache hit, if the *Must* age of the targeted cache block is less than the cache's associativity. For shared caches, the classification process must account for inter-core interference in addition to intra-task interference. An intuitive approach is to consider all potentially interfering cache blocks as actually interfering with the analyzed access. The number of such interfering blocks is called the *cache block conflict number* (CCN). When deriving cache hit classifications, the associativity of the shared cache is effectively reduced by the conflict number. This approach was presented by Hardy et al. (2009) and later refined by Liang et al. (2012) to consider only tasks that may execute in parallel. The pessimism in this approach is apparent as interfering tasks may not access all potentially interfering cache blocks simultaneously, repeatedly, and at any point in time.

Formally, the interference caused by a task $\varphi \in T_{C'}$ is defined as:

$$CCN_\varphi = |\mathcal{B}_\varphi| = |\{cb(a) \mid a \in Acc_\varphi\}|. \tag{2}$$

An access $a \in Acc_\tau$ issued by task τ running on core C to the shared cache is classified as a cache hit if and only if:

$$Must^2(cb(a)) + \sum_{C' \neq C} \sum_{\varphi \in T_{C'}} CCN_\varphi < \mathcal{A}. \tag{3}$$

Instead of classifying accesses to the shared cache individually, Nagar and Srikant (2014, 2016); Nagar (2016) analyzed shared cache interference by computing an upper bound on the additional delay. The actual WCET of a task in a multi-core environment is calculated as the sum of the WCET without interference and the extra execution time due to inter-core interference.

Next, we will provide an overview of the approach, which works by solving the so-called *worst-case interference placement* (WCIP) problem. It is important to note that the approach is only formulated for a single instance of interfering tasks in the existing literature (Nagar and Srikant 2014, 2016; Nagar 2016). We will also introduce an extension of the approach that allows for the analysis of multiple interfering jobs from the same task after explaining the basic approach.

The goal of the approach is to find the worst-case distribution of interfering accesses in the control-flow graph of the analyzed task, resulting in the largest increase in execution time. The number of interfering accesses is determined beforehand from the control-flow graph of the interfering tasks. We call this value the *interference budget*. For a single instance of an interfering task, this interference budget can be determined by solving an ILP that maximizes the number of accesses to the shared cache. This method uses the *implicit path enumeration technique* (IPET) as presented in Li and Malik (1997).

Nagar and Srikant present an ILP that provides a precise and safe solution to the problem and an approximate algorithm, which also provides a safe result. The approximate algorithm is comparable in precision to the ILP solution but requires much lower computational effort (Nagar 2016). In the following, we will focus on the approximate algorithm to solve the WCIP problem.

The algorithm approximates the problem by assuming that all accesses to the shared cache of the analyzed task lie on a singular path. These accesses are then ordered by the cache age of the targeted cache block. Accesses with a high age in the cache are easily evicted and are placed at the front of the queue. The algorithm then greedily removes accesses from the front of the queue. For each removed access, the interference budget is reduced according to the resilience of the targeted cache block, and the interference penalty is increased by one cache miss. The algorithm stops when the budget is depleted or all accesses have been converted to cache misses.

A single interfering access can contribute to the eviction of multiple cache blocks. For this reason, the interference budget is multiplied by the *overlapping factor*. The overlapping factor is the maximum number of accesses that can be affected by a single interfering access. This is the second approximation performed by the algorithm. The overlapping factor can be determined by a data-flow analysis, which keeps track of the maximal number of overlapping cache hit paths. A cache hit path of an access is a path along which the access will experience a cache hit Nagar and Srikant (2014).

The pseudocode for the approximate WCIP algorithm is shown in Algorithm 1. In the algorithm, the total interference penalty I is computed by computing a penalty for each cache set s. The value B_s denotes the interference budget, i.e., the number of interfering accesses, and $CCN(s)$ is the sum of conflicting cache blocks from all interfering tasks. The value $numhits_s^k$ corresponds to the number of cache accesses resulting in a cache hit, where the targeted cache block has a resilience of k. This means that for $numhits_s^k$ accesses, the access may result in a cache miss after at least k interfering accesses. Consequently, the maximal age of the targeted cache block is $\mathcal{A} - k$. The value o_s is the overlapping factor as described above.

Algorithm 1 Compute the approximate WCIP penalty (Nagar and Srikant 2016).

```
Input: For each cache set $s$:
interference budget $B_s$, the cumulative CCN for cache set $s$: $CCN(s)$,
the number of shared cache accesses $numhits_s^k$ with resilience $k$,
the overlapping factor $o_s$.
Output: Worst-case cache interference penalty $I$
$I \leftarrow 0$
for all cache sets $s$ do
    $B_s \leftarrow o_s \cdot B_s$
    for $k \leftarrow 1, \ldots, \mathcal{A}$ do
        if $CCN(s) \geq k$ then
            if $B_s \leq k \cdot numhits_s^k$ then
                $I \leftarrow I + (\lceil \frac{B_s}{k} \rceil \cdot L2_{miss})$
                $B_s \leftarrow 0$
            else
                $I \leftarrow I + (numhits_s^k \cdot L2_{miss})$
                $B_s \leftarrow B_s - (numhits_s^k \cdot k)$
            end if
        end if
    end for
end for
```

The evaluation in Nagar (2016) reports WCET improvements for 9 out of 27 tasks from the MRTC benchmark suite (Gustafsson et al. 2010). In the evaluation, interference is generated by a single instance of the task *nsichneu*.

Note that this algorithm assumes that the number of interfering accesses can be determined a priori. This value is then used to compute the final WCET estimate of a task. However, when tasks are executed repeatedly, a single interfering task φ may be executed multiple times during the execution of the analyzed task τ. Thus, the interference budget actually depends on the maximal execution time.

Consider the following example for clarification. Let $\text{WCET}_0(\tau)$ be the time required to execute τ without any inter-core interference and let the interfering task φ be activated periodically with a minimum inter-arrival time of $\mathit{Period}(\varphi)$ cycles. φ may be executed up to $\lceil \frac{\text{WCET}_0(\tau)}{\mathit{Period}(\varphi)} \rceil + 1$ times in a time frame of $\text{WCET}_0(\tau)$ cycles.

To get the total interference budget from φ, the execution count of φ has to be multiplied by the number of interfering accesses per execution of φ. The result can then be used to compute the interference penalty I_0 using the WCIP approach.

Let $\text{WCET}_1(\tau) = \text{WCET}_0(\tau) + I_0$. Due to the increased execution time, the interfering task φ may be activated more often. I.e., it is possible that the following holds:

$$\left\lceil \frac{\text{WCET}_0(\tau)}{\mathit{Period}(\varphi)} \right\rceil < \left\lceil \frac{\text{WCET}_1(\tau)}{\mathit{Period}(\varphi)} \right\rceil. \tag{4}$$

Consequently, the interference budget increases due to the higher execution time, and it is necessary to repeat the computation of the penalty value of the WCIP approach using an increased interference budget. For this reason, we make the following observation:

Observation 1 When using the WCIP approach to compute the WCET of a task, the WCIP problem has to be solved iteratively, as the number of interfering accesses is not known a priori. The iteration can be stopped when the resulting WCET value has converged.

Without this iterative application, the WCIP approach is only applicable to systems where each task is executed at most once. To our knowledge, this dependence between repeated task executions and the WCIP approach has not been discussed previously. We compare the approach presented in this paper to the iterative WCIP approach in Sect. 7.

We will now discuss another approach to analyze conflicts in shared caches. Zhang et al. (2022) present a different quantitative analysis approach. In their approach, infeasible cache conflicts are eliminated from the interference computation by considering the order in which conflicts have to occur.

This is done by transforming the control-flow graph of the analyzed task and the interfering task into an *unordered region* (UR) control-flow graph. An unordered region is defined to be a cache access that is not contained in a loop, or sets of cache accesses contained in an out-most loop. The unordered regions are the nodes of the UR-CFG. The key property of the UR-CFG is that it is loop free. Thus, when the control of the program has advanced to another UR, it may never return to a previous UR. All possible paths through the UR-CFG are enumerated and are named *constant execution order path*s (CEOP). Due to its reliance on the constant execution order paths, we refer to this method as the CEOP analysis approach.

It is now possible to detect infeasible conflicts in the shared cache as the conflict pairs between unordered regions can be mutually exclusive. Intuitively, when an access at the start of τ experiences interference in the shared cache from an access near the end of φ, an access performed later by τ will never conflict with an access performed by φ at the beginning. Consider two tasks τ and φ consisting of two unordered regions (UR_1^τ, UR_2^τ) and $(UR_1^\varphi, UR_2^\varphi)$, respectively. When an access contained in UR_1^τ experiences interference from an access contained in UR_2^φ, the accesses contained in UR_2^τ will not conflict with accesses contained in UR_1^φ. Note that this obviously only holds for a single instance of the interfering task φ.

The approach then computes the maximal number of shared cache accesses that are subject to inter-core interference. The interference penalty is determined by multiplying the number of affected accesses with the L2 cache miss penalty. To model multiple instances of an interfering task φ, the penalty is multiplied by the number of jobs from φ that may execute in parallel to the analyzed task τ.

The evaluation performed in Zhang et al. (2022) compares the analysis precision of the CEOP analysis to the CCN approach for a single interfering task instance. The evaluation is performed using the MRTC benchmark suite (Gustafsson et al. 2010). In total 169 different task pairings are evaluated. For 63 (37.3%) task pairings no WCET improvement is achieved. For a total of 92 (54.4%) task pairs the improvement is $\leq 1\%$. On average, the WCET is reduced by 13%.

The number of interfering instances is assumed to be known in Zhang et al. (2022). This means that, like noted for the WCIP approach in Observation 1, an iterative computation is necessary to compute the final WCET value. Again, to the best of our knowledge, the iterative nature of the approach for repeated task executions has not been discussed before.

We note that while the interference computed by the CEOP approach, as described above, scales linearly with the number of conflicting jobs, the WCIP approach scales sub-linearly. I.e., in the WCIP method, the penalty occurring due to two instances of an interfering task is less or equal to two times the penalty of a single instance. This follows directly from the construction of the WCIP problem. The number of interferences required to trigger an additional cache miss increases monotonically until all shared cache accesses are considered cache misses. We capture this fact in the following observation:

Observation 2 The WCIP approach scales better than the CEOP approach, when the number of interfering cores / tasks / jobs increases.

The respective evaluations of the WCIP approach and the CEOP approach assume a single interfering job. The improvement over the baseline CCN analysis will be smaller when considering repeated task activations. As the median WCET improvement for the MRTC benchmark suite is 0% and 1%, for WCIP and CEOP respectively, it is expected that these quantitative approaches perform similar to the CCN analysis for the periodic task model considered in this paper. We compare the precision of the WCIP approach to the presented analysis, which solves the harder problem of classifying individual accesses, in Sect. 7.

4 Analysis overview

We will now give an overview of the analysis presented in this paper. The aim of the analysis is to classify each access to the shared cache as either a cache hit or a potential cache miss. This means that the presented analysis is a qualitative analysis, unlike the WCIP and CEOP approaches, which are quantitative analyses and only provide an upper bound on the total delay due to cache sharing.

The key idea of the proposed analysis is to analyze the timing of accesses to the shared cache. At the start of the analysis, we determine how quickly an interfering task can issue conflicting accesses to the shared cache. Then, we determine for each access from the analyzed task, how long ago the requested data was inserted into the shared cache. Using this information, it is possible to determine whether interfering tasks running on other cores could have evicted the accessed data prior to the analyzed access. The analysis approach is divided into the following five steps:

1. Initial best-case and worst-case execution time analysis for each task.
2. Cache interference analysis from individual tasks using event-arrival curves.
3. Bounding interference for non-preemptively scheduled task sets.
4. Data-flow analysis to determine cache hit classifications.
5. Final WCET analysis using the new hit classifications.

We will now give a detailed description of the five analysis steps.

Step 1**:** To perform the shared cache analysis, we require both worst-case and best-case timing information on a basic block level. Thus, as the first step, an isolated timing analysis is conducted for each task. To compute WCET estimates, accesses to the shared cache are considered to be cache misses. The best-case execution time (BCET) is computed using a classical age-based abstract domain as presented by Ferdinand and Wilhelm (1999), without considering the impact of inter-core interference. The BCET analysis assumes that tasks are not sharing code across cores. Otherwise, a task running on another core could effectively prefetch data into the shared cache, while the analyzed task is performing some other action, such as performing computations using data stored in the private L1 cache. This would invalidate the BCET estimate. In order to allow for sharing code across cores, the BCET analysis could assume that every access to shared code in the shared cache results in a cache hit.

Step 2**:** Following the initial timing analysis, the event-arrival curves of cache access events originating from interfering tasks $\varphi \in T_{C'}$ are derived. More precisely, it is determined how much time has to pass for φ to access a particular number of interfering cache blocks by solving an ILP model. The ILP model contains a parameter to specify the required level of interference. As the replacement algorithm for each cache set operates independently, the ILP model is solved repeatedly for all cache sets and interference levels from 1 to the cache associativity. The solutions of the ILP for the different parameters give rise to the interference curve of φ. During this step, the BCET information is used to arrive at a safe upper bound on the interfering traffic on the shared cache. This step is discussed in detail in Sect. 5.2.

Step 3**:** The previous step yields interference information for all interfering tasks $\varphi \in T_{C'}$. However, when multiple tasks are assigned to execute on a core C', these tasks can collaborate to evict data from $\tau \in T_C$. In the third step, we extend the concept of interference curves from individual tasks to a set of tasks running on a single core C'.

We analyze the impact of non-preemptively scheduling tasks on an interfering core C' by evaluating potential sequences of task executions on C' and performing a convolution of the associated event-arrival curves in the max-plus algebra. This allows us to compute a safe upper bound on the total interference created by an interfering core executing multiple different tasks in a non-preemptive fashion. This step is explained in Sect. 5.3. The total inter-core interference experienced by a task $\tau \in T_C$ is then given as the sum of the interference caused by other cores $C' \in \mathcal{C} \setminus \{C\}$.

Step 4**:** Based on the information from the previous analysis steps, a backward *data-flow analysis* (DFA) is performed on the control-flow graph of the analyzed task (Aho et al. 2007, p. 597ff). The DFA investigates all accesses which potentially result in a cache hit. An access is considered as a *potential-hit*, if it results in a cache hit without any inter-core interference. For each *potential-hit* there exist corresponding cache accesses which initially caused the relevant cache block to be loaded into the shared cache. The DFA determines the maximal duration between these pairs of cache accesses, which load the block into the shared cache and subsequently access it again. Additionally, it also keeps track of any intra-task interference. By evaluating the interference curves of interfering cores for the maximal load-access path duration, the inter-core interference can be safely bounded. This means that potential-hits can be classified as either definite cache hits or potential cache misses. The foundations for hit classifications based on timing information are presented in Sect. 6.1 and the data-flow analysis is presented in Sect. 6.2.

Step 5**:** We use the access classifications computed in the previous step in a WCET analysis to determine the final WCET for each task. The results achieved in this WCET analysis are evaluated in Sect. 7.

5 Interference curves for shared cache interference

This section deals with the first key component of the timing-aware cache interference analysis presented in this paper. In Sect. 5.1, we introduce the concept of interference curves and motivate their use in the analysis of shared cache interference. The following Sect. 5.2 corresponds to the second step of the analysis as described in the previous section. In it, we show how interference curves can be derived from the control-flow graph of a task using best-case execution time information. Finally, in Sect. 5.3, the third step of the analysis is discussed. The interference curves of multiple tasks are combined to compute the total interference resulting from a non-preemptively scheduled set of tasks.

5.1 Definition and motivation of interference curves

We formally introduce interference curves and motivate why it is beneficial to analyze shared cache interference from this perspective in this section.

