Shared Cache Analysis under Preemptive Scheduling

Thilo L. Fischer
Hamburg University of Technology
Hamburg, Germany
thilo.leon.fischer@tuhh.de

Heiko Falk
Hamburg University of Technology
Hamburg, Germany
heiko.falk@tuhh.de

***Abstract*—When sharing a cache between multiple cores, the inter-core interference has to be considered in the worst-case execution time (WCET) analysis. Current interference models are overly pessimistic or not applicable to preemptively scheduled systems. We propose a novel technique to model interference in a preemptive system to classify accesses as cache hits or potential misses. We account for inter-core interference by considering the potential execution scenarios on the interfering core and find the worst-case interference pattern. The resulting access classifications are then used to compute the cache-related preemption delay. Our evaluation shows that the proposed analysis significantly increases the cache hit classifications, reduces WCET on average by up to 11.7%, and reduces worst-case response times on average by up to 15.4% compared to the existing classification technique.**

I. Introduction

In hard real-time systems, the timing properties of tasks are verified to ensure that no deadline will be violated. To this end, the worst-case execution time (WCET) of each task is analyzed. As the memory access latency is a major contributor to the execution time, the behavior of caches needs to be analyzed. The cache behavior can be captured by classifying accesses using cache-hit-miss-classifications (CHMC) [1]. The CHMC distinguishes between accesses that always hit, always miss, or produce uncertain behavior. Based on these access classifications, the WCET can safely be estimated.

In multi-core systems, the last-level cache is often shared between cores, which causes inter-core interference to emerge. The inter-core interference complicates the access classification as interfering cores can evict data from the shared cache. The access classification needs to account for this interference to arrive at a safe WCET estimate. The standard approach to account for inter-core interference counts the number of conflicting cache blocks from tasks running on other cores [2]. All potential conflicts are assumed to occur on each access to the shared cache — this is obviously pessimistic.

Multiple techniques to increase the analysis precision have been proposed [3]–[7]. However, all of these approaches have significant limitations. The exhaustive analysis in [3] computes precise classifications but is computationally infeasible for complex systems. [4] and [5] only compute a timing penalty bound from shared cache interference instead of classifying individual accesses. Additionally, [4] assumes that the total number of conflicting accesses is known; the impact of (non-) preemptive scheduling on the analysis is not discussed. And [5] is limited to non-preemptive scheduling as the access ordering does not account for preemption effects which increase the interference. Finally, [6] and [7] are only applicable to systems with a single task per core.

In this paper, we present a classification method for accesses to a shared cache in a preemptively scheduled multi-core system. Instead of considering all potentially conflicting cache blocks in an access classification, we determine how quickly a task creates interference at the shared cache. Timing properties were first included in the access classification by [6] and [7]; however, without taking the impact of scheduling decisions into account.

The challenge posed by a preemptively scheduled system for access classification is twofold: 1) multiple jobs with different interference profiles can interfere with a single access, and 2) a preempted job may create additional interference when resuming execution.

To solve the first challenge, we construct a regular language, which captures all possible interleavings of jobs on the interfering core. We then compute a bound on the interference of a core by combining the interference caused by the execution of individual jobs. We solve the second challenge by differentiating between previously preempted and unpreempted job executions. For previously preempted jobs, we consider the additional L1 misses due to preemption in the L2 interference computation.

The key contributions of this paper are:

- We develop a model of shared cache interference in preemptive multi-tasking systems based on the inter-arrival time of conflicting cache accesses.
- We incorporate the results of the access classification in the cache-related preemption delay (CRPD) calculation to determine the worst-case response time (WCRT).
- Our evaluation shows significant performance improvements compared to the baseline analysis [2], [8].

In Sec. II, we highlight related work. Sec. III gives an overview of the analysis. A formal language to model scheduling decisions is presented in Sec. IV. Interference curves are computed in Sec. V, while Sec. VI focuses on CRPD. Sec. VII presents the evaluation results. Sec. VIII concludes the paper.