Traditionally, shared cache interference has been quantified by counting the number of cache blocks that an interfering task may load into the shared cache. The

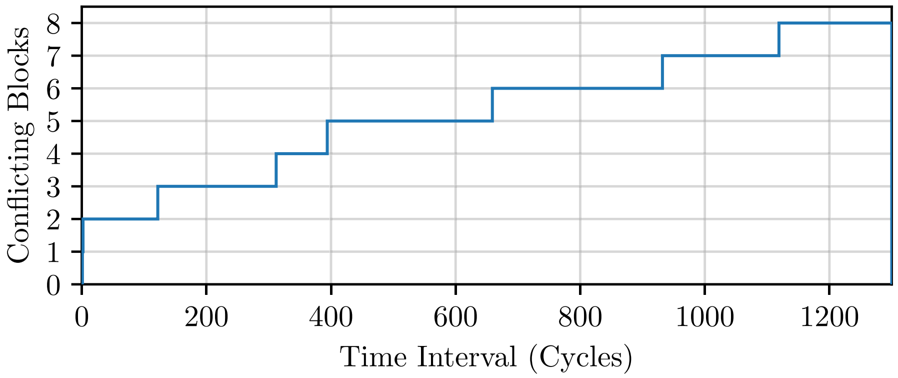

Fig. 1 Example of an event-arrival curve expressing shared cache interference. Derived from the `aiifft01` benchmark of the EEMBC AutoBench suite

pessimism of this approach becomes apparent when the time that passes between accesses to the conflicting blocks is considered.

Under the LRU replacement policy, cache blocks are ordered using the least-recently-used property. Consequently, the age of a cache block $b \in \mathcal{B}_\tau$ corresponds to the number of different cache blocks, which were stored in the cache since the last access to b. It follows that multiple accesses to the same conflicting cache block do not cause further aging of b in the cache, as the set of conflicting blocks does not change.

Using this observation, we define the notion of an *aging-event for a cache-block* in the context of shared cache interference as an access by an interfering task, which causes the age of an analyzed cache block to increase. Consequently, multiple events correspond to multiple accesses targeting a set of distinct cache blocks.

Definition 1 (Interference curve) An interference curve η_φ is a function, which maps a time frame of Δt cycles to the maximal number of cache blocks accessed by $\varphi \in T_{C'}$: $\eta_\varphi : \mathbb{N}_0 \rightarrow \mathbb{N}_0$, where $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

A mapping $\eta_\varphi(\Delta t) = n$ means that there exists a scenario in which φ will access n different cache blocks during a time frame of Δt cycles. The statement $\eta_\varphi(\Delta t) = n$ is thus equivalent to the fact that there exists an initial hardware state and path in the CFG of φ that causes accesses to n distinct cache blocks under the constraint that the best-case duration of the path is less or equal to Δt cycles. Note that we refer to the best-case execution time here, as the fastest execution of an interfering task causes the worst-case interference for the analyzed task.

Figure 1 shows an example for such an interference curve. The x-axis shows a time interval, and the y-axis represents how many different cache blocks may be accessed. In this example, the interference grows quickly in the beginning. Two interfering accesses may happen almost instantaneously. After a time frame of 122 cycles, the task may have issued accesses to 3 different cache blocks. However, the time interval required to observe a larger amount of interference is significantly higher. To observe interference from 8 conflicting blocks takes 1119 cycles. This example curve has been derived from the `aiifft01` benchmark from the EEMBC benchmark suite (Poovey et al. 2009), with a shared cache size of 4 KB using the approach presented in the upcoming subsection.

This clearly demonstrates the pessimism in the currently available analysis techniques (Liang et al. 2012; Nagar 2016; Zhang et al. 2022). All these techniques assume that every conflicting block may be accessed instantaneously by an interfering task. However, we can see here that it takes a substantial amount of time for the interfering task to generate the traffic on the shared cache. Accounting for this delayed onset of interference after storing data in the shared cache is the key idea behind the timing-aware shared cache analysis presented in this paper.

5.2 Deriving interference curves of a task

This section shows how an interference curve, as seen in Fig. 1, can be derived from a task's control-flow graph.

To derive an event arrival curve from the CFG of a task φ, we have to associate cache accesses to the nodes in the CFG (V_φ, E_φ). For data caches, this relation arises from the memory-accessing instructions contained in the program and their respective target addresses. For instruction caches, cache accesses originate from fetching operations in the processor pipeline, so we have to consider the pipeline behavior.

Let $\rightsquigarrow \subset V_\varphi \times \mathcal{B}_\varphi$ be a relation that contains the pair (v, b) iff executing the node $v \in V_\varphi$ may cause an access to the block $b \in \mathcal{B}_\varphi$. A path π is a sequence of nodes $\pi = (v_1, \ldots, v_p)$ with $(v_i, v_{i+1}) \in E_\varphi,\ i \in \{1, \ldots, p-1\}$. The execution of a path π in the CFG of φ causes another task τ to experience n interfering cache accesses:

$$n = \left| \bigcup_{v \in \pi} \left\{ b \in \mathcal{B}_\varphi \mid v \rightsquigarrow b \right\} \right|. \tag{5}$$

Using this relation, we derive the interference curve of a task φ. Note that η_φ is a step function. Furthermore, we are only interested in the arrival of the first $\mathcal{A}$ events, as after $\mathcal{A}$ events all previously cached blocks are evicted from an LRU cache. Thus, we can capture a task's cache access behavior by deriving the minimal time required to access $1,\ 2,\ \ldots,\ \mathcal{A}$ distinct cache blocks. These values correspond to the location of the steps in Fig. 1, where the interference jumps upwards.

To compute the minimal time required for a particular number of events, we utilize an ILP model based on the *implicit path enumeration technique* (Li and Malik 1997). We base the model on the variant presented by Oehlert et al. (2018), which evaluates bus contention in a multi-core system. As the basic construction of the ILP model is out of the scope of this paper, we focus only on the additional constraints required to model cache interference. In the following, we present the variables and constraints introduced into the model to capture inter-core cache interference. The notation is summarized in Table 2.

To decide whether a particular cache block will contribute to the interference, we introduce a binary decision variable for each cache block. This decision variable indicates whether the cache block is accessed on the considered path. The variables are denoted by t_b for $b \in \mathcal{B}_\varphi$. The indicator variables t_b are constrained by the constraints given in Eq. (6) and Eq. (7), where q_v is the variable containing the execution count of v.

$$0 \leq t_b \leq 1 \tag{6}$$

$$t_b \leq \sum_{v:\ v \rightsquigarrow b} q_v \tag{7}$$

Thus, if any node v is executed which may access the block b, the indicator variable t_b may take the value 1, otherwise it is set to 0. The number of variables t_b depends on the number of cache blocks potentially accessed by φ. For an instruction cache, this scales linearly with the code size of the analyzed task φ and is inversely proportional to the cache line size.

To determine the time frame during which n cache blocks may be accessed, only paths containing accesses to at least n cache blocks are considered:

$$n \leq \sum_{b \in \mathcal{B}_\varphi} t_b\ . \tag{8}$$

In summary, the cache interference modeling requires an additional binary variable for each cache block (see Eq. 6), a corresponding constraint (see Eq. 7), and a constraint enforcing the required level of interference (see Eq. 8). The size of the ILP model is thus similar to the size of the underlying IPET formulation (Oehlert et al. 2018), with the additional constraints growing linearly in the code size of the analyzed task.

In the model, n is a parameter which may be set to values in $\{1, \ \ldots, \ \mathcal{A}\}$ to determine the minimal cycle count required for different levels of interference.

Consequently, the objective of the ILP is to minimize the number of cycles required to cause accesses to n distinct cache blocks, as shown in Eq. (9). The path duration is computed as the sum of the execution time contributions z_v of each node $v \in V_\varphi$:

$$\text{minimize} \sum_{v \in V_\varphi} z_v\ . \tag{9}$$

Table 2 ILP variables

Symbol	Meaning
$q_v \in \mathbb{N}$	Execution count of a node $v \in V_\varphi$
$z_v \in \mathbb{N}$	Execution time contribution from $v \in V_\varphi$
$t_b \in \{0, 1\}$	Indicates whether a conflicting block $b \in \mathcal{B}$ is accessed
$s_v \in \{0, 1\}$	Indicates whether $v \in V_\varphi$ is the starting node of the path
$e_v \in \{0, 1\}$	Indicates whether $v \in V_\varphi$ is the final node of the path

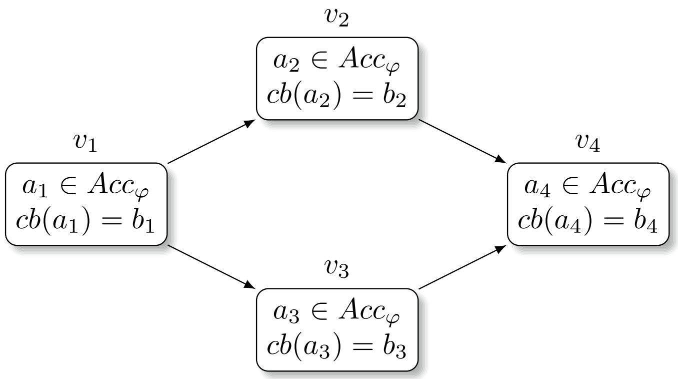

Fig. 2 Example CFG containing four nodes $v_1, v_2, v_3, v_4 \in V_\varphi$. Nodes contain accesses $a_1, a_2, a_3, a_4 \in Acc_\varphi$, and target cache blocks $b_1, b_2, b_3, b_4 \in \mathcal{B}_\varphi$, respectively. The accesses a_2 and a_3 are mutually exclusive. The maximal interference caused by this CFG is 3

The execution time contribution z_v depends on the best-case execution time and the execution count q_v of the node v. As we do not make assumptions about the distribution of events inside the nodes, the time contributions of the first and last nodes of the (partial) path are reduced to a single cycle.

To clarify this, consider a simple linear CFG with three nodes, v_1, v_2, v_3, as an example. Assume that each node in the graph contains an access to a single cache block and has a BCET of 100 cycles. When we want to answer the question of how quickly contents in the shared cache may age three times, only the complete path (v_1, v_2, v_3) is a viable candidate. The other partial paths in the CFG generate two or fewer events. The BCET of the complete path is 300 cycles. However, the access to the shared cache in node v_3 might not happen at the very end of processing v_3, but near the beginning of v_3. Thus, in a real execution, three events may happen already after, for example, 210 cycles. To generate a safe lower bound, we assume that accesses associated to a node will happen at the very beginning (end) of the final (first) node in the considered path. Hence, the first and last node are considered to contribute only a single cycle to the execution time. A safe lower bound on the time to observe three events for this example would thus be 102 cycles. This approximation is needed to ensure the safety of the result but also introduces some pessimism into the analysis. To combat this pessimism, the analysis can be performed at a finer scale by virtually splitting large basic blocks in the CFG into multiple smaller nodes.

Note that by determining interference on the basis of paths in the CFG, we implicitly eliminate mutually exclusive accesses from the interference computation. For example, consider the control-flow graph shown in Fig. 2. The nodes $v_1, v_2, v_3, v_4 \in V_\varphi$ each contain an access targeting a different cache block. The maximal interference caused by this CFG is 3 blocks: either the blocks $\{b_1, b_2, b_4\}$ or $\{b_1, b_3, b_4\}$ are accessed. The blocks b_2 and b_3 are never accessed together on the same path. Such mutually exclusive access behavior is therefore safely excluded from the event-arrival curves, leading to a tight estimation of the possible interference. In contrast, the standard CCN interference estimation technique simply counts the number of potentially accessed blocks and would thus conclude a maximal interference of 4.

To construct the complete interference curve for a task $\varphi \in T_{C'}$, the presented ILP is solved for all parameter values $1 \leq n \leq \mathcal{A}$. Let $\Delta t_1, \dots, \Delta t_{\mathcal{A}}$ denote the solution of the ILP for a task φ for interference levels $n = 1, \dots, \mathcal{A}$, respectively. If the task never causes n or more interfering accesses during any execution, the ILP will be infeasible for that parameter. In that case we set the value $\Delta t_n = \infty$. Using these results, the interference curve for a task φ is defined as follows:

$$\eta_\varphi(\Delta t) = \begin{cases} 1 & \text{if } \Delta t_1 \leq \Delta t < \Delta t_2, \\ 2 & \text{if } \Delta t_2 \leq \Delta t < \Delta t_3, \\ & \vdots \\ \mathcal{A} & \text{if } \Delta t_{\mathcal{A}} \leq \Delta t, \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

As noted before, the interference on different cache sets is analyzed independently of each other. Interference curves are thus derived seperately for each cache set. Assuming that a single task is assigned to each core, we can now give a safe upper bound on the total interference using the computed curves.

Theorem 1 *The interference $\gamma_\tau(\Delta t)$ at the shared cache over a time frame of Δt cycles, experienced by the task $\tau \in T_C$ from other cores $C' \neq C$, given that all cores C' only process a single task is bounded by the addition of the curves η_φ from co-running tasks $\varphi \in T_{C'}, \forall C' \in \mathcal{C} \setminus \{C\}$:*

$$\gamma_\tau(\Delta t) = \sum_{\varphi:\ T_{C'} = \{\varphi\},\ C' \neq C} \eta_\varphi(\Delta t). \tag{11}$$

Proof We prove the theorem by contradiction. Assume that there exists a task that is subject to n' interfering accesses at the shared cache over a time frame of Δt cycles with $n' > \gamma_\tau(\Delta t)$. Due to the least-recently-used replacement property, interfering accesses issued by two different cores C'_1 and C'_2 can not collaborate to create larger interference than the sum of both contributions. It follows that there must be a core $C' \neq C$ with $T_{C'} = \{\varphi\}$, that issued more than $n_\varphi = \eta_\varphi(\Delta t)$ interfering accesses. However, the solution computed for the interference parameter n_φ by the ILP gives the smallest duration of any path from φ causing at least n_φ interfering accesses. We conclude that no path causing more than n_φ interference with duration $\leq \Delta t$ exists in φ's CFG and have proven the theorem by contradiction. □

Using the cumulative interference function γ_τ, the time-to-live of a cache block $b \in \mathcal{B}_\tau$ can be expressed as a function of its *Must* age.

Definition 2 (Time-to-live) The time-to-live (TTL) of a cache block $b \in \mathcal{B}_\tau$ in the shared cache is given by the function $\text{TTL}_\tau : \{0, \dots, \mathcal{A} - 1\} \rightarrow \mathbb{N}_0 \cup \{\infty\}$:

$$\mathrm{TTL}_\tau(age) = \sup_{0 \le \Delta t} \{ \Delta t \mid \gamma_\tau(\Delta t) < \mathcal{A} - age \} \tag{12}$$

The value *age* refers to the maximal number of conflicting blocks that have been accessed by τ since the last access to b, without considering the inter-core interference, i.e., the *Must* age of b.

Note that the TTL of a block may be $+\infty$ cycles in case the interfering tasks never create sufficient interference to trigger the eviction.

5.3 Analyzing non-preemptive scheduling

The approach described in the previous section computes an interference curve η_φ based on the control-flow graph of a singular task $\varphi \in T_{C'}$. The total interference γ_τ experienced by the analyzed task $\tau \in T_C$ at the shared cache is then computed as the sum of the interference curves of all tasks $\varphi \in T_{C'}$ running on interfering cores C', as formulated in Eq. (11). This method is only applicable to systems where a single task causes interference per core. When multiple different tasks are executed on the same core, these tasks can collaborate to evict data from the shared cache.

The interference curve of one particular task may be vastly different from the interference curve of another task. To safely classify cache accesses to a shared cache in a such a system, we must determine the worst-case interference caused by the set of tasks $T_{C'}$ running on each interfering core $C' \neq C$.

In this section, we compute the interference curve resulting from multiple tasks executing on a single core under non-preemptive scheduling. This constitutes the third step of the presented analysis.

Under non-preemptive scheduling, once a task has started to execute, it will not be interrupted until it has finished its computations. This contrasts preemptive scheduling, where the scheduler may decide to interrupt the currently running task in favor of a different task. An application of interference curves to preemptive scheduling is presented in Fischer and Falk (2024).

We do not assume any precedence constraints between different tasks. I.e., the order in which tasks execute on an interfering core C' is not restricted. Consequently, a task running on core C can experience interference from all other tasks $\varphi \in \bigcup_{C' \neq C} T_{C'}$. Furthermore, we do not require a minimal inter-arrival time between two instances of the same task. This means that the interference bounds derived in this section are general and apply to all non-preemptive schedules.

This is another differentiating factor to the existing WCIP and CEOP analysis approaches by Nagar (2016) and Zhang et al. (2022). The WCIP and CEOP approaches inherently operate on the assumption that the number of interfering jobs is known. Thus, the final WCET value has to be computed in an iterative fashion using the periods of the interfering tasks (see Observation 1). The approach presented in this section makes no such assumption and the resulting interference curve is valid for arbitrary task activations patterns.

The rest of this section is structured as follows: Sect. 5.3.1 provides a brief introduction to the calculation of event-arrival curves in the max-plus algebra. We formally describe the interference computation for non-preemptive scheduling in Sect. 5.3.2 and construct an algorithm to compute the resulting interference curve in Sect. 5.3.3.

5.3.1 Operations in max-plus algebra

When multiple tasks execute on the same core, they all contribute to the interference at the shared cache. To compute the total interference curve for a set of tasks, the interference curves of the individual tasks running on the core have to be combined. This calculation is performed using the max-plus algebra.

The max-plus algebra, and its counterpart the min-plus algebra, have been used in the area of *network calculus* (Le Boudec and Thiran 2001) to analyze the traffic in a network, and in *real-time calculus* to determine the schedulability of a system (Thiele et al. 2000).

In the max-plus algebra, the operators of addition and multiplication from the standard algebra are replaced by the maximum and addition operators, respectively. A crucial operator for the calculation of interference curves is the max-plus convolution. The max-plus convolution of two interference curves η_{φ_1} and η_{φ_2} from tasks $\varphi_1, \varphi_2 \in T_{C'}$ is represented using the $\circledast$ operator:

$$(\eta_{\varphi_1} \circledast \eta_{\varphi_2})(\Delta t) = \max_{0 \le \delta \le \Delta t} \{\eta_{\varphi_1}(\delta) + \eta_{\varphi_2}(\Delta t - \delta)\}. \tag{13}$$

The max-plus convolution determines the maximal interference caused by two tasks over a total duration of Δt cycles. This calculation considers all possible distributions of the Δt cycles between the two tasks, ensuring that the worst-case interference is identified by taking the maximum interference value across all time distributions over η_{φ_1} and η_{φ_2}.

An interference curve $\eta_{\varphi}(\Delta t)$ addresses the question of how many cache blocks may cause interference after Δt cycles. However, it can be useful to view cache interference from an alternative perspective: how much time must elapse to potentially experience n or more interference events? This perspective can be taken using the pseudo-inverse of an event-arrival curve. The pseudo-inverse of an event-arrival curve is defined as:

$$\eta^{-1}(n) = \inf\{\Delta t \ge 0 \mid \eta(\Delta t) \ge n\}. \tag{14}$$

Thus, the pseudo-inverse $\eta^{-1}(n)$ gives the minimal duration over which n interfering accesses may occur and satisfies the following properties (Le Boudec and Thiran 2001):

$$\eta(\Delta t) \ge n \implies \eta^{-1}(n) \le \Delta t, \tag{15}$$

$$\eta^{-1}(n) < \Delta t \implies \eta(\Delta t) \ge n. \tag{16}$$

Furthermore, in max-plus algebra, $-\infty$ serves as the absorbing element, meaning that $\forall n \in \mathbb{N}_0 : -\infty + n = -\infty$. We use the value $-\infty$ to signal infeasible situations. Such situations arise when computing the interference created by a full execution of a task φ, but the allotted time frame is insufficient to fully execute φ according to its best-case execution time.

To eliminate infeasible situations and thus increase the precision of the final result, we introduce a window function W_t. The window function $W_t : \mathbb{N} \rightarrow \{0, -\infty\}$ maps to either 0 or $-\infty$, depending on the parameter t and the input Δt:

$$W_t(\Delta t) = \begin{cases} 0 & \text{if } \Delta t \geq t, \\ -\infty & \text{else.} \end{cases} \tag{17}$$

For example, consider the calculation: $((\eta_{\varphi_1} \circledast \eta_{\varphi_2}) + W_{100})(\Delta t)$. A window W_{100} with parameter 100 is added to the convolution of η_{φ_1} and η_{φ_2}. The resulting curve thus yields $-\infty$ for $\Delta t < 100$. By applying a window to a curve, it can be marked as infeasible for short durations. In the context of interference curves, applying W_t refines the interference computation by enforcing a minimal duration for the scenario associated to that curve.

We will use these concepts in the next section to compute a safe upper bound on the interference stemming from a core executing a set of tasks using non-preemptive scheduling.

5.3.2 Modelling non-preemptive scheduling

To apply timing-aware interference analysis to non-preemptive scheduling, we compute an interference curve that captures the worst-case interference created by multiple tasks assigned to a single interfering core. This encompasses, for example, scenarios where a task $\varphi_1 \in T_{C'}$ ends its execution and another task $\varphi_2 \in T_{C'}$ starts execution immediately afterward. Consequently, between two subsequent accesses to a cache block from task $\tau \in T_C$, multiple tasks can contribute interference for the cache block and trigger its eviction. The goal of this section is to capture all possible interference scenarios and determine the worst-case interference caused by any of these scenarios. To this end, we define the different scenarios that may occur on an interfering core C'.

Definition 3 (Execution Scenario) An execution scenario for an interfering core C' is a sequence of tasks $(\varphi_1, \ldots, \varphi_j) \in (T_{C'})^j$. Let $\mathcal{S}_{C'} = (T_{C'})^{\mathbb{N}} = \bigcup_{j \in \mathbb{N}} (T_{C'})^j$ be the set containing all execution scenarios.

The tasks in an execution scenario $(\varphi_1, \ldots, \varphi_j) \in \mathcal{S}_{C'}$ are executed sequentially in ascending order by the interfering core C'. We are interested in the behavior of C' during a cache block reuse windows, which is the time frame from the point of storing information in the shared cache up to the point where another access to that content performed. This means an execution scenario does not represent a complete execution of the system from startup to shutdown, but models the behavior of the interfering core between accesses to blocks stored in the shared cache. Thus, for an execution scenario $(\varphi_1, \dots, \varphi_j) \in \mathcal{S}_{C'}$, a cache block is stored in the shared cache by the analyzed task τ while the interfering core executes task φ_1. The access to be classified as either a cache hit or potential miss occurs while the interfering core executes φ_j.

To issue cache hit classifications that hold even in the worst case, we must compute the worst-case interference curve over all execution scenarios $\mathcal{S}_{C'}$. Let F be a function that maps an execution scenario of an interfering core C' to the associated interference curve:

$$F_{C'} : \mathcal{S}_{C'} \rightarrow (\mathbb{N}_0 \rightarrow (\mathbb{N}_0 \cup \{-\infty\})). \tag{18}$$

Note that the image of $F_{C'}$ contains functions mapping to $-\infty$, as a particular execution scenario may be infeasible for short time durations. A scenario containing a complete execution of a task $\varphi \in T_{C'}$ may not occur in a time frame that is shorter than the BCET of φ. By mapping the interference value for $\Delta t < \text{BCET}(\varphi)$ to $-\infty$, the scenario will not contribute to the maximal interference for $\Delta t < \text{BCET}(\varphi)$.

The worst-case interference created by multiple tasks on a non-preemptive scheduled core C' is given by:

$$\mu_{C'}(\Delta t) = \max_{s \in \mathcal{S}_{C'}} \{F_{C'}(s)(\Delta t)\}. \tag{19}$$

In the remainder of this section, we develop the precise definition of $F_{C'}$ to complete the formal model of shared cache interference under non-preemptive scheduling.

The contribution of a task φ to the total interference depends on the analyzed time frame Δt during which interference can occur. Additionally, it depends on the

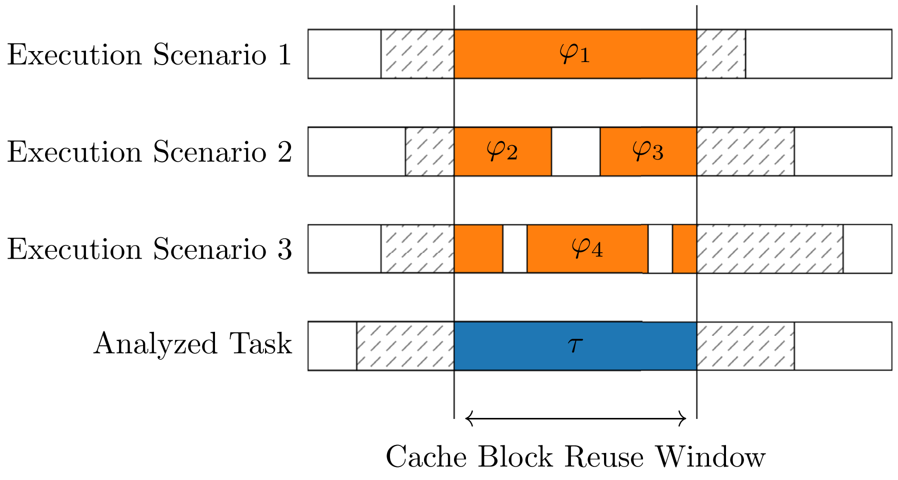

Fig. 3 Three different execution scenarios for an interfering core C'. The cache block reuse window of the analyzed task, running on core C, is marked in blue. The orange sections of the co-running tasks contribute to the shared cache interference

fact whether the co-running task φ starts and/or ends its execution within the analyzed reuse window.

Multiple different execution scenarios interfering with the reuse of a cache block are shown in Fig. 3. The time frame between the initial access to the cache block and its reuse by the analyzed task τ is shown in blue and labeled as the *Cache Block Reuse Window*. Three different execution scenarios are shown for the interfering core. The orange sections in the different scenarios indicate the source of the inter-core cache interference from co-running tasks on the core C'. The co-running tasks creating interference in the three scenarios are labeled as φ_1, φ_2, φ_3, and φ_4, respectively.

We differentiate between four different categories of interfering tasks, named as follows:

1. `single-task`: A task already executing at the start of the reuse window that does not finish executing during the reuse window (φ_1 from Fig. 3).
2. `in-task`: A task already executing at the start of the reuse window that finishes executing during the reuse window (φ_2 from Fig. 3).
3. `out-task`: A task that starts during the reuse windows and does not finish during the reuse window (φ_3 from Fig. 3).
4. `body-task`: A task that starts and finishes during the reuse window (φ_4 from Fig. 3).

We will use these categories to quantify how quickly a task may create interference. Differentiating between these different categories is useful as we may place additional constraints on the paths from which the interference curves are derived, allowing us to create a tighter estimate on the total interference. The interference curve of a task φ in one of the four categories is denoted using a superscript, i.e., η_φ^{single}, η_φ^{in}, η_φ^{out}, and η_φ^{body}.

The curve η_φ^{single} for `single-task` corresponds to the unconstrained curve as described in Sect. 5.2. An `in-task` already executes at the start of the reuse window and will finish executing during the reuse window. Therefore, the curve for an `in-task` of φ is computed considering only paths ending in a sink of φ's CFG. This is beneficial because by the end of a task's execution, the private L1 cache has warmed up, reducing the likelihood of L1 cache misses, which cause shared cache interference, compared to the task's initial execution phase. Similarly, an `out-task` begins executing during the reuse window. Thus, the η_φ^{out} curve of φ is computed considering only paths starting at the initial node of the CFG of φ. This consideration is useful as set-associative caches are divided into multiple cache sets. For example, to interfere with the third cache set, a task may first need to execute code mapped to the first and second cache set, Hence, the accesses to the third set are delayed and the interference behavior relative to the unconstrained curve η_φ^{single} improves, leading to better cache hit classification performance.

Formally, we observe that by construction $\eta_\varphi^{in}(\Delta t) \leq \eta_\varphi^{single}(\Delta t)$, and $\eta_\varphi^{out}(\Delta t) \leq \eta_\varphi^{single}(\Delta t)$ for all $\Delta t \in \mathbb{N}$. When determining the curve of a *body* task, it is guaranteed that the task will fully execute from start to finish. It is thus infeasible for the curve $\eta_{\varphi}^{body}(\Delta t)$ to contribute to interference for time frames $\Delta t < \mathrm{BCET}(\varphi)$.

Incorporating these constraints imposed by the four categories improves accuracy of the interference curves. Consequently, the interference bound can be tightened by differentiating between the stages a task is in while contributing to the shared cache interference.

Next, we integrate these constraints into the curve derivation ILP presented in Sect. 5. As before, we adopt the notation of Oehlert et al. (2018) to formulate the ILP constraints: The binary ILP decision variable $s_v \in \{0, 1\}$ indicates whether a node $v \in V_{\varphi}$ in the CFG is used as the starting point in the ILP solution. Similarly, the binary ILP variable $e_v \in \{0, 1\}$ signifies whether the considered path ends at v.

Let $v^0 \in V_{\varphi}$ be the initial node in the CFG of φ, whereas $V_{\varphi}^{end} \subset V_{\varphi}$ contains all sinks of the CFG. v^0 corresponds to the initial node of the entry function, e.g. the `int main()` function, and nodes $v^{end} \in V_{\varphi}^{end}$ correspond to `returns` from the entry function. Adding the constraints shown in Eq. (20a) and Eq. (20b) into the ILP formulation from Sect. 5 allows us to compute interference curves for `out-tasks` and `in-tasks`, respectively.

$$s_{v^0} = 1 \tag{20a}$$

$$\sum_{v^{end} \in V_{\tau}^{end}} e_{v^{end}} = 1 \tag{20b}$$

It is not necessary to perform a dedicated curve derivation for the `body-task` curves as they can be derived from the unconstrained curves, which are used to compute `single-task` interference. As a `body-task` φ needs to execute for at least $\mathrm{BCET}(\varphi)$ cycles, η_{φ}^{body} is induced by η_{φ}^{single} using an additional window:

$$\eta_{\varphi}^{body}(\Delta t) = \eta_{\varphi}^{single}(\Delta t) + W_{\mathrm{BCET}(\varphi)}(\Delta t). \tag{21}$$

We apply a windowing function $W_{\mathrm{BCET}(\varphi)}$, which ensures that a scenario containing a `body-task` of φ impacts the final interference computation only for time durations $\Delta t \geq \mathrm{BCET}(\varphi)$, because such a scenario is infeasible for shorter durations.

Using the interference curves of each task in the four categories, we can calculate the total interference created by a core C'. The scenarios in $\mathcal{S}_{C'}$ can be partitioned into two types: scenarios that consist only of a `single-task`, i.e., $s = (\varphi) \in \mathcal{S}_{C'}$, and scenarios that contain an `in-task`, `out-task` and zero or more `body-tasks`, i.e., $s = (\varphi_1, \ldots, \varphi_j) \in \mathcal{S}_{C'}$. We thus differentiate the definition of $F_{C'}$ based on the size of the execution scenario:

$$F_{C'}\left(\varphi_1, \ldots, \varphi_j\right) = \begin{cases} \eta_{\varphi_1}^{\text{single}} & \text{if} j = 1, \\ \eta_{\varphi_1}^{in} \circledast \left(\underset{1<i<j}{\circledast} \eta_{\varphi_i}^{body} \right) \circledast \eta_{\varphi_j}^{out} & \text{otherwise.} \end{cases} \tag{22}$$

When a scenario $s \in \mathcal{S}_{C'}$ contains only a single task φ, i.e., $s = (\varphi)$, its interference curve is equal to the interference curve of that task. For scenarios that contain more than a single task, we compute the worst-case interference as the max-plus convolution of the curves from the `in-task`, `body-tasks`, and the `out-task`. The resulting curve requires at least $\sum_{1<i<j} \mathrm{BCET}(\varphi_i)$ cycles to be feasible. For shorter durations, $F_{C'}(\varphi_1, \dots, \varphi_j)$ yields $-\infty$.

Note, in an actual execution of the system a scenario may occur where no `in-task` or `out-task` exists. This might happen if the interfering core remains idle for some time during the cache block reuse window. However, as we are interested in the worst-case interference, it is safe to disregard such situations in this computation, as they are subsumed in the considered scenarios. I.e., an idle core will not create worse interference conditions than an active core: formally $\forall \varphi \in T_{C'} : 0 \leq \eta_{\varphi}^{in} \wedge 0 \leq \eta_{\varphi}^{out}$, by construction of the interference curves.

Using the function $F_{C'}$, defined in Eq. (22), we can determine the maximal interference caused by a specific scenario $s \in \mathcal{S}_{C'}$. Moreover, by taking the maximal interference over all possible scenarios, we obtain a safe upper bound on the total interference caused by a non-preemptively scheduled core. This upper bound is given by $\mu_{C'}$, as defined in Eq. (19).