II. Related Work

Hardy et al. [2] analyze shared cache interference by counting the number of conflicting cache blocks. All interfering blocks are considered to cause interference on each access to the shared cache. Liang et al. [9] determine which tasks may run in parallel to reduce the conflict number. Dharishini and Murthy [10] use synchronization points to bound the inter-thread interference in a multi-threaded program. Nagar [4]

979-8-3503-4859-0/DATE24/© 2024 EDAA

presents an analysis for shared caches by determining the worst-case placement of interfering accesses in the control-flow graph. The total delay is bounded using the number of interfering accesses. Zhang et al. [5] use a *happens-before* ordering for conflicting cache accesses. This eliminates infeasible conflicts from the interference computation. The analysis computes an upper bound on the delay caused by cache interference. Xiao et al. [11] analyze the inter-core interference in a non-preemptive system. Inter-core interference is measured as the number of conflicting blocks accessed by interfering tasks. The delay due to interference is computed using the number of accesses targeting potentially evicted blocks.

In contrast to [2] and [9], the approaches [4], [5], [11] are *quantitative* analyses. Instead of classifying each access, only an upper bound on the total delay is given.

Kelter [3] explicitly considers all possible interleavings of accesses to the shared cache. This approach suffers from state space explosion and is not applicable to complex systems. Fischer and Falk [6], [7] use event-arrival curves to analyze how quickly an interfering task may evict data from the shared cache. The reuse time of cache blocks is determined and used to classify accesses as hits or potential misses. However, the analysis is limited to systems with a single task per core.

In contrast to previous research, the approach proposed in this paper is applicable to systems where multiple tasks are scheduled preemptively. Furthermore, the analysis produces a hit or potential miss classification for each access to the shared cache, which is used to determine a bound on the CRPD.

III. System Architecture and Overview

In this section, we describe the system architecture and give an overview of the proposed analysis.

The system architecture consists of multiple cores with private L1 caches. The cores are connected via a round-robin bus to the shared L2 cache. The L2 cache is set-associative with $\mathcal{A}_{L2}$ ways and uses the least-recently-used (LRU) replacement policy. The cache hierarchy is non-inclusive.

We consider a periodic task model; the period of a task τ is denoted by P_τ. Each core is assigned a fixed set of tasks T and uses fixed-priority preemptive scheduling. For tasks $\tau, \varphi \in T$ we write $\tau < \varphi$ iff φ has higher priority than τ. We assume that the scheduling overhead is negligible and does not affect the behavior of the shared cache.

The analysis presented in this paper uses the concept of event-arrival curves to model cache interference [6], [7], [12]. An interference curve expresses how much interference is caused by an interfering core over a specific time frame. Formally, it is a monotonic function $\eta : \mathbb{N}_0 \to \mathbb{N}_0$, which maps a time frame, measured in cycles, to the maximal number of distinct cache blocks that are accessed at the shared cache. This concept allows us to bound the interference based on the time between two accesses to the same cache block.

We will give a brief overview of the analysis described in [7] for systems with a single task per core. It consists of two steps: 1) deriving interference curves and 2) analyzing the cache block reuse time.

The interference curve of a task is derived from its control-flow graph (CFG). To determine the interference over a time frame of Δt cycles, paths in the CFG with a duration less or equal to Δt cycles are considered. The maximal number of accesses to distinct cache blocks is determined using an ILP based on implicit path enumeration. In the second step, potential hits, i.e., accesses to cache blocks that were previously loaded into the cache and have not been evicted by intra-task interference, are analyzed. A backward data-flow analysis determines the maximal duration from the initial access to the potential hit. This value is termed the reuse time. Evaluating the interference curve at the reuse time gives a bound on the interference. Thus, it is possible to classify the access as either a cache hit or potential miss. The approach [7] is restricted to systems with a single task per core as the interference curves are derived from the CFG of individual tasks.

In this paper, we apply the concept of interference curves to shared caches in preemptively scheduled systems. The key idea is to construct a formal language that models the execution of interfering jobs. The language describes all possible interleavings of job executions; each word corresponds to a particular sequence of job executions. Thus, each word in the language has a corresponding interference curve. We can compute the interference curve associated with a word by convolving the individual job curves in the max-plus algebra. We account for preemption effects in this step. Accesses classified as L1 cache hits may access the shared cache after a preemption. This additional interference is included in the interference curve.