5.3.3 Algorithmic computation of the worst-case interference

The upper bound on the interference $\mu_{C'}(\Delta t)$ depends on all possible execution scenarios. As the tasks in the analyzed system may be activated repeatedly, there are infinitely many different execution scenarios. This makes it intractable to compute interference curves for all scenarios in $\mathcal{S}_{C'}$ and determine the maximum interference from the results. However, we are not interested in the interference generated by any specific sequence of tasks but rather the maximal interference for any sequence of tasks. In this section, we present an efficient algorithm that computes the function $\mu_{C'}$ by directly determining the worst-case interference for any Δt. The key idea of the presented algorithm is to intelligently rearrange the required computations, leveraging the properties of the max-plus convolution, particularly its associativity and commutativity (Le Boudec and Thiran 2001).

Algorithm 2 Compute the worst-case interference curve μ for a core C'

```
Input: Task set $T_{C'}$ with interference curves $\eta_\varphi^{single}, \eta_\varphi^{in}, \eta_\varphi^{out}, \eta_\varphi^{body}$ for all $\varphi \in T_{C'}$.
Output: Worst-case interference curve $\mu$ for up to $\mathcal{A}$ events.
1:  $\mu \leftarrow (\Delta t \mapsto 0)$
2:  for all $\varphi \in T_{C'}$ do
3:      $\mu(\Delta t) \leftarrow \max\{\mu(\Delta t),\ \eta_\varphi^{single}(\Delta t)\}$
4:  end for
5:  for all $\varphi_1 \in T_{C'}$ do
6:      for all $\varphi_2 \in T_{C'}$ do
7:          $\mu(\Delta t) \leftarrow \max\{\mu(\Delta t),\ (\eta_{\varphi_1}^{in} \circledast \eta_{\varphi_2}^{out})(\Delta t)\}$
8:      end for
9:  end for
10: repeat
11:     $done \leftarrow$ True
12:     for all $\varphi \in T_{C'}$ do
13:         $\mu'(\Delta t) \leftarrow (\mu \circledast \eta_\varphi^{body})(\Delta t)$
14:         if $\exists \delta \in \mathbb{N}, 1 \le \delta \le \mu^{-1}(\mathcal{A}) : \mu'(\delta) > \mu(\delta)$ then
15:             $\mu(\Delta t) \leftarrow \max\{\mu(\Delta t), \mu'(\Delta t)\}$
16:             $done \leftarrow$ False
17:         end if
18:     end for
19: until $done =$ True
```

The pseudocode to compute the worst-case interference is shown in Algorithm 2. Line 1, initializes the result to the constant zero function. Lines 2–4 compute the maximal interference stemming from `single-task` scenarios by taking the maximum interference over all curves η_φ^{single} for all $\varphi \in T_{C'}$. Then, in the following lines 5–9, scenarios that contain exactly two tasks are considered. The `in-task` and `out-task` curves are combined using a max-plus convolution. This calculation gives the maximal interference caused by any two tasks executing in sequence.

Additionally, this step also prepares the computation of the interference from scenarios containing more than two tasks. This is the case, as the max-plus convolution is commutative and associative, i.e., $\eta_{\varphi_1}^{in} \circledast \eta_{\varphi_2}^{body} \circledast \eta_{\varphi_3}^{out} = (\eta_{\varphi_1}^{in} \circledast \eta_{\varphi_3}^{out}) \circledast \eta_{\varphi_2}^{body}$. Thus, it is possible to perform the convolutions required by Eq. (22) in any order. We use this property to first compute the contribution by the `in` and `out` tasks before iteratively adding additional `body` tasks to the intermediate result.

The outer loop spanning lines 10–19 iterates until there are no more changes observed on the result. This is realized using the *done* flag. If there was a change made to the curve in the current iteration, the flag is set to **False** (line 16). Otherwise, the algorithm terminates.

The inner loop, from lines 12–18, checks whether adding a `body` task of any φ to the intermediate result μ causes the interference to increase. The modified curve

μ' is computed in line 13 by performing a max plus convolution of μ with the `body` curve of the currently considered task φ.

In line 14, we determine the minimal number of cycles required to generate $\mathcal{A}$ events using the current result μ. The value is computed using the pseudo-inverse of μ: $\mu^{-1}(\mathcal{A})$. This bound is determined as after $\mathcal{A}$ accesses to distinct cache blocks, all data will be evicted from an $\mathcal{A}$-way LRU cache. Note that $\mu^{-1}(\mathcal{A})$ is the infimum (see Eq. 14), so that it may be equal to $+\infty$, if the curve μ never creates $\mathcal{A}$ events.

Using this upper time limit $\mu^{-1}(\mathcal{A})$, it is checked whether the curve μ', containing an additional `body` task of φ, creates higher interference than μ for time frames $1 \leq \Delta t \leq \mu^{-1}(\mathcal{A})$. In an implementation, this check can be performed in $\mathcal{A}$ comparisons using the pseudo-inverse of μ and μ'.

If the interference increased, the current result is updated to the maximum of μ and μ' (line 15). The maximum is taken as there may be values Δt for which $\mu(\Delta t) > \mu'(\Delta t)$. In particular, this is the case for $\Delta t < \mathrm{BCET}(\varphi)$. Finally, line 16 updates the *done* flag to **False** to ensure that another iteration of the outer loop is performed. As noted above, this process terminates if adding another `body` task of any $\varphi \in T_{C'}$ does not create a worse interference scenario.

Theorem 2 *The loop in lines* 10 – 19 *in Algorithm* 2 *terminates after at most* $\mathcal{A} - 1$ *iterations*.

Proof The loop terminates if there is no further task $\varphi \in T_{C'}$, such that convolving the intermediate result μ with η_{φ}^{body} creates a higher interference curve for $\Delta t \leq \mu^{-1}(\mathcal{A})$. We will prove the theorem by induction. Let the value μ after i iterations be denoted by μ_{i+2}, and the maximal interference curve by $\mu_{C'}$.

Base case**:** Before reaching the loop, the algorithm computes an intermediate value, which we denote by μ_2.

This curve is maximal for up to two events. In other words, the pseudo-inverse μ_2^{-1} is minimal for $n \leq 2$. I.e., $\mu_2^{-1}(n) = \mu_{C'}^{-1}(n)$ for $n \leq 2$. This is the case as the code in lines 2 – 4 has determined the minimal time required for a single event, whereas the code in lines 5 – 9 has already performed the convolution of two tasks. Convolving additional *body* tasks to the result will not cause 2 events to occur more quickly than $\mu_2^{-1}(2)$ cycles.

Induction step**:** Given a curve μ_N such that $\mu_N^{-1}(n) = \mu_{C'}^{-1}(n)$ for $n \leq N < \mathcal{A}$. Another iteration of the loop in lines 10 – 19 will produce μ_{N+1}, such that:

$$\forall n \leq (N+1) \, : \, \mu_{N+1}^{-1}(n) = \mu_{C'}^{-1}(n). \tag{23}$$

Given Eq. (23) holds, the theorem follows directly, as the algorithm only continues to iterate as long as there is a change in the interference for $\Delta t \leq \mu^{-1}(\mathcal{A})$. As the loop starts with the given value μ_2 and needs an additional iteration to notice no change has happened, the loop will stop after $(\mathcal{A} - 1)$ iterations.

We will now prove by contradiction that Eq. (23) holds. Assume that there exists an M, such that M is the smallest $M > N + 1$ with

$$\mu_M^{-1}(N+1) < \mu_{N+1}^{-1}(N+1) \tag{24}$$

Let $\Delta t = \mu_M^{-1}(N+1)$, i.e., Δt is the minimal duration to observe at least $N+1$ events, according to μ_M. The curve μ_M creates higher interference than μ_{N+1} over Δt cycles. This means that $\mu_M(\Delta t) > \mu_{N+1}(\Delta t)$.

By definition of μ_M, there is a task φ such that we can construct μ_M as the convolution of μ_{M-1} and a body task of φ. In particular, by definition of the convolution, there is a $\delta \leq \Delta t$ such that Eq. (25) holds:

$$\mu_M(\Delta t) = \mu_{M-1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta). \tag{25}$$

There are two different cases based on the value of $0 < \delta \leq \Delta t$. If $\delta = \Delta t$, we have arrived at the contradiction due to the choice of M:

$$\mu_M(\Delta t) = \mu_{M-1}(\Delta t) + \eta_\varphi^{body}(0) \leq \mu_{N+1}(\Delta t). \tag{26}$$

Equation (26) holds because $\mu_{M-1}(\Delta t) \leq \mu_{N+1}(\Delta t)$ by choice of M and $\eta_\varphi^{body}(0) \leq 0$ for any task φ. Equation (24) and Eq. (26) contradict. We thus assume that δ is strictly smaller than Δt.

We note that an iteration only increases the interference. The computed curve never decreases after another iteration, due to the use of the max operator in line 15.

We know that $\mu_{M-1}^{-1}(n) = \mu_{N+1}^{-1}(n)$ for $n \leq N$ because these values are already optimal due to $\mu_N^{-1}(n) = \mu_{C'}^{-1}(n)$ for $n \leq N$ and because $N < (M-1)$. Furthermore, it is true due to the choice of M that $\mu_{M-1}^{-1}(N+1) \geq \mu_{N+1}^{-1}(N+1) > \Delta t$. From this it follows that $\mu_{M-1}(\delta) = \mu_{N+1}(\delta)$ as $\delta < \Delta t$:

$$\begin{aligned} \mu_M(\Delta t) &= \mu_{M-1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &= \mu_{N+1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta). \end{aligned} \tag{27}$$

We know that $\mu_N(\delta) = \mu_{N+1}(\delta)$ because $\delta < \Delta t < \mu_{N+1}^{-1}(N+1)$ and $\mu_N^{-1}(n) = \mu_{N+1}^{-1}(n)$ for $n \leq N$:

$$\begin{aligned} \mu_M(\Delta t) &= \mu_{M-1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &= \mu_{N+1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &= \mu_N(\delta) + \eta_\varphi^{body}(\Delta t - \delta). \end{aligned} \tag{28}$$

By definition of μ_{N+1}, we have arrived at the contradiction:

$$\begin{aligned} \mu_M(\Delta t) &= \mu_{M-1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &= \mu_{N+1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &= \mu_N(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &\leq \mu_{N+1}(\Delta t). \end{aligned} \tag{29}$$

We have consequently arrived at a contradiction by assuming that there exists a curve μ_M, computed by the iterative convolution of `body` tasks, that yields greater interference than μ_{N+1} for time frames $< \mu_{N+1}^{-1}(N+1)$. As the loop only continues iterating as long as there is an increase in the curve for interference levels $\leq \mathcal{A}$, we have shown that the loop will not change the result of μ after $\mathcal{A}-2$ iterations. This is true as the initial value of μ when entering the loop is already maximal for 2 events. A final iteration is needed by the algorithm as written in Algorithm 2 to detect that no change has occured, resulting in the final maximal loopbound of $\mathcal{A}-1$ iterations. □

Note in particular that it also follows from the proof of Theorem 2 that the value μ computed by Algorithm 2 is equal to the worst-case interference curve $\mu_{C'}$ for up to $\mathcal{A}$ events:

$$\forall \Delta t \leq \mu_{C'}^{-1}(\mathcal{A}) \,:\, \mu(\Delta t) = \mu_{C'}(\Delta t). \tag{30}$$

Corollary 1 *The complexity of Algorithm 2 is* $\mathcal{O}(|T_{C'}|^2\mathcal{A}^2 + |T_{C'}|\mathcal{A}^3)$.

Proof The interference curves and the intermediate result can be stored as their pseudo-inverse, which can be represented as an array of $\mathcal{A}$ numbers. Consequently, the loop in lines 2–4 requires $|T_{C'}|\mathcal{A}$ operations. The nested loops in lines 5–9 perform $|T_{C'}|^2$ convolutions, each of which requires $\mathcal{A}^2$ operations. The maximum of the convolution and the intermediate result can be determined in $\mathcal{A}$ operations. Therefore, lines 5–9 are performed in $\mathcal{O}(|T_{C'}|^2\mathcal{A}^2)$ operations. From Theorem 2, we know that the loop spanning lines 10–19 is executed at most $\mathcal{A}-2$ times. In each iteration, a convolution is performed for each task in $T_{C'}$. Updating the intermediate result μ requires $\mathcal{A}$ operations.

For this reason, the algorithm has a time complexity of $\mathcal{O}(|T_{C'}|\mathcal{A} + |T_{C'}|^2\mathcal{A}^2 + |T_{C'}|\mathcal{A}^3)$, which is equivalent to $\mathcal{O}(|T_{C'}|^2\mathcal{A}^2 + |T_{C'}|\mathcal{A}^3)$. □

For a given hardware architecture, the associativity of the shared cache is a constant, relatively small number. Therefore, the complexity of the algorithm collapses to $\mathcal{O}(|T_{C'}|^2)$ for large task sets.

We can safely bound the inter-core interference at a shared cache for non-preemptive scheduling using Algorithm 2. The total interference from multiple interfering cores is bounded by the sum of the interference of each individual core:

$$\gamma_\tau(\Delta t) = \sum_{C' : \tau \in C \wedge C' \neq C} \mu_{C'}(\Delta t). \tag{31}$$

6 Timing-aware cache hit classification

In Sect. 6.1, we construct a cache hit classification criterion for individual cache accesses using the interference curves introduced in the previous section. We use this classification approach to construct a data-flow analysis in Sect. 6.2, which determines whether a cache access will hit or potentially miss in the LLC. Hence, this section corresponds to the fourth step of the presented analysis approach. The output of the DFA are the hit classifications that are used in the final WCET analysis, which is the final step of the timing-aware shared cache analysis.

6.1 Cache hit condition

Both intra-task and inter-core interference contribute to the eviction of cache blocks. We account for these interference sources as follows: We determine the intra-task interference on the path between subsequent accesses to a particular block $b \in \mathcal{B}_\tau$ as the set of accessed cache blocks $b' \neq b$, which are mapped to the same cache set. Inter-core interference is quantified using the interference curves described in the previous section. The key observation is that, in an LRU cache, an access to a cache block moves the block to the youngest position in the cache. Therefore, if a subsequent access to that same block happens quickly enough, inter-core interference will not be able to evict the block from the shared cache. For this reason, we determine the path duration between cache accesses to the same cache block.

To analyze intra-task interference, we abstract a path in the CFG to a sequence of accesses $(a_i)_{i=1}^m$. The access sequence of a path $\pi = (v_1, \dots, v_n)$ is given by concatenating the access sequences associated to each individual node $v \in \pi$.

Definition 4 Let $\pi = (v_1, \dots, v_p)$ be a path with associated access sequence $(a_i)_{i=1}^m$. The intra-task interference on π between two accesses a_1 and a_m to the same cache block $cb(a_1) = cb(a_m)$ is given by

$$int((a_i)_{i=1}^m) = \left\{ cb(a_s) \;\middle|\; \begin{array}{l} 1 \leq s \leq m : \\ cb(a_s) \neq cb(a_m) \wedge \\ \mathrm{CAC}(a_s) \neq N \end{array} \right\}. \tag{32}$$

Without loss of generality, we assume that the analyzed access pair occurs as the first and last accesses in the sequence associated to π, i.e. a_1 is the access to move the target block into the youngest cache position and a_m is the access to be classified. Accesses occurring prior to the initial access to the targeted cache block or after the classified access do not contribute to the intra-task interference.

The set $int((a_i)_{i=1}^m)$ consists of all cache blocks which may be accessed in the sequence $(a_i)_{i=1}^m$ that conflict with the target cache block $cb(a_m)$. Given the associativity $\mathcal{A}$ of the shared cache, we can define the eviction distance along a path.

Definition 5 The eviction distance ξ of a cache block $cb(a_m)$ along a path with access sequence $(a_i)_{i=1}^m$ is the minimal number of additional interfering cache accesses that could lead to a cache miss for a_m:

$$\xi((a_i)_{i=1}^m) = \mathcal{A} - \left|int((a_i)_{i=1}^m)\right|. \tag{33}$$

As mentioned in Sect. 4, we want to analyze paths that may lead to a cache hit on the shared cache. These paths are characterized by a positive eviction distance. To denote such paths, we introduce the notion of a *potential-hit path.*

Definition 6 A path π with associated cache access sequence $(a_i)_{i=1}^m$ is called a potential-hit path for the access a_m iff:

$$\text{CAC}(a_m) \neq N, \tag{34a}$$

$$cb(a_1) = cb(a_m) \wedge \text{CAC}(a_1) = A, \tag{34b}$$

$$0 < \xi((a_i)_{i=1}^m). \tag{34c}$$

Equation (34a) contains the requirements for the access a_m to potentially result in a hit on a particular path: (a) the access a_m may reach the shared cache, (b) the targeted cache block is loaded into the cache by a_1, (c) intra-task interference does not cause the eviction of $cb(a_m)$.

The only missing piece to classify the access a_m as a cache hit is to check whether the inter-core interference is less than the eviction distance on the analyzed path π. As seen in Eq. (11) and Eq. (31), using interference curves, the inter-core interference may be quantified as a function of time. By evaluating γ_τ for the WCET of the considered path, a safe bound on the inter-core interference can be given. We call the WCET of the path π the temporal reuse distance of the cache block $cb(a_m)$.

We will now construct a cache hit classification that depends on the temporal reuse distance of the cache block and the interference function derived from the event-arrival curves discussed in Sect. 5.

Theorem 3 *An access $a \in Acc$ always results in a cache hit if it may only be reached by traversing potential-hit paths π with access sequence $(a_i)_{i=1}^m$, $a_m = a$ satisfying*:

$$\gamma_\tau(\text{WCET}(\pi)) < \xi((a_i)_{i=1}^m). \tag{35}$$

Proof Consider the scenario that the access a could result in a cache miss. This may happen for three different reasons: the targeted cache block was (a) not loaded into the shared cache previously, (b) evicted due to intra-task interference, or (c) evicted due to inter-core interference.

However, all executions containing a must load the targeted cache block into the cache and this block will not be evicted due to intra-task interference, as π is a potential-hit path. Furthermore, $\gamma_\tau(\text{WCET}(\pi))$ is a safe upper bound on the number of aging events due to interfering accesses from competing tasks. As Eq. (35) requires that the eviction distance is strictly greater than the interference, the cache block $cb(a)$ will not be evicted due to inter-core interference. Thus, the access will always result in a cache hit. □

The condition given in Eq. (35) can be written equivalently using the time-to-live function:

$$\mathrm{WCET}(\pi) \leq \mathrm{TTL}_\tau\left(\left|int((a_i)_{i=1}^m)\right|\right). \tag{36}$$

At this point, we are able to quantify cache interference using event-arrival curves and have formulated a sufficient condition to classify an access as a hit. In the next section, we describe how we can efficiently use these two components to derive a hit / potential-miss classification for every access.

6.2 Cache access path analysis

In this section, we utilize Theorem 3 to determine whether specific accesses to the shared cache definitively result in a cache hit. To this end, we perform a data-flow analysis (DFA) on the CFG of the analyzed task $\tau \in T_C$. In the DFA, we examine accesses that would result in a cache hit, provided that no inter-core interference occurs. Checking the condition given in Eq. (35) for a particular access $a \in Acc_\tau$ requires knowledge of the WCET of potential hit-paths leading to the access and their respective eviction distance. Consequently, we may abstract a path from a concrete sequence of nodes to a safe upper bound on its execution time and a bound on the intra-task interference, measured as a set of conflicting blocks potentially accessed on the path.

Path information for access classification is represented in an abstract domain using a semi-lattice $D = (\mathbb{N} \times 2^{\mathcal{B}_\tau}) \cup \{\bot, \top\}$. Elements $(\Delta t, B) \in \mathbb{N} \times 2^{\mathcal{B}_\tau}$ represent a maximal path duration of Δt cycles and intra-task interference $B \subseteq \mathcal{B}_\tau$. The value $\top$ corresponds to a potential cache miss, while $\bot$ indicates a finished load-access path. I.e., another access will load $cb(a)$ into the shared cache en route to the access a, resulting in a cache hit. The tuples are ordered $(\Delta t_1, B_1) \leq (\Delta t_2, B_2) \iff \Delta t_1 \leq \Delta t_2 \wedge B_1 \subseteq B_2$, while $\top$ is the greatest element and $\bot$ is the least element. Joining of two elements is performed by the operator $\sqcup$:

$$(\Delta t_1, B_1) \sqcup (\Delta t_2, B_2) = (\max(\Delta t_1, \Delta t_2), B_1 \cup B_2), \tag{36a}$$

$$d \sqcup \top = \top,\ d \sqcup \bot = d,\ d \in D. \tag{36b}$$

To compute the data-flow information for all nodes in the control-flow graph, we conduct a data-flow analysis in the backward direction. We specify the data-flow information as the mappings $in[v] : Acc_\tau \rightarrow D$ and $out[v] : Acc_\tau \rightarrow D$ for $v \in V_\tau$. Although the analysis is conducted in the backward direction, we use the word *in* (*out*) to denote the data-flow information at the beginning (end) of a node in the regular sense.

Every node $v \in V_\tau$ possesses an associated sequence of cache accesses, which is denoted using the superscript v, $(a_i^v)_{i=1}^m$. The impact of a single cache access a_i^v issued during the execution of v on the analyzed access a is determined by the function f_i^v:

$$f_i^v(a,(\Delta t,B)) = \begin{cases} (a,\top) & \text{if } \gamma_\tau(\Delta t) \geq \mathcal{A} - |B'| \\ (a,\bot) & \text{else if } \begin{matrix} \mathrm{CAC}(a_i^v) = A \wedge \\ cb(a) = cb(a_i^v) \end{matrix} \\ (a,(l,B')) & \text{otherwise} \end{cases} \tag{36c}$$

$$f_i^v(a,\top) = (a,\top),\ f_i^v(a,\bot) = (a,\bot), \tag{36d}$$

where

$$B' := B \cup \left\{ cb(a_i^v) \,\middle|\, cb(a) \neq cb(a_i^v) \wedge \mathrm{CAC}(a_i^v) \neq N \right\}. \tag{37}$$

The set B', defined in Eq. (37), contains all blocks that may be accessed by the analyzed task τ on paths leading to the analyzed access a and conflict with the target block $cb(a)$.

The first case in Eq. (36c) covers the situation in which the interference on the shared cache is too high to guarantee a cache hit. This decision is made based on the bounded path duration Δt and the cumulative interference curve γ_τ. If the interference is at least as high as the resilience $\mathcal{A} - |B'|$, where $|B'|$ is an upper bound on the intra-task interference, the block may be evicted by inter-core interference $\gamma_\tau(\Delta t)$. Thus, the value is updated to $\top$, showing that this access is a potential miss.

In the second case, the path duration is acceptable and the access a_i^v actually causes the target block $cb(a)$ to be loaded into the cache. The value is updated to $\bot$ to show that the load-access path is terminated at this point and a will result in a definite hit from this location. In the final case, the data-flow information is propagated further with the updated set of conflicting cache blocks B'. The functions f_i^v can be composed to operate on sequences of accesses $f_{\alpha,\dots,\beta}^v = f_\alpha^v \circ \dots \circ f_\beta^v$.

Data-flow information for accesses contained in v is initially generated by $G[v]$:

$$G[v] = \left\{ f_{1,\dots,(t-1)}^v(a_t^v,(\mathrm{WCET}(v),\emptyset)) \,\middle|\, 1 \leq t \leq m \,:\, a_t^v \text{ is potential-hit} \right\}. \tag{38}$$

For every access $a_t^v \in (a_i^v)_{i=1}^m$ associated to the node v, potentially resulting in a cache hit, $G[v]$ inserts a tuple representing an abstract path. The initial path duration is approximated by the worst-case execution time of the node v: WCET(v). The initial path information, consisting of the duration WCET(v) and the empty set for intra-task interference, is propagated backwards inside the node v using the function $f_{1,\dots,(t-1)}^v$. Note that only accesses from v occurring prior to a_t^v, i.e., a_i^v, $1 \leq i < t$ can create a conflict for a_t^v at this stage.

Data-flow information arriving at v from successor nodes is contained in $out[v]$. This information is propagated backwards along v by $P[v]$:

$$P[v] = \left\{ f_{1,\dots,m}^v(a,(\Delta t + \mathrm{WCET}(v),B)) \,\middle|\, (a,(\Delta t,B)) \in out[v] \right\}. \tag{39}$$

The propagation function $f_{1,\dots,m}^v$ is applied to the accesses contained in $out[v]$ which are not mapped to $\top$ or $\bot$. We do not need to process elements $(a,\top) \in out[v]$ and $(a,\bot) \in out[v]$, as the analysis has already concluded that the access a will result in either a potential cache miss or definite cache hit, respectively. The path duration Δt is increased to $\Delta t + \mathrm{WCET}(v)$ to reflect the fact that it may increase by up to $\mathrm{WCET}(v)$ cycles by extending the path over v.

Increasing the WCET of the path by $\mathrm{WCET}(v)$ can overapproximate the actual path duration and reduce the analysis precision. This overestimation occurs because the required time of accesses and computations that are not part of the path are included in $\mathrm{WCET}(v)$. Consider the following example for clarification: Assume a basic block issues the access sequence b_1, b_2, b_1, b_3 to the shared cache. We want to analyze the second access to the cache block b_1. The potential-hit path for that access thus consists of the first three accesses b_1, b_2, b_1. Using Eq. (38), the duration of the path is safely estimated to be the WCET of the complete basic block $\mathrm{WCET}(v)$. We can refine this calculation by considering the bus stalling delay in isolation. Instead of including the bus stalling penalties directly in the execution time of a basic block, we may account for potential bus contention separately. To determine the maximal duration of a potential-hit path, we add the maximal stall time for every access to the shared cache that occurs on the path. This approach leads to more precise path durations, as the bus stalling penalty is considered only for accesses that actually lie on the analyzed path. This optimization can be implemented by computing the value $\mathrm{WCET}(v)$ without potential bus stalling delays and adjusting Eq. (36c) to increment the duration l for each access to any cache set. Note that this refinement requires a compositional hardware architecture.

The incoming data-flow information for node v is the combination of $G[v]$ and $P[v]$ (Eq. 40), whereas the outgoing data-flow information consists of the merged information from all successors $w \in V_\tau$ with $(v, w) \in E_\tau$ (Eq. 41).

$$in[v] = G[v] \sqcup P[v] \tag{40}$$

$$out[v] = \bigsqcup_{(v,w)\in E_\tau} in[w] \tag{41}$$

Here, the join operation $\sqcup$ is extended to data-flow mappings by pair-wise joining of elements associated to the same access event.

Each node is initialized with the empty set: $in[v] = out[v] = \emptyset$. The values of $in[v]$ and $out[v]$ are computed iteratively until they converge. Then, an access a can be safely classified as a cache hit if no node v exists with $(a, \top) \in in[v]$.

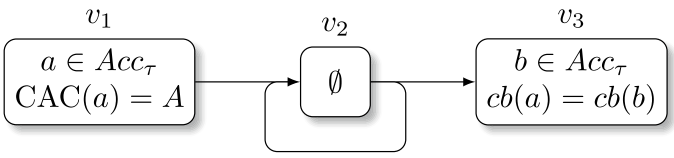

Fig. 4 Example CFG containing three nodes v_1, v_2, v_3. The nodes v_1 and v_3 each contain an access to the same cache block. The node v_2 is part of a loop and may be executed multiple times

6.2.1 Ensuring termination

In its current formulation, the semi-lattice D used in the DFA has infinite height, as the path duration is not limited. Therefore, D does not satisfy the ascending chain condition. In other words, it is not guaranteed that the data-flow information converges after a finite number of iterations. We modify the analysis to guarantee termination in the following way.

We know that all feasible paths contained in the analyzed CFG are of finite duration as we assume that all tasks terminate and thus have a finite WCET. In practice, however, non-termination of the analysis can occur when information is propagated along a path that is infeasible in reality.

Consider the example CFG in Fig. 4. The graph contains three nodes v_1, v_2, v_3. The nodes v_1 and v_3 access the same cache block and the access in v_1 is performed in all circumstances. The access in v_3 may result in a cache hit depending on the duration of the path from v_1 to v_3. However, the CFG contains a loop over node v_2, which is part of the path from v_1 to v_3.

Assume that the interfering cores will never create a sufficient amount of interference to evict the block accessed in v_3. The data-flow analysis will not terminate, as the loop duration is potentially unbounded. Note that this is the case even when an upper loop bound exists, as this information is not available to the data-flow analysis. The DFA is inherently unaware of how many loop iterations have been performed already.

To correct this behavior, it is possible to limit the maximal duration value in the abstract domain. Instead of allowing the path duration to take an arbitrary duration $\Delta t \in \mathbb{N}$, only a limited interval $[1,\ L_\tau] \subset \mathbb{N}$ can be permitted. A natural choice for L_τ is the lowest duration to experience the maximal interference over one full execution of τ, i.e., $\gamma(\mathrm{WCET}(\tau))$. L_τ is thus given by:

$$L_\tau = \gamma_\tau^{-1}(\gamma_\tau(\mathrm{WCET}(\tau))). \tag{42}$$

In case a path duration exceeds this value while being propagated along a node, the duration is instead capped at the upper limit L_τ. This limit on the path duration is safe, as it never underestimates the potential inter-task interference due to the choice of L_τ. Using this upper limit on the path duration, only a finite number of updates may be applied to an abstracted path until the value necessarily stabilizes.

Termination is now ensured, but in practice, the number of iterations required for termination may still be prohibitively large. To solve this problem, we widen an abstracted path $(\Delta t, B)$ to $\top$ after the number of performed propagations over nodes exceeds a certain threshold value. This procedure is safe as we do not introduce faulty cache hit classifications here. However, some precision is sacrificed. We evaluate the impact of different threshold values in Sect. 7.3.

6.2.2 Relative precision

In this section, we discuss the relative precision of the presented analysis compared to the baseline analysis, which was first presented in Hardy et al. (2009).

We abbreviate the timing-aware classification presented in this paper as the TAC method.

It is clear that the quantification of interference over time using event-arrival curves can yield more precise results than assuming that every potentially conflicting block causes interference at all points in time. However, the TAC classification approach presented in this paper is not strictly more precise than the CCN method. This is caused by the inherent loss of precision in the path abstraction.

Consider the control-flow graph in Fig. 4 as an example. Assume that the shared cache is 4-way associative and that interfering tasks may store up to 3 conflicting blocks in the cache. The CFG of the analyzed task contains three nodes v_1, v_2, v_3. The accesses $a \in Acc_\tau$ and $b \in Acc_\tau$ are contained in the nodes v_1 and v_3, respectively. In this scenario, the access a in v_1 causes the block $cb(a)$ to be loaded into the shared cache, while the node v_2 does not contain any accesses to the shared cache. When performing the access b with $cb(b) = cb(a)$, no further blocks were loaded into the cache by the analyzed task. Thus, the block age of $cb(b)$ based on intra-task interference is 0.

An eviction of $cb(b)$ prior to the access b would require interfering tasks to store 4 conflicting blocks in the shared cache. As, in this example, the interfering task only accesses up to 3 conflicting blocks, the CCN approach is able to classify the access as a definite cache hit.

The DFA utilized in the TAC approach has to process the loop which allows the node v_2 to be executed multiple times between v_1 and v_3. The duration of the path from v_1 to v_3 thus depends on the number of iterations performed by the loop v_2. Consequently, the path information associated to the access b originating from v_3 can be propagated many times along the loop v_2. This is the case as the duration Δt in the abstract path information $(b, (\Delta t, \emptyset))$ does not converge until the limit L_τ is reached. As we have discussed in the previous section, the DFA is configured to have a propagation limit for path information, which ensures the quick convergence of the analysis. Depending on the value of the propagation threshold, abstract path information may be widened to $(b, \top)$. The resulting classification for b by the TAC approach could thus be a potential cache miss.

We conclude that the TAC technique is not strictly more precise compared to the classification provided by the standard CCN analysis. However, it is possible to combine the two techniques to create a more precise analysis. To consider a cache access as a hit, it is sufficient that either the CCN analysis or the TAC analysis classifies the access as a cache hit. To combine the two classifications approaches, first, the CCN classification is determined. If the result is not a conclusive cache-hit, TAC analysis can be utilized to decide whether the access is a cache hit or potential miss.

It is important to note that the cache hit classifications of the combined approach are not merely the union of the hit classifications from the timing-aware and baseline techniques when considered in isolation. In fact, a beneficial interaction emerges when both approaches are combined: A cache access that is not classified as a cache hit by either the timing-aware or baseline classifications alone can be classified as a cache hit when both approaches are combined. This situation can occur when the timing-aware classification fails to classify an access as a cache hit because the potential hit-path is too long. The potential hit path duration can be reduced in the combined analysis if a cache access on the potential hit path is classified as a cache hit by the baseline classification but not the timing-aware classification. In the combined analysis, this reduction of the potential-hit path duration can lead to additional hit classifications that are not possible when the approaches are considered in isolation.