The worst-case interference a cache block may experience can be determined by taking the maximum interference over all curves induced by the language. As the language contains infinitely many words, we perform a branch and bound search to determine the worst-case interference curve. We are thus able to make qualitative statements on the behavior of the cache by classifying the cache accesses using the worst-case interference curve. This allows us to safely estimate the WCET accounting for inter-core cache interference.

In a preemptively scheduled system, it is insufficient to only consider the WCET; the delay caused by preemptions has to be considered also. This includes the execution time of the preempting task but also additional context switching costs. In particular, the state of caches is modified during a preemption. Thus, additional cache misses may occur after a preemption.

The available methods [8] [13] for CRPD analysis of non-inclusive two-level caches rely on the standard classification approach [9]. Accesses considered a cache hit by the timing analysis contribute to the CRPD [14]. As the classification approach presented in this paper increases the precision of the hit classification, the additional hit classifications must be accounted for in the CRPD computation. We provide a bound on the additional CRPD in Section VI.

IV. Modelling Interference under Preemptive Scheduling

In this section, we determine how much interference can be created by a core over a particular duration. To compute the worst-case interference curve caused by a set of tasks executing

on a core, we consider all possible execution scenarios, which describe the order in which jobs are processed and how they preempt each other.

There are two types of scheduling events: 1) starting a job, and 2) finishing a job. We call the time frame between two scheduling events a *segment*. In a segment, the core processes the active job. We differentiate the state of the active job into three different categories. They are:

1) **Start:** A job which starts executing in this segment.
2) **Run:** A job which neither starts nor ends in the segment and has been preempted previously.
3) **End**: A job which ends in the segment and may have been preempted.

To represent a job of task τ in the state **Start**, **Run**, or **End**, we utilize the symbols s_τ, r_τ, e_τ respectively. The set Σ contains all possible active jobs (1).

$$\Sigma = \bigcup_{\tau \in \mathrm{T}} \{s_\tau, r_\tau, e_\tau\} \tag{1}$$

The function $C : \Sigma \to (\mathbb{N}_0 \to \mathbb{N}_0)$ maps a segment to its event curve. As noted above, the event curve for a single segment is derived from the CFG of the corresponding task. To compute the value $C(\sigma_\tau)(\Delta t), \sigma_\tau \in \Sigma$, paths of τ that require at most Δt cycles are considered. From these paths, the maximal number of cache blocks stored in the shared cache is determined. When determining the curve for a job that may have been preempted, the additional L2 interference due to the L1 misses caused by the preemption have to be considered. This can be done by considering L1 hits in the unpreempted execution to create L2 interference in the preempted execution.

We will now focus on how the execution of multiple jobs is interleaved due to fixed-priority preemptive scheduling.

Definition 1. *An execution scenario is a sequence of letters from the set* Σ*.*

As an example, consider the following word over Σ for $T = \{1, 2, 3\}$: $s_1 s_2 e_2 r_1 s_3 e_3 e_1$. The interpretation of the word is as follows: First task 1 starts (s_1); it is preempted by task 2 (s_2); after task 2 finishes (e_2), task 1 resumes execution (r_1); task 1 is preempted again, this time by task 3 (s_3); after task 3 ends (e_3), task 1 finishes (e_1).

Furthermore, we define a *valid scenario* as a scenario that is conforms to the notion of fixed-priority scheduling.

Definition 2. *An execution scenario is valid if it satisfies the following conditions:*

1) A job may only be preempted by a job of higher priority.
2) After a job has finished, control is passed to the unfinished job with the highest priority. If there is no unfinished job, any task may spawn a new instance.

Based on these conditions we can model scheduling decisions in a formal language.