We compare the precision of the combined TAC+CCN approach with the pure TAC analysis in the evaluation in Sect. 7.

6.2.3 Iterative application

The DFA of the presented classification approach contains an implicit dependence between the classification of different accesses. To determine a safe upper bound for the duration of a path, the WCET of the contained nodes is accumulated. Initially, these WCET values are derived under the assumption that no access to the shared cache results in a cache hit. However, after classifying some accesses as definitive cache hits, these WCET values may be reduced. As a consequence, the WCET estimate of paths in the CFG also reduces. The DFA can thus be performed a second time using the tighter WCET values. Accesses previously classified as potential misses may now be classified as cache hits due to the shorter duration of the potential-hit paths. Hence, it is possible to apply the DFA iteratively to gain more and more precise cache hit information in each iteration. The iterative DFA analysis is performed as long as there are changes in the access classifications. This process will terminate, because the there is a limited number of cache accesses that can be classified as cache hits and a hit classification made in a previous iteration will never be invalidated, as the potential-hit path duration decreases monotonically.

Note that this iterative process has no impact on the interference curves. The interference curves are derived under the best case assumption, that no interference will occur at the shared cache. Thus, the interference curves remain the same for all iterations. Only the upper bound on the path duration is tightened.

7 Evaluation

In order to evaluate the performance and effectiveness of the timing-aware classification approach, we implemented it in a WCET analyzer (Falk and Lokuciejewski 2010). This tool chain supports the code-generation and WCET analysis for the ARMv4T instruction set. The target architecture consists of multiple cores with a 3-stage in-order pipeline. Each core is equipped with a private L1 cache, which is connected to a shared L2 cache using a round-robin arbitrated bus. Our evaluations cover systems with 2 and 4 cores, where each core executes up to 4 different tasks. The systems use partitioned scheduling, which means that the mapping of tasks to their core is fixed. For all configurations, we generated 20 random task sets. The tasks are taken from the EEMBC AutoBench 1.1 benchmark suite (Poovey et al. 2009) to create a realistic workload. Tasks are released repeatedly, with the minimal inter-arrival time being denoted by the value *Period*(·). We generated task periods

Table 3 Worst-case access timing including the bus access delay using round-robin arbitration

Cores	L2 hit	L2 miss	Hit-miss ratio
2	50	80	0.63
4	130	160	0.81

using the UUnifast approach (Bini and Buttazzo 2005), with a target utilization of 0.7. The target utilization refers to the best-case scenario, without any inter-core interference.

References, which are located in a loop often exhibit different behavior on the first loop iteration compared to subsequent loop iterations. For this reason, we employ virtual inlining and unrolling, limited at a depth of 3 contexts. This means that the analyzer differentiates between the first, second, and all following iterations of a loop, as well as different function call contexts, thereby improving the precision of the WCET estimate.

As discussed in Sect. 6.2.1, the DFA to classify cache accesses possesses a propagation threshold parameter. This means that the path information is not propagated beyond a certain number of nodes, in order to ensure quick termination of the analysis In this evaluation, the propagation limit of the analysis was set to 30 nodes, meaning that hit paths containing up to 30 nodes can be detected by the DFA to result in a cache hit. The impact of this parameter on the precision and runtime is evaluated in Sect. 7.3.

We configured the private L1 caches with a size of 256 bytes, using direct-mapping and a cache block size of 16 bytes. For the shared cache, we evaluated cache sizes ranging from 4 to 64 KB, with 8-way associativity, and cache block size of 64 bytes. Both cache levels use the LRU replacement policy. Data accesses were not cached and the timing for each access was set to take 3 cycles. The instruction access timings were set to 1 cycle for an L1 hit, 10 cycles for an L2 hit and 40 cycles in case of an L2 miss. These timing values do not include potential bus access delays.

An instruction fetch may be stalled for up to $(|\mathcal{C}| - 1) \cdot 40$ cycles at the shared bus due to accesses from other cores. The worst-case access timing consists of the sum of the bus access delay and the L2 access time. Table 3 shows the different timing values including the worst-case bus stall time. As the number of cores increases, the relative difference between a cache hit and cache miss diminishes, because the worst-case access delay is primarily determined by the delay to access the shared bus. Thus, improvements in the cache hit classifications may have a smaller impact on the WCET of a task than expected, since the WCET is also affected by other factors such as the bus access delay.

To evaluate the performance, we utilize two different metrics. The first metric is the percentage of accesses contained in the CFG that could be classified as a cache hit. The second metric is the relative WCET achieved by the analyses compared to the WCET using the baseline CCN classifications as the reference value.

We use the hit ratio in addition to the relative WCET, as the WCET value does not capture improved classifications outside the critical path. Additionally, changes in WCET might be small, even though a significant number of accesses were newly classified as cache hits as the WCET is also influenced by other factors such as the bus access delay.

We compare the performance of four different analysis approaches:

1. Baseline classification (CCN)
2. Worst-case interference placement (WCIP)
3. Timing-aware classification (TAC)
4. Timing-aware classification combined with baseline classification (TAC+CCN)

The baseline analysis counts the number of conflicting cache blocks from tasks running on interfering cores for each cache set. The analysis then assumes that all potential conflicts occur for every access to the shared cache. This approach was used in Hardy et al. (2009) and Liang et al. (2012). An access to a cache block is only classified as a cache hit if the maximal block age is less than the cache associativity minus the number of potential conflicts.

The second approach we evaluate is the Worst-Case Interference Placement analysis, as presented by (Nagar and Srikant 2014, 2016; Nagar 2016). An upper bound on the number of interfering cache accesses is determined and distributed to create the maximal possible cache miss penalty. In the evaluation, we use the approximate WCIP algorithm, as it is reported to have similar performance to the precise solution, with lower analysis overhead. The technique is a quantitative analysis as it only computes a WCET value without classifying individual accesses. Thus, we are only able to compare the resulting WCET values but not the number of issued hit classifications. We selected the WCIP approach over the CEOP approach (Zhang et al. 2022) as the comparison because it promises to scale better for multiple interfering tasks (see Observation 2).

The third approach is the timing-aware classification based on interference curves, as it is presented in this paper. We determine the interference curves as described in Sect. 5 and classify individual cache accesses using the DFA presented in Sect. 6.2.

As we have discussed in Sect. 6.2.2, the precision of the timing-aware classification and baseline classification approaches are incomparable, as both techniques can perform better than the other one under certain circumstances. We also evaluate the performance of the combination of both classification approaches. This means that an access will be considered to be a cache hit if either the baseline classification or the timing-aware classification has deemed the access to be a cache hit.

The evaluation is structured into multiple parts. In Sect. 7.1, we evaluate the cache hit classification performance and WCET reduction, and in Sect. 7.2, we

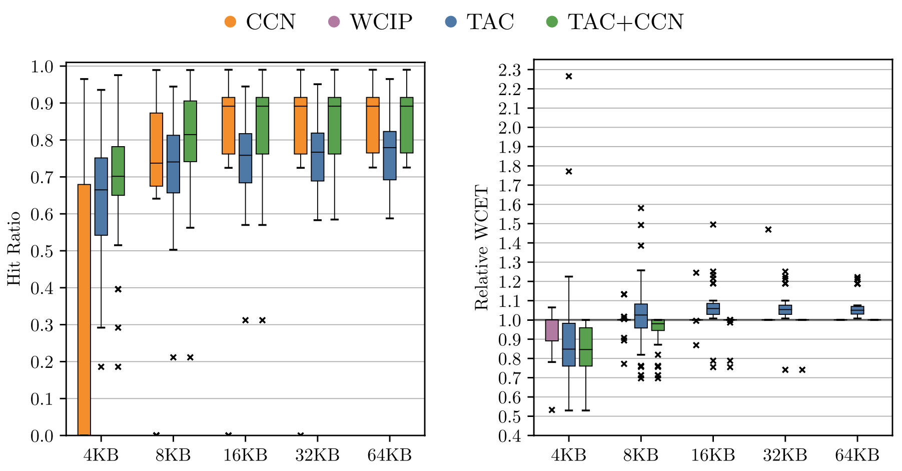

Fig. 5 Hit Ratio and relative WCET for dual-core systems with 1 task per core. WCET results are relative to the baseline CCN approach

evaluate the analysis overhead incurred by the timing-aware classification when determining the WCET estimate. The impact of the propagation threshold value, which was discussed in Sect. 6.2.1, is evaluated in Sect. 7.3.

7.1 WCET reduction

In this subsection, we evaluate the performance of the presented analysis and compare it to other techniques. We use box plots to visualize the results. The line in the middle of each box represents the median value, whereas the lower and upper bounds of a box show the 25th and 75th percentile. The whiskers are at most 1.5 times as large as the central box. Data points outside this range are considered outliers and are marked by a small cross. The metrics are evaluated for each task and grouped for each system configuration.

The results of the baseline CCN approach are colored orange. The WCIP approach is colored purple. The timing-aware classification results, as a standalone analysis and combined with the baseline approach, are colored in blue and green, respectively.

Note that the WCIP approach is a quantitative approach, which does not issue hit classifications. Therefore, the method is absent in the hit ratio evaluation. For the WCET comparison, we normalize the result of the CCN approach at 1.0. A relative WCET smaller than 1.0 corresponds to a reduced WCET, while a larger value means the approach performed worse than the CCN approach.

The results for dual-core systems and quad-core systems are presented in Sect. 7.1.1 and Sect. 7.1.2, respectively.

7.1.1 Results for dual-core systems

We analyzed dual-core systems with 1, 2, and 4 tasks per core. For each task set size, we generated 20 task sets to be analyzed.

The performance results for single task systems are visualized in Fig. 5. The left-hand figure shows the hit ratio, i.e., the percentage of cache accesses in the control-flow graphs, which were successfully classified as a cache hit. The WCIP approach is a quantitative analysis, therefore it does not appear in the comparison of hit ratios. The right-hand figure shows the relative WCET compared to the CCN approach.

For the smallest cache size of 4 KB, the baseline analysis achieves a median hit ratio of 0% (26.6% avg.). Using the TAC approach, as presented in this paper, the median hit ratio increases to 66.5% (63.2% avg.). When the two approaches are combined, the hit ratio further increases to 70.2% (69.2% avg.). Consequently, we see that by leveraging timing information the hit classification precision can be drastically increased.

When increasing the shared cache size to 8 KB, the CCN approach is able to classify significantly more accesses as cache hits. The median hit ratio rises to 73.7%. However, the average is only 64.4% as there are some systems with 0% hit classifications. The TAC achieves a similar median hit ratio of 74.1%, while the average lies at 72.2%.

While both, the CCN and TAC approaches have a comparable median hit ratio, when combining both approaches, the median hit ratio increases to 81.5% (80.3% avg.). This exemplifies the positive effects that occurs when the TAC classification can make use of the CCN classifications to classify additional accesses as cache hits.

For the higher cache sizes from 16 KB to 64 KB, the CCN and TAC+CCN approaches converge to a median hit ratio of 89.1%. The pure TAC approach stabilizes at a hit ratio between 75.0% and 77.6%.

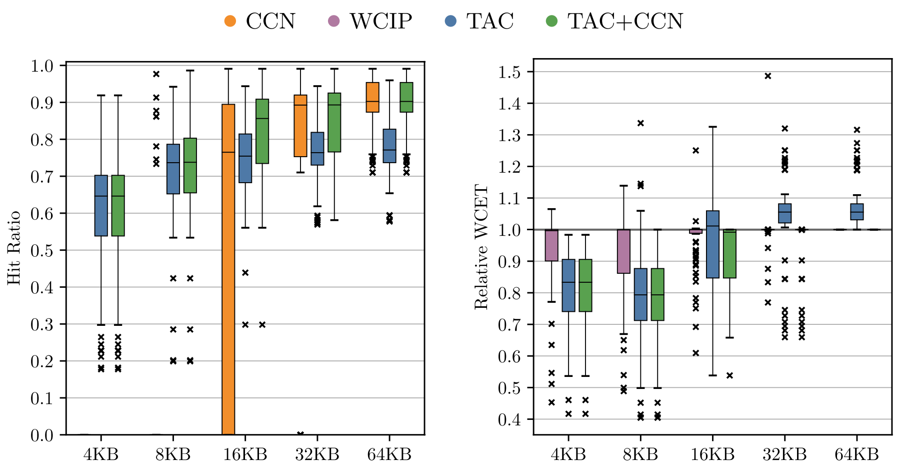

Fig. 6 Hit Ratio and relative WCET for dual-core systems with 2 tasks per core. WCET results are relative to the baseline CCN approach

The impact on the WCET estimate relative to the CCN approach is shown on the right-hand side of Fig. 5.

The biggest changes occur for the smallest 4 KB cache, as this is the configuration, where the CCN approach had a median hit ratio of 0%. In the median, the WCIP approach has no effect on the WCET compared to the CCN approach. On average, we observed 5.1% reduction in the WCET. This is congruent with the reports from Nagar and Srikant (2016), which also reports 0% median WCET reduction. More precisely WCET improvements were reported for 9 out of 27 benchmarks (Nagar and Srikant 2016). In some cases, the WCET computed using WCIP, was larger than the reference value computed using the CCN approach.

The approach presented in this paper performs better than WCIP. We measured a median WCET improvement of 15.2% and 15.4%, for the TAC and TAC+CCN methods, respectively. The average improvement for TAC is 8.9%. The smaller average improvement results from the fact that there are benchmarks in which TAC performs worse than CCN. However, when combining the TAC and CCN approaches, the average reduction stays high at 16.5%. Note, that the combined TAC+CCN approach will never perform worse than the CCN approach.

Using WCIP for the larger 8 KB cache configuration, the average and median WCET reduction is 0.3% and 0.0%. Using only the classifications from the TAC method, the WCET increases by 2.5% in the median. However, for the TAC+CCN approach, the WCET is reduced by a median of 2.0% and 6.3% on average.

For larger cache size the WCIP and TAC+CCN methods perform equally well as the CCN approach, while only using TAC classifications increases the WCET by 5.1% to 5.9% in the median.

We conclude that it is beneficial to use a combined approach, which falls back on the results of the CCN analysis, in case the more complex analyses fail to provide better results. This is the case for both the WCIP, and TAC approach, which can be outperformed by the standard CCN analysis.

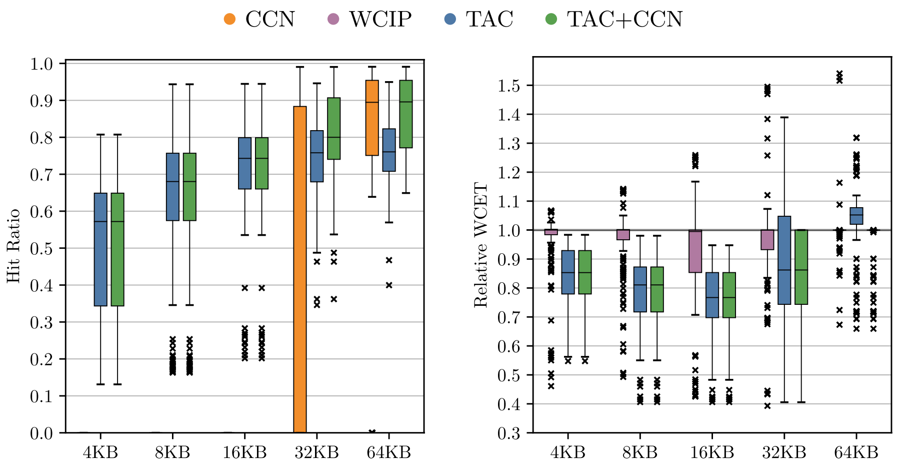

Fig. 7 Hit Ratio and relative WCET for dual-core systems with 4 tasks per core. WCET results are relative to the baseline CCN approach

We also analyzed dual-core systems with two tasks assigned to each core. Again, we evaluated 20 task sets for this configuration. The hit ratio and relative WCET reduction are shown in Fig. 6.

The results show a similar picture compared to the single task systems: For small cache configurations up to 8 KB, the standard analysis is unable to issue any hit classifications, due to the total code size of interfering tasks.

The WCIP approach is able to achieve reductions in the average WCET of 6.2% and 7.4% for the 4 KB and 8 KB L2, respectively. However, the median improvement is 0.3% and 0.1%, respectively.

The TAC and TAC+CCN methods yield significantly higher hit ratios and WCET reductions for the 4 KB and 8 KB cache sizes. The median hit ratios of 64.6% and ca. 73.8% cause a median WCET reduction of 18.4% for 4 KB and 20.5% to 21.6% for 8 KB caches, respectively. For 16 KB caches the TAC+CCN approach reduce the average WCET by 8.7%.

For the 32 KB and 64 KB cache size, the TAC approach, without the CCN classifications as fallback, performs slightly worse than the pure CCN approach, resulting in a median WCET increase of 5.6% and 5.5%. The CCN analysis gives high cache hit ratios, which prevents improvements in the WCET estimate. For these cache sizes, the cache is able to accommodate most of the code at the same time, resulting in few inter-core conflict misses. In the median, the WCET is not changed for when using the combined TAC+CCN approach. For the 32 KB shared cache, the WCET is reduced by 2.9% on average.

We further increased the task set size and analyzed dual-core systems containing four tasks per core. The hit ratio and relative WCET reduction are shown in Fig. 7. A pattern similar to the 1 task-per-core and 2 tasks-per-core systems can be observed for these systems. The presented TAC analysis is able to issue hit classifications even for small caches and thus generates a reduction in the relative WCET. For larger shared caches, the baseline analysis performance rapidly increases once a

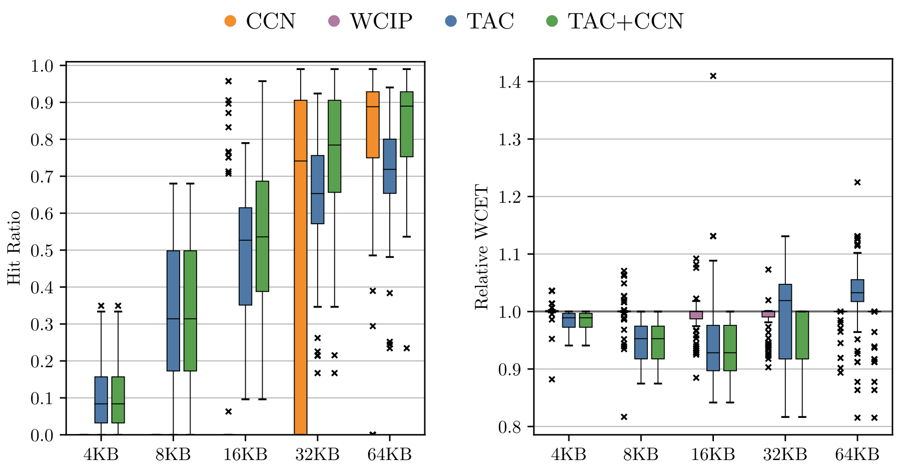

Fig. 8 Hit Ratio and relative WCET for quad-core systems with 1 tasks per core. WCET results are relative to the baseline CCN approach

threshold cache size has been reached. For even larger caches, the inter-core interference becomes less significant as most code is able to fit in the shared cache simultaneously. On average, the code size for each core was 27.7 KB.

The average WCET reductions using the WCIP approach were 3.5%, 3.7%, 7.4%, 3.6%, and −0.2% for the 4 KB to 64 KB cache sizes. In contrast, the qualitative TAC+CCN method achieved significantly higher performance compared to the CCN and WCIP methods. The average WCET reductions were 15.0%, 20.3%, 23.3%, 14.7%, and 3.0%. The performance of TAC was similar to TAC+CCN for all cache sizes below 64 KB, as the CCN approach had a median hit ratio of 0% in these configurations. Consequently, falling back to the CCN classification provided no significant precision increase.

7.1.2 Results for quad-core systems

To analyze the scalability of the presented approach to higher core counts, we also analyzed quad-core systems containing 1, 2, and 4 tasks per core, with shared caches between 4 to 64 KB.

The evaluation results for 20 systems containing a single task per core are shown in Fig. 8. Compared to the dual-core systems with one task per core, as shown in Fig. 5, the number of cache hit classifications are considerably lower. The reason behind this is that by doubling the number of cores from 2 to 4, the number of interfering cores triples from 1 to 3. Thus, a quad-core system has inherently higher inter-core interference, which makes it harder to issue definite hit classifications.

For all cache size configurations, the performance of the WCIP approach was close to the CCN approach. The average WCET reduction was between −0.1% and 1.3%. It can be seen that in the small configurations, where the CCN approach classifies all accesses as cache misses, the WCIP approach may compute a larger WCET value than the CCN method, which classifies all L2 accesses as cache misses. This

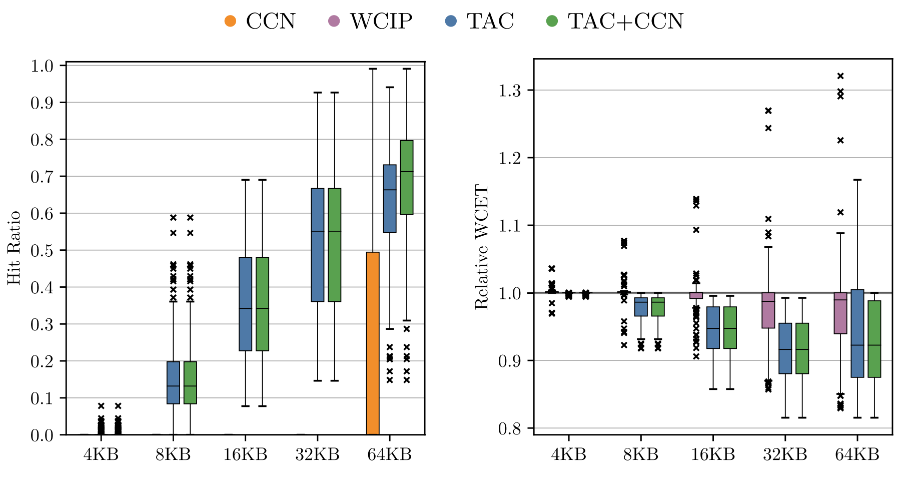

Fig. 9 Hit Ratio and relative WCET for quad-core systems with 2 tasks per core. WCET results are relative to the baseline CCN approach

overestimation is a result of the approximations made in the WCIP approach. The WCIP approach assumes that all potential cache hits lie on the worst-case path in the CFG and computes the penalty accordingly. When the potential interference is so large that all potential hits are classified as cache misses, the resulting penalty can lead to a WCET estimate which exceeds the WCET assuming all L2 cache accesses will result in a cache miss.

For the 4 KB cache the TAC analysis is classifying a smaller fraction of L2 cache accesses as hits compared to the dual-core system. The median hit ratio is 8.4%, leading to a 1.1% median WCET reduction. The performance of the analysis increases for larger cache sizes. The WCET is reduced further for 8 KB and 16 KB caches, where the TAC+CCN analysis has a median hit ratio of 31.4% and 53.3%, respectively. In contrast, the CCN analysis has a median hit ratio of 0% for up to 16 KB. These hit classifications result in a median WCET reduction of 4.7% (5.2% avg.) and 7.2% (6.7% avg.) for 8 KB and 16 KB shared caches.

For the 32 KB shared cache configuration, the CCN analysis is able to issue hit classifications for 74.1% of accesses in the median, compared to 78.5% for the TAC+CCN analysis. This gap in the hit ratio leads to an average WCET reduction of 3.7%. For the largest evaluated cache size of 64 KB, the CCN and TAC+CCN approaches lead to a similar hit classification ratio, as the impact of inter-core interference on the worst-case performance diminishes. However, there are outliers for which better hit classifications where made, leading to an average WCET reduction of 1.2%.

On average, the three interfering cores had a total code size of 19.9 KB. This is reflected in the performance of the CCN approach. For the 16 KB cache configuration, the median hit ratio is 0%. However, as soon as the shared cache size reaches 32 KB, and exceeds the code size of the interfering tasks, the hit ratio jumps to 74.1%. These results clearly demonstrate the pessimism of the CCN approach, in particular for small cache sizes.

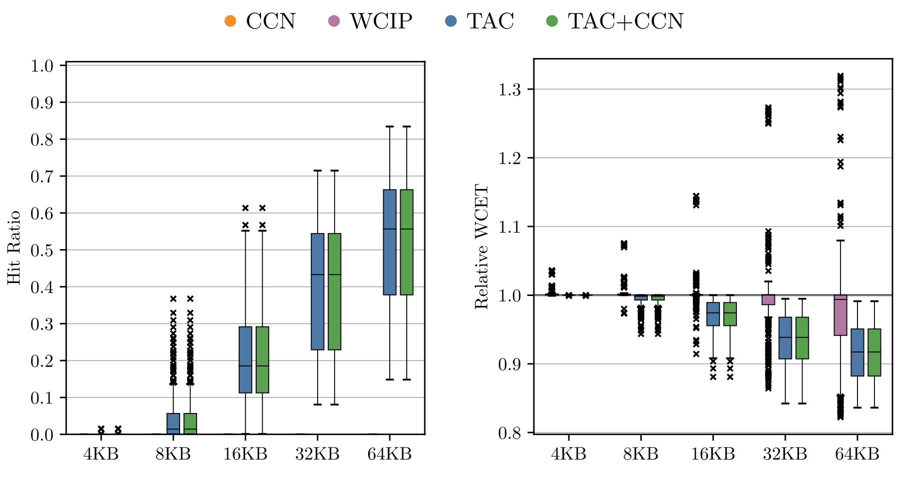

Fig. 10 Hit Ratio and relative WCET for quad-core systems with 4 tasks per core. WCET results are relative to the baseline CCN approach

We also analyzed quad-core systems with two tasks assigned to each core. This means that six different tasks can contribute to the eviction of data from the shared cache. The evaluation results for this configuration are shown in Fig. 9. The large potential of inter-core interference hinders the CCN approach to issue many hit classifications. The median hit ratio is 0% for all cache sizes.

The TAC+CCN analysis achieves a median hit ratio of 0.0%, 13.2%, 34.2%, 55.1%, and 71.3%, for the cache sizes 4 KB – 64 KB respectively. The resulting average WCET reduction is 0.1%, 2.2%, 5.4%, 8.4%, and 7.6%. The TAC and TAC+CCN methods perform identically for 4 KB – 32 KB, as the CCN does not issue hit classifications for these cache sizes. In the 64 KB configuration, relying solely on TAC hit classifications yields a WCET reduction of 5.9% over CCN, compared to 7.6% when both qualitative analyses are combined.

Using the WCIP algorithm, the WCET is reduced on average by −0.2%, −0.3%, 0.2%, 2.2%, and 2.9%. The median relative WCET remains unchanged for 4 KB to 16 KB, whereas a 1.3% and 1.0% decrease are observed for the 32 KB and 64 KB configuration.

The results for quad-core systems with 4 tasks per core are shown in Fig. 10. This means that in total 12 different tasks can contribute to the eviction of data stored in the shared cache. Furthermore, these 12 tasks can be executed repeatedly, which further increases the potential inter-core interference. As the CCN approach does not issue hit classifications, i.e., assumes all shared cache access to result in cache misses, the TAC and TAC+CCN approaches perform identically.

In contrast, the timing aware classification presented in this paper achieves an average hit ratio of 0%, 4.3%, 21.2%, 39.6%, and 51.2%. The WCET median WCET is reduced by 0.0% (0.0% avg.), 0.2% (0.6% avg.), 2.6% (3.0% avg.), 6.1% (6.3% avg.), and 8.3% (8.5% avg.), for 4 KB – 64 KB L2 caches respectively.

The quantitative WCIP method has no impact on the median WCET for cache sizes up to 32 KB. For the 64 KB cache, the relative WCET is decreased by 0.6% in the median and 1.9% on average.

7.2 Runtime evaluation

In this subsection, we take a look at the analysis overhead of the proposed analysis and compare it to the runtime of the CCN and WCIP approach. The evaluations were conducted on an Intel Xeon Server containing 48 cores running at 3.2GHz. All analyses were configured to use only a single processor core.

The runtime of the CCN method consists of the cache block conflict number summation for every cache set and a WCET analysis. It is the approach with the lowest analysis overhead.

To compute the WCET using the WCIP approach, the following steps are performed: (a) calculate WCET without inter-core interference, (b) determine number of conflicting accesses per task execution for each cache set, (c) determine number of cache hits with resilience $k = 1, \ldots, \mathcal{A}$, (d) iteratively solve the interference placement problem until the WCET converges.

In Nagar and Srikant (2016), the number of cache hits and the interference budget is calculated by determining loop bounds and the maximal execution count of each function. By determining these values, it is possible to bound the execution count of each access to the shared cache. The total access count to the shared cache and the number of hits with particular resilience is then given as the sum of all execution counts for the corresponding accesses.

We determine the interference budget by solving an IPET ILP that maximizes the number of L2 cache accesses for a single execution of the task. We apply the same ILP based approach to determine the number of cache hits with different levels of resilience. This means, that for each task and each cache set we solve $\mathcal{A} + 1$ ILPs. The approach of using ILPs, which we employ in this paper, is mentioned as a potentially more precise alternative to the execution count based estimate in Nagar and Srikant (2016).

Using the period of the interfering tasks, we determine how many times each interfering task may be released during a single instance of the analysed task. The approximate WCIP algorithm computes the interference penalty based on these values. Algorithm 1 is performed multiple times until the WCET estimate converges as described in Sect. 3.2.

The runtime of the presented analysis consists of: (a) the BCET/WCET analysis used to derive the event-arrival curves and required to perform the DFA, (b) the derivation of the interference curves, (c) the computation of interference due to non-preemptive scheduling, (d) the DFA to classify accesses, (e) the final WCET analysis.

To derive the interference curves, up to $\mathcal{A}$ ILPs are solved for each task and cache set. Each of these ILPs provides a solution that is used to define the interference curves, as given in Eq. (10). Up to $2\mathcal{A}$ additional ILPs have to be solved to determine the interference curves for the `in-task` and `out-task` categories. It is not always necessary to solve all ILPs, because for example, a task that only accesses three distinct cache blocks in a cache set will never generate four or more events.

Table 4 Average analysis time in minutes

Cores	Cache size	1 Task			2 Tasks			4 Tasks		
		CCN	WCIP	TAC	CCN	WCIP	TAC	CCN	WCIP	TAC
2	4 KB	0.2	0.8	2.1	0.7	2.6	5.7	1.3	5.3	11.7
	8 KB	0.3	1.5	3.1	1.0	5.3	8.2	1.8	12.1	16.9
	16 KB	0.4	3.1	3.2	1.3	10.9	9.1	2.5	23.6	19.4
	32 KB	0.3	5.7	3.4	1.1	20.3	9.9	2.1	44.8	20.6
	64 KB	0.3	10.4	3.2	0.8	34.7	9.7	1.9	80.5	23.7
4	4 KB	0.6	2.3	5.2	1.4	6.0	12.3	2.9	12.7	25.5
	8 KB	0.8	5.0	7.3	2.0	12.5	17.5	4.2	29.7	36.7
	16 KB	1.1	9.8	8.0	2.7	25.9	20.2	5.5	61.1	42.0
	32 KB	1.0	18.8	8.2	2.5	48.4	21.6	4.9	124.5	47.8
	64 KB	0.7	35.8	8.3	1.6	88.2	23.1	3.5	220.0	57.7

Hence, the ILPs determining the time required for more than three events will be infeasible and do not need to be solved.

The interference curves are then combined according to Algorithm 2. The resulting total interference curves are used in the following data-flow analysis to classify the individual cache accesses.

Table 4 shows the average analysis runtimes of the three analysis approaches.

Using the presented analysis, the time required on average to analyse a dual-core system with a single task per core ranges from 2.1 to 3.4 minutes depending on the cache size. The TAC analysis has an overhead between 8.9× and 12.2× compared to the baseline analysis. For single task quad-core systems, the analysis took between 5.2 and 8.3 minutes on average. The overhead over the CCN analysis ranges from 7.3× to 11.4×. Setting the number of tasks per core to 2, the runtime of the TAC analysis increases to 5.7–9.9 minutes for dual-core systems and 12.3–23.1. For a task set size of 4 tasks per core, the timing-aware analysis requires between 11.7 and 23.7 minutes for two cores and from 25.5 to 57.7 minutes for four cores on average. Over all configurations, the overhead of the presented TAC analysis ranges from 7.3× to 16.6×.