Theorem 1. *The set of valid execution scenarios forms a regular language* L*.*

We formulate a regular grammar (V, Σ, P, I) for L to prove Theorem 1 by construction. V denotes the set of non-terminal variables, Σ is the alphabet, P are the production rules, and I is the initial non-terminal. The state of the core is abstracted to the active job and the preempted jobs (2).

$$\mathcal{TS} = \{(\tau_{i_1}, \ldots, \tau_{i_n}) \mid j < k \implies \tau_{i_j} < \tau_{i_k}\} \tag{2}$$

$\mathcal{TS}$ contains all sequences of tasks ordered by ascending priority. The interpretation of a sequence $(\tau, \varphi) \in \mathcal{TS}$ is that a job of τ was running and got preempted by a job of φ. The last task in the sequence indicates the currently active job, in this example a job of task φ. The set of non-terminal variables V contains the initial non-terminal I and is further induced by the set $\mathcal{TS}$ and the category of the last scheduling event (3).

$$V = \{I\} \cup \bigcup_{ts \in \mathcal{TS}} \{S_{ts}, R_{ts}, E_{ts}\} \tag{3}$$

The non-terminals S_{ts} signify that the last scheduling event was the start of a new job. Similarly, the non-terminals R_{ts} show that the system currently processes a **Run** job, while E_{ts} shows that a job was just finished. Using these non-terminals, we formulate regular production rules P in (4).

$$\forall(\ldots, \tau) \in \mathcal{TS}: \; I \to s_\tau S_{(\ldots,\tau)} \mid r_\tau R_{(\ldots,\tau)} \mid e_\tau E_{(\ldots)} \tag{4a}$$
$$\forall \varphi > \tau: \; R_{(\ldots,\tau)} \to s_\varphi S_{(\ldots,\tau,\varphi)} \tag{4b}$$
$$\forall \varphi > \tau: \; S_{(\ldots,\tau)} \to s_\varphi S_{(\ldots,\tau,\varphi)} \tag{4c}$$
$$\forall(\ldots, \tau) \in \mathcal{TS}: \; S_{(\ldots,\tau)} \to e_\tau E_{(\ldots)} \tag{4d}$$
$$\forall(\ldots, \tau) \in \mathcal{TS}: \; E_{(\ldots,\tau)} \to r_\tau R_{(\ldots,\tau)} \mid e_\tau E_{(\ldots)} \tag{4e}$$
$$\forall \tau \in T: \; E_{()} \to s_\tau S_{(\tau)} \tag{4f}$$
$$\forall ts \in \mathcal{TS}: \; S_{ts} \to \varepsilon, \; R_{ts} \to \varepsilon, \; E_{ts} \to \varepsilon \tag{4g}$$

The initial non-terminal I is transformed into an active job τ in one of the three states (4a). Note that the rule does not restrict the initial state of the preempted jobs. All possible scenarios in $(\ldots, \tau) \in \mathcal{TS}$ are considered because we are interested in the scheduling decisions that occur between two usages of a cache block, which happen at an arbitrary point in time. The rule (4b) covers the scenario that a job τ in the category of **Run** is preempted. The start symbol s_φ for the new job φ is appended and the stack of running jobs is extended to $(\ldots, \tau, \varphi)$. By definition, a non-terminal $R_{(\ldots,\tau)}$ can only lead to a preemption and not to the ending of the current job. After starting a new job, there are two possibilities for the next event. Either, the newly started job gets preempted (4c), or it will finish its execution without preemption (4d). In the latter case, φ is removed from the active task stack. The rules (4e) and (4f) are concerned with the scheduling events that happen after a job has finished executing. Rule (4e) covers the situation where a preempted task τ resumes execution after a preemption. It may either run until it is preempted again or finish without being preempted again. (4f) covers the case that the last active job has finished executing. The only action that is possible in this situation is to start a new job. The final rule (4g) allows all non-terminals to be converted to the empty word ε to end the derivation.

Consider a task set $T = \{1, 2, 3\}$, listed in ascending priority. The scenario of a preemption of 1 by 3 immediately followed by starting 2, i.e., $r_1 s_3 e_3 s_2$, is not directly included in L.

However, the behavior is equivalently covered by $r_1 s_3 e_3 r_1 s_2$, as the second r_1 segment may be considered to last for 0 cycles.

Thus, we have created a regular grammar (V, Σ, P, I), which generates the scheduling language L and have proven Theorem 1 by construction. The words in L describe all execution scenarios which may occur on an interfering core. This language is used in the next section to examine the emergent interference behavior at the shared cache. Each scenario $l \in L$ has a different interference pattern on the shared cache. As we want to make hit classifications that hold even under the highest level of interference, we have to consider the worst-case interference generated from all scenarios.

V. Computing Interference Curves

In this section, we will compute the interference curve of a single execution scenario and find the scenario causing the worst-case interference. Let F represent a mapping from a scenario to the induced curve (5).

$$F : L \to (\mathbb{N}_0 \to \mathbb{N}_0^-), \text{ where } \mathbb{N}_0^- = \mathbb{N}_0 \cup \{-\infty\} \tag{5}$$

To properly define the function F, we first need to construct the basic components. Event-arrival curves are computed using the *max-plus* algebra [15]. In the max-plus algebra, $-\infty$ is the absorbing element, i.e., $\forall n \in \mathbb{N}_0 : -\infty + n = -\infty$. We use the value $-\infty$ to signalize infeasible situations. We write the max-plus convolution of η_1 and η_2 as the $\circledast$ operator (6).

$$(\eta_1 \circledast \eta_2)(\Delta t) = \max_{0 \le \delta \le \Delta t} \{\eta_1(\delta) + \eta_2(\Delta t - \delta)\} \tag{6}$$

The function W_t either maps to 0 or $-\infty$ depending on the parameter and applies a *window* to an interference curve (7).

$$W_t(\Delta t) = \begin{cases} 0 & \text{if } \Delta t \ge t \\ -\infty & \text{else} \end{cases} \tag{7}$$

Consider $((\eta_1 \circledast \eta_2) + W_{100})(\Delta t)$ as an example. The window W_{100} is added to the convolution of η_1 and η_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. Thus, applying W_t refines the interference computation for a scenario by enforcing a minimal time between scheduling events.

Each scenario has a minimal duration to be feasible. For shorter durations, it does not contribute to the interference calculation. Let $minDur(\sigma_1 \dots \sigma_n)$ be the minimal duration of a scenario. We give two lower bounds for the minimal duration.

For the first bound, we introduce the helper function $\omega : \Sigma^2 \to \mathbb{N}_0$. It gives a lower bound for the time that needs to pass between scheduling events (8).

$$\omega(a, b) = \begin{cases} BCET(\tau) & \text{if } a = s_\tau, b = e_\tau \\ 0 & \text{else} \end{cases} \tag{8}$$

Eq. (8) states that the minimal time between starting a job and finishing that job is given by the best-case execution time (BCET). Thus, the minimal duration of a scenario is bounded by the time required for all complete job executions (9).

$$minDur(\sigma_1 \dots \sigma_n) \ge \sum_{1 \le i < n} \omega(\sigma_i, \sigma_{i+1}) \tag{9}$$

We create a second bound on the minimal duration by considering the number of jobs that have started for each task (10). Let $\#\tau$ denote the number of starts of τ in the considered scenario $\sigma_1 \dots \sigma_n$. The value $P_\tau \cdot (\#\tau - 2)$ is a lower bound on the time required to activate the task $\#\tau$ times.

$$minDur(\sigma_1 \dots \sigma_n) \ge \max_{\tau \in T} \{P_\tau \cdot (\#\tau - 2) \mid \#\tau > 2\} \tag{10}$$

Using these building blocks, we define F in (11). The function $G_{(\sigma_1 \dots \sigma_n)}$ recursively convolves the individual curves $C(\sigma_m)$ from segments contained in the scenario and applies a window to the result. The window parameter is the larger value from (9) and (10) for $\sigma_1 \dots \sigma_m$.

$$F(\sigma_1 \dots \sigma_n) = G^n_{(\sigma_1 \dots \sigma_n)} \tag{11a}$$

$$G^0_{(\sigma_1 \dots \sigma_n)} = 0 \tag{11b}$$

$$G^m_{(\sigma_1 \dots \sigma_n)} = (G^{m-1}_{(\sigma_1 \dots \sigma_n)} \circledast C(\sigma_m)) + W_{minDur(\sigma_1 \dots \sigma_m)} \tag{11c}$$

The maximal interference, denoted by η^*, is then given as the maximal interference over all scenarios (12).

$$\eta^*(\Delta t) = \max_{l \in L} \{F(l)(\Delta t)\} \tag{12}$$

We are now able to compute an upper limit on the inter-core interference. However, the search space of scenarios in L is infinitely large. To explore it efficiently, we perform a branch and bound search. For this purpose, the scenarios are organized in a prefix tree. Each node represents a scenario $l \in L$; edges correspond to a single derivation step (4); child nodes are scenarios that possess l as a prefix. This is possible as L is a regular language.

We can terminate the exploration at a node with scenario $l \in L$ if its minimal duration exceeds the time required to evict all data from the cache. The scenario will not contribute to the maximal interference curve, thus it is safe to discard it and (by monotonicity of $minDur(\cdot)$) any scenario that possesses l as a prefix.

Note that the language L is an over-approximation of the feasible job sequences as it is constructed independently of the task periods. By considering the period of each task, we can identify infeasible scenarios. A feasible scenario has to satisfy the constraint (13). Let φ^* be the highest priority task.

$$\#\tau \le \left\lceil \frac{(\#\varphi^* + 1) \cdot P_{\varphi^*}}{P_\tau} \right\rceil \tag{13}$$

As φ^* preempts all other tasks and is released on a periodic schedule, $(\#\varphi^* + 1) \cdot P_{\varphi^*}$ is an upper bound on the execution time of the scenario. The maximal number of activations of tasks $\tau \neq \varphi^*$ in a feasible scenario is thus limited by (13).

Using η^*, an individual access to the shared cache can now be classified as a cache hit or potential miss. The worst-case inter-core interference experienced by the cache block is limited by $\eta^*(t_{max})$, where t_{max} is the maximal duration since the last access to the cache block. The access will result in a cache hit if the resilience of the cache block is larger than the inter-core interference. Multiple interfering cores can be modeled by adding the interference value of the individual cores.

VI. Cache-related Preemption Delay

In the previous section, cache accesses are categorized into cache hits and potential misses using the interference curve η^*. The classifications can be used to compute a task's WCET. However, in a preemptively scheduled system, the CRPD needs to be considered to determine the WCRT. In this section, we will discuss how the CRPD can be computed based on the presented hit classification approach.

A preemption impacts the cache behavior in two fundamental ways: there are direct and indirect preemption effects [8]. The direct effect occurs due to cache blocks aging (and being evicted) by the preempting task; the indirect effect occurs due to the increased intra-task interference after a preemption. A cache access resulting in an L1 hit without preemptions can result in an L1 miss due to the direct preemption effects. The access thus gets forwarded to the L2 cache only if a preemption has occurred previously. The additional access to the L2 can cause further evictions and lead to an even higher delay; this is the indirect effect.

Chattopadhyay and Roychoudhury [8] developed a method to compute the CRPD for non-inclusive cache hierarchies using the standard access classification method [2]. It is unsafe to use this CRPD value for the more precise hit classification we have presented in this paper. We will now discuss the necessary extension to compute a safe CRPD value.

The key difference between the standard approach and the one presented in this paper is that timing properties of paths are considered in the classification. This fundamentally challenges a basic assumption made in many cache analyses: Cache sets are assumed to operate independently of each other. This is no longer the case when the classification process is refined to include time. The timing of an access targeting a cache set influences the classification of an access to a different cache set. To safely bound the WCRT, we have to consider this fact in the CRPD computation. In addition to the CRPD value computed by [8], we account for the penalty due to timing effects by assuming that every block reuse path affected by direct or indirect preemption effects will degrade to a cache miss.

From the classifying data-flow analysis (DFA) in [7], we can determine how many hit classifications may be impacted by preemption effects at each program location p from the set of all program locations $\mathbb{P}$ (14). Let $CHMC$ ($CHMC_{STD}$) denote the classification of the presented (standard) approach.

$$LHP(p) = \left\{ acc \;\middle|\; \begin{array}{l} CHMC(acc) = Hit \wedge \\ DFA_{in}[p](acc) \neq \bot \wedge \\ CHMC_{STD}(acc) \neq Hit \end{array} \right\} \tag{14}$$

$LHP(p)$ is the set of all accesses that may degrade to a cache miss due to preemption effects for location p. The condition $CHMC(acc) = Hit$ states that the access is classified as a L2 hit. The condition $DFA_{in}[p](acc) \neq \bot$ means that the targeted cache block will not be refreshed prior to its use by acc [7]. A preemption effect occurring at p may thus cause the access to miss. We do not need to consider accesses classified as a hit by the standard method, as their contribution to the CRPD is already accounted for in [8].

A bound on the number of paths affected per preemption induced cache miss is given in (15). We take the maximum number of live hit paths $LHP(p)$ over program points $p \in \mathbb{P}$.

$$LHP_{max} = \max_{p \in \mathbb{P}} |LHP(p)| \tag{15}$$

We adopt the notation of [8] to group accesses that are susceptible to preemptions. The sets $\mathbb{M}_1(p)$ and $\mathbb{M}_2(p)$ contain accesses that are L1 hits in the absence of preemption but may miss due to a preemption for a program location p. The preemption itself and all accesses contained in $\mathbb{M}_1$ and $\mathbb{M}_2$ disturb data in the shared cache. Thus, the value $\mathbb{M}_{max} = 1 + \max_{p \in \mathbb{P}}\{|\mathbb{M}_1(p)| + |\mathbb{M}_2(p)|\}$ is an upper bound on the number of disturbances.

Consider three accesses a, b and, c where a is contained in the reuse path leading to b and the usage of b is contained in the reuse path for c. Suppose a preemption causes a to miss. The reuse duration of b is extended and may lead to a miss for b, which leads to a miss for c. This pattern can continue for an arbitrary number of accesses. To prevent such a scenario from happening, we account for the worst-case duration (due to a miss from a) in the path analysis for b. Thus, it is not necessary to account for L2 hits that are degraded to L2 misses due to a preemption, as we can account for such misses in the worst-case path duration.

The additional CRPD due to more precise access classifications is given by (16), where $Miss_{L2}$ is the L2 miss penalty.

$$CRPD_{add} = \mathbb{M}_{max} \cdot LHP_{max} \cdot Miss_{L2} \tag{16}$$

The total CRPD value is then given by the value computed by [8] plus the additional penalty $CRPD_{add}$.

VII. Evaluation

We evaluated the performance of the presented approach by implementing it in a WCET analyzer [16]. The systems consist of two ARM7TDMI cores with private L1 caches connected via a round-robin bus to a shared L2 cache. The caches utilize the LRU replacement policy; L1 caches are direct-mapped containing 256 bytes; the shared cache is set-associative with 8 ways. A cache block contains 64 bytes. We evaluated shared cache sizes from 4 KB to 32 KB. The L1-Hit / L2-Hit / L2-Miss timings are 1 / 10 / 40 cycles. We focused our evaluation on instruction caches, as the memory layout of the code is known at compile time. This is not a limitation of the analysis method. Workloads are taken from the EEMBC AutoBench 1.1 suite [17]. We randomly generated 20 systems with 2 tasks per core and 20 systems with 4 tasks per core. Task periods were generated using UUnifast for a target utilization of 0.7 per core. Priorities are assigned according to rate-monotonic scheduling. The analyses were performed on an Intel Xeon server containing 48 cores at 3.2 GHz. Each analysis used only a single processor core. We evaluated three metrics: the percentage of hit classifications, the WCET, and the WCRT.

Fig. 1 and Fig. 2 show the percentage of accesses classified as an L2 cache hit for 2 and 4 tasks, respectively. The presented analysis is shown in blue; the standard analysis is shown in orange. Almost always, no hit classification was made by the

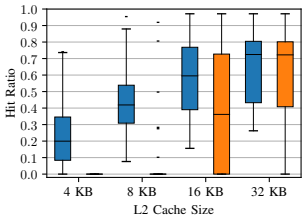

Fig. 1. Hit ratio. 2 tasks per core. The proposed analysis is in blue; the standard analysis in orange

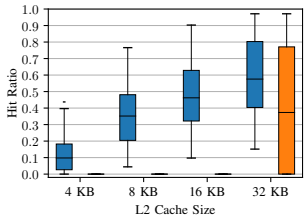

Fig. 2. Hit ratio. 4 tasks per core. The proposed analysis is in blue; the standard analysis in orange

TABLE I
AVERAGE ANALYSIS RESULTS

Tasks	Cache	WCET Red.	WCRT Red.	Time Overhead
2	4 KB	5.1%	6.3%	7.6×
	8 KB	9.6%	12.0%	8.1×
	16 KB	5.3%	6.6%	7.0×
	32 KB	1.5%	1.9%	8.0×
4	4 KB	2.2%	3.4%	7.8×
	8 KB	8.5%	11.3%	9.0×
	16 KB	11.7%	15.4%	7.4×
	32 KB	5.4%	7.5%	7.9×

standard analysis for cache sizes 4 KB – 8 KB for 2 tasks per core. And no hit classification was made for 4 KB – 16 KB for 4 tasks per core. This is due to the code size of the interfering tasks; on average 14 KB (27 KB) from the 2 (4) interfering tasks. As all potential conflicts are considered for every access, the standard analysis is almost always unable to make hit classifications in these configurations. For cache sizes that approach or exceed the code size of the interfering core, the gap between the two classification approaches shrinks.

The presented analysis increases the average hit ratio by 22.9 / 40.6 / 19.6 / 6.4 percentage points (pp) for two tasks per core and cache sizes 4 KB / 8 KB / 16 KB / 32 KB; for four tasks per core, it is increased by 11.1 / 35.6 / 47.5 / 22.1 pp. Table I shows the average analysis results for the different task set and cache sizes. The column WCET reduction shows the decrease of the WCET compared to the standard analysis. The average WCET reduction ranges from 1.5% to 9.6% for two tasks per core, with the largest reduction at the 8 KB cache configuration. For larger caches, the difference between the hit classifications reduces, which results in a smaller WCET improvement. The lowest average reduction occurred at the largest cache configuration. This cache is large enough to fit most of the code from interfering tasks and the analyzed task. For four tasks per core, WCET reduction ranges from 2.2% to 11.7%. The highest reduction was measured for the 16 KB cache. The WCRT reduction follows the same trend as the WCET reduction, while its magnitude is consistently higher. The highest average WCRT reduction is 12.0% for two tasks per core, and 15.4% for four tasks per core.

The average analysis runtime for a complete system varies with the cache size and task set size. It took on average 6.3–10.6 minutes for two tasks per core and 16.1–30.8 minutes for 4 tasks. The overhead compared to the standard approach of simply counting the number of conflicting blocks is shown in the last column of Table I. The overhead ranges between $7.0\times$–$8.1\times$ ($7.4\times$–$9.0\times$) for 2 (4) tasks. Computing the interference curves $C(\cdot)$ scaled linearly in the number of tasks. Although $\mathcal{TS}$ grows exponentially in the number of tasks, we did not observe an increase in the time to compute the interference curves for 4 tasks compared to 2 tasks per core. Both of these components are highly parallelizable, which can be leveraged to reduce the runtime.

VIII. Conclusion

We have presented a hit classification for shared caches in a preemptively scheduled multi-core systems. By modeling the scheduling decisions in a regular language, we can compute an event curve for inter-core interference. Additionally, we have integrated the hit classifications in the CRPD computation to determine the WCRT of a task. Our evaluation showed significant improvements of the cache hit classification percentage, WCET, and WCRT. In the future the scalability could be improved and a tighter bound on the CRPD could be developed.

References

[1] D. Hardy and I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches," in *Proc. of RTSS*, 2008, pp. 456–466.
[2] D. Hardy, T. Piquet, and I. Puaut, "Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches," in *Proc. of RTSS*, 2009, pp. 68–77.
[3] T. Kelter, "WCET Analysis and Optimization for Multi-Core Real-Time Systems," Ph.D. dissertation, Technische Universität Dortmund, 2014.
[4] K. Nagar, "Precise analysis of Private and Shared Caches for tight WCET Estimates," Ph.D. dissertation, Indian Institute of Science Bangalore, 2016.
[5] W. Zhang, M. Lv, W. Chang, and L. Ju, "Precise and Scalable Shared Cache Contention Analysis for WCET Estimation," in *Proc. of DAC*, 2022, pp. 1267–1272.
[6] T. L. Fischer and H. Falk, "WCET Analysis of Shared Caches in Multi-Core Architectures using Event-Arrival Curves," in *Proc. of DATE*, 2023, pp. 1–2.
[7] ——, "Analysis of Shared Cache Interference in Multi-Core Systems using Event-Arrival Curves," in *Proc