From the construction of the analysis, a linear scaling of the analysis time is expected when increasing the system size. This can trend be observed in Table 4. When doubling the number of tasks for dual-core systems, the analysis runtime increases by a factor of 2.8× from 1 to 2 tasks and 2.2× from 2 to 4 tasks. For quad-core systems, the runtime scales by a factor of 2.6× and 2.2× respectively. When keeping the number of tasks constant and doubling the core count from 2 to 4 cores, the runtime scales by a factor of 2.5×, 2.2×, and 2.3×, for 1, 2, and 4 tasks per core respectively.

For the CAC and TAC analysis, the runtime for large caches can decrease compared to the medium-sized caches. For example, for dual-core systems the analysis of 64 KB caches is faster than for 32 KB caches for 1 and 2 tasks per core. With 4 tasks per core, this trend is also visible in the CCN analysis but not the TAC analysis. As this trend occurs also in the CCN analysis, we conclude that it is not inherent to the analyses but rather a property of the underlying WCET analysis tool.

For the presented analysis, the main contributor to the runtime are the ILPs formulated in Sect. 5 to derive the event-arrival curves. As the ILPs are independent of each other, this step can be parallelized to reduce the time requirement. To explore this aspect, we performed the analysis of the largest systems, four cores each executing four tasks, while allowing up to four ILPs to be solved in parallel. The required time for the analysis dropped to 12.6, 18.4, 22.9, 24.7, and 28.4 minutes, for the different cache sizes respectively. These values indicate a reduction of the required runtime by a factor of 2×. Thus, we conclude that the scalability of the presented analysis can be improved by parallelizing the required computations.

The data-flow analysis presented in Sect. 6.2 was insignificant compared to deriving the interference curves, with an average runtime of less than two seconds per system. Combining the interference curves to account for non-preemptive scheduling of tasks, i.e., executing Algorithm 2, did not create significant overhead, taking less than a second to compute.

We observe, that the runtime of our implementation of the WCIP approach is significantly higher compared to the presented TAC analysis. This is a consequence of the way the interference budget and number of cache hits are determined in our implementation of the approach. We solve multiple ILP models to determine these values. As mentioned above, Nagar and Srikant (2016) used an efficient alternative, which operates directly on the control-flow graph of each task. Thus, a more efficient implementation of the WCIP approach is possible. However, as seen in the previous section, the WCIP approach performs similar to the CCN approach, and is significantly outperformed by the presented TAC analysis.

7.3 Propagation limit sensitivity

To classify shared cache access as cache hits, the duration of potential-hit paths is analysed in the presented TAC approach. The maximal duration of these paths is determined using the data-flow analysis developed in Sect. 6.2.

In order to guarantee termination of the data-flow analysis, an upper bound L_τ is imposed on the abstract path duration. This bound corresponds the duration required to experience the maximal inter-core interference. Bounding the path duration ensures that the analysis terminates. However, it may require many iterations for a path to converge on this upper limit.

To ensure quick convergence, we introduced a propagation limit, which widens the information for an access to $\top$, when the propagation limit is exceeded. This means that after propagating path information over a certain number of nodes in the CFG without finding the start of the potential-hit path, i.e. the access that loads the target block into the shared cache, the DFA gives up on this path and classifies the access as a potential miss.

We now explore whether this cut off threshold is necessary and how it affects the analysis precision. The results from the previous sections were determined using a propagation limit of 30 nodes. This means, that potential-hit paths containing up to 30 nodes may be recognized by the DFA as resulting in a cache hit. To explore the sensitivity of the analysis to this parameter, we performed the analysis of single-task systems with different threshold values.

First the propagation limit is decreased to 5 nodes. As only short hit paths may be recognized with this setting, a decrease in the number of successful cache hit classifications is expected. We observed that a lower path propagation threshold had no significant impact on the analysis precision. The average hit ratio results were within 0.2% of the results obtained with the higher threshold of 30 nodes. To check whether a higher propagation limit is useful, we increased the propagation limit to 150 nodes. As for the lower propagation limit, the hit ratio results were within 0.2

percentage points. Thus, a higher propagation limit yields only minor improvements. These results suggest that accesses to the shared instruction cache exhibited highly localized behavior.

The average runtime of the data-flow analysis was 0.17 seconds with the propagation limit set to 30. Decreasing the limit to 5 reduced the runtime to 0.035 seconds, whereas increasing the limit to 150 caused the analysis to require 1 s on average. When removing the propagation limit altogether, 32 of the 200 analyzed systems did not terminate after a cutoff of 4 h per core. We conclude that the propagation limit is necessary, but the analysis precision is not very sensitive to the choice of the parameter value.

8 Conclusion

In this paper, we have presented a novel analysis perspective for shared caches in multicore systems. In the analysis, the inter-core interference between tasks running on different cores is expressed using event-arrival curves. These curves quantify how much time must pass until an interfering task accesses multiple conflicting cache blocks. A core executing multiple tasks using non-preemptive scheduling can be analyzed in this framework by considering different task execution scenarios. The total interference of the scenarios is then computed by convolving the individual task curves in the max-plus algebra. This perspective enables us to view inter-task cache interference as a function of time.

To classify accesses to the shared cache as cache hits, we designed a data-flow analysis, which determines the temporal reuse distance of cache blocks stored in the shared cache. These two components, the event-arrival curves and the temporal reuse distance, allow us to derive safe cache hit classifications for individual accesses to the shared cache.

We evaluated the performance of the analysis using realistic workloads from the EEMBC AutoBench 1.1 benchmark suite. We evaluated systems containing up to 4 cores with 1, 2, and 4 tasks per core. The shared cache size ranged from 4 KB to 64 KB. Utilizing the classifications derived using the presented approach yielded better results compared to the baseline classifications.

The baseline analysis collects all potentially accessed cache blocks and assumes that these blocks may be accessed at any time, which caused the analysis to not produce any hit classifications for many systems. Compared to this baseline analysis, the presented analysis performed particularly well for systems with a small cache size relative to the total program size.

Additionally, we compared the WCET estimates with the quantitative WCIP approach presented by Nagar and Srikant (2014, 2016); Nagar (2016). We observed that the presented approach gave more precise results for all system configurations.

In the future, it would also be interesting to develop an analysis that determines precise temporal reuse values, e.g., using an IPET ILP to determine the longest duration for each potential-hit path. Such an analysis would likely not scale but it could provide valuable insights into the optimal classification of accesses. In particular, it is of interest whether the hit ratio of large systems is limited by the precision of the analysis, or whether it is inherent to the system itself and no better classifications are possible.

We imposed no restrictions on the set of feasible scenarios $\mathcal{S}_{C'}$. This means that the results are general as all sequences of task executions on C' are considered. In the future, the properties of the chosen scheduling approach could be used to disregard infeasible scenarios. The interference curves could be tightened in this way, improving classification performance.

To reduce the required runtime of the analysis, it is conceivable to compute safe approximations of the interference curves instead of precise curves. For example, instead of computing the exact interference curves, as defined in Eq. (10), a subset of the step locations could be determined, while safely approximating the interference curve at the unknown values. This trade off between runtime and precision could be explored in future work.

As presented in this paper, the interference generated from multiple cores is computed in isolation. Considering dependencies or potential interleavings of tasks on multiple cores could increase the analysis precision. This is a topic for future work.

Another direction for future research are memory layout optimizations. Interference curves could be used to guide the placement of data objects and code in the memory layout to reduce interference and increase worst-case performance.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability The evaluation results used to generate the figures are available upon request from the corresponding author.

Declarations

Conflict of interest The authors have no Conflict of interest to declare that are relevant to the content of this article.

References

Aho AV, Lam MS, Sethi R et al (2007) Compilers: principles, techniques and tools, 2nd edn. Pearson Education, London

Altmeyer S, Maiza Burguière C (2011) Cache-related preemption delay via useful cache blocks: survey and redefinition. J Syst Archit 57(7):707–719. https://doi.org/10.1016/j.sysarc.2010.08.006

Fischer TL, Falk H (2023) Analysis of shared cache interference in multi-core systems using event-arrival curves. Proc Real-Time Netw Syst 23:23–33. https://doi.org/10.1145/3575757.3593643

Fischer TL, Falk H (2024) Shared cache analysis under preemptive scheduling. In: Proceedings of Design, Automation & Test in Europe Conference (DATE), pp 1–6. https://doi.org/10.23919/DATE58400.2024.10546581

Bini E, Buttazzo GC (2005) Measuring the performance of schedulability tests. Real-Time Syst 30:129–154. https://doi.org/10.1007/s11241-005-0507-9

Chattopadhyay S, Roychoudhury A (2014) Cache-Related preemption delay analysis for multilevel noninclusive caches. ACM Trans Embed Comput Syst (TECS) 13(5s):1–29. https://doi.org/10.1145/2632156

Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 238–252, https://doi.org/10.1145/512950.512973

Dharishini PPP, Murthy PVR (2021) Precise shared instruction cache analysis to estimate WCET of multi-threaded programs. In: Proc. of INDICON, pp 1–7, https://doi.org/10.1109/INDICON52576.2021.9691620

Falk H, Lokuciejewski P (2010) A compiler framework for the reduction of worst-case execution times. Real-Time Syst 46(2):251–300. https://doi.org/10.1007/s11241-010-9101-x

Ferdinand C, Wilhelm R (1999) Efficient and precise cache behavior prediction for real-time systems. Real-Time Syst 17(2):131–181. https://doi.org/10.1023/A:1008186323068

Gustafsson J, Betts A, Ermedahl A, et al (2010) The mälardalen WCET benchmarks—past, present and future. In: 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), pp 136–146, https://doi.org/10.4230/OASIcs.WCET.2010.136

Hardy D, Puaut I (2008) Wcet analysis of multi-level non-inclusive set-associative instruction caches. In: Proceedings of Real-Time Systems Symposium, pp 456–466, https://doi.org/10.1109/RTSS.2008.10

Hardy D, Piquet T, Puaut I (2009) Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In: Proceedings of 30th IEEE Real-Time Systems Symposium, pp 68–77, https://doi.org/10.1109/RTSS.2009.34

Kelter T (2014) Wcet analysis and optimization for multi-core real-time systems. Technische Universität Dortmund, Dortmund

Le Boudec JY, Thiran P (2001) Network calculus: a theory of deterministic queuing systems for the internet, 1st edn. Springer, Berlin

Lee CG, Hahn H, Seo YM et al (1998) Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. IEEE Trans Comput 47(6):700–713. https://doi.org/10.1109/12.689649

Li YTS, Malik S (1997) Performance analysis of embedded software using implicit path enumeration. IEEE Trans Comput-Aided Design Integr Circuits Syst 16(12):1477–1487. https://doi.org/10.1109/43.664229

Liang Y, Ding H, Mitra T et al (2012) Timing analysis of concurrent programs running on shared cache multi-cores. Real-Time Syst 48(6):638–680. https://doi.org/10.1007/s11241-012-9160-2

Lv M, Guan N, Reineke J et al (2016) A survey on static cache analysis for real-time systems. Leibniz Trans Embed Syst. https://doi.org/10.4230/LITES-v003-i001-a005

Maiza C, Rihani H, Rivas JM et al (2019) A survey of timing verification techniques for multi-core real-time systems. ACM Comput Surv (CSUR) 52(3):1–38. https://doi.org/10.1145/3323212

Nagar K (2016) Precise analysis of private and shared caches for tight WCET estimates. Indian Institute of Science Bangalore, Bengaluru

Nagar K, Srikant YN (2014) Precise Shared Cache Analysis using Optimal Interference Placement. In: IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 125–134, https://doi.org/10.1109/RTAS.2014.6925996

Nagar K, Srikant YN (2016) Fast and precise worst-case interference placement for shared cache analysis. ACM Trans Embed Comput Syst. https://doi.org/10.1145/2854151

Oehlert D (2021) Worst case execution time oriented code optimization of hard real-time multicore systems. Technische Universität Hamburg, Hamburg

Oehlert D, Saidi S, Falk H (2018) Compiler-based extraction of event arrival functions for real-time systems analysis. In: 30th Euromicro Conference on Real-Time Systems (ECRTS 2018), pp 4:1–4:22, https://doi.org/10.4230/LIPIcs.ECRTS.2018.4

Poovey JA, Conte TM, Levy M et al (2009) A benchmark characterization of the EEMBC benchmark suite. IEEE Micro 29(5):18–29. https://doi.org/10.1109/MM.2009.74

Rashid SA, Nelissen G, Tovar E (2022) Tightening the crpd bound for multilevel non-inclusive caches. J Syst Archit. https://doi.org/10.1016/j.sysarc.2021.102340

Reineke J (2018) The Semantic Foundations and a Landscape of Cache-Persistence Analyses. Leibniz Trans Embed Syst. https://doi.org/10.4230/LITES-v005-i001-a003

Thiele L, Chakraborty S, Naedele M (2000) Real-time calculus for scheduling hard real-time systems. In: 2000 IEEE International Symposium on Circuits and Systems (ISCAS), pp 101–104 vol.4, https://doi.org/10.1109/ISCAS.2000.858698

Touzeau V, Maïza C, Monniaux D, et al (2019) Fast and exact analysis for LRU caches. Proceedings of the ACM on Programming Languages (POPL). https://doi.org/10.1145/3290367

Xiao J, Altmeyer S, Pimentel A (2017) Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In: 2017 IEEE Real-Time Systems Symposium (RTSS), pp 199–208, https://doi.org/10.1109/RTSS.2017.00026

Xiao J, Shen Y, Pimentel AD (2022) Cache interference-aware task partitioning for non-preemptive real-time multi-core systems. ACM Trans Embed Comput Syst (TECS) 21(3):1–28. https://doi.org/10.1145/3487581

Yan J, Zhang W (2008) WCET analysis for multi-core processors with shared L2 instruction caches. In: 2008 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 80–89, https://doi.org/10.1109/RTAS.2008.6

Zhang W, Yan J (2009) Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In: 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp 455–463, https://doi.org/10.1109/RTCSA.2009.55

Zhang Z, Koutsoukos X (2016) Cache-related preemption delay analysis for multi-level inclusive caches. In: Proceedings of the 13th International Conference on Embedded Software (EMSOFT), https://doi.org/10.1145/2968478.2968481

Zhang W, Lv M, Chang W, et al (2022) Precise and scalable shared cache contention analysis for WCET estimation. In: Proceedings of the 59th ACM/IEEE Design Automation Conference, pp 1267–1272, https://doi.org/10.1145/3489517.3530613

Thilo L. Fischer is currently a doctoral candidate at the Institute of Embedded Systems, Hamburg University of Technology. He received his master's degree with distinction in Computer Science from the Hamburg University of Technology in 2021. His research interests include compiler-based analyses and optimizations, with a particular focus on multi-level caches for real-time systems.

Heiko Falk received the PhD degree in computer science from the University of Dortmund, Germany, in 2004. From 2004 until 2011, he worked as assistant professor in the embedded systems group at the Technical University of Dortmund. Between 2011 and 2015, he worked as full professor for embedded systems and real-time systems at Ulm University (Germany). Since 2015, he is full professor at Hamburg University of Technology (TUHH) and heads the Institute of Embedded Systems. His PhD focused on high-level source code optimizations. Typical embedded multimedia applications only use a small fraction of their execution time to compute audio or video data. Most of the execution time is used to evaluate complex control flow. Motivated by this observation, he developed novel techniques for control flow optimization at the source code level. Another focus of his work is on code generation and optimization for performance and predictability of safety-critical real-time systems. The WCC compiler initially established by him and developed by the research teams led by him is the currently only known compiler which is able to systematically reduce the worst-case execution time (WCET) of programs by tightly integrating static timing analyses into the code generation and optimization stage. He works on novel concepts to handle parallelism during real-time analysis and optimization. Mutual interferences between tasks running preemptively on the same CPU, or between tasks running on different cores and using shared resources are the key challenges of his current work. Since embedded systems regularly have to meet various additional design constraints besides real-timeliness, his current research also puts an emphasis on multi-criterial optimizations. He and his team develop novel compiler optimizations that are able to systematically trade, e.g., performance versus energy efficiency versus code size. He regularly publishes his research results at prestigious conferences and journals edited, e.g., by Springer, ACM or IEEE. He regularly serves as program committee member and reviewer for the international research community, and he was conference chair and local host of the 2023 edition of the ACM/IEEE Embedded Systems Week (ESWEEK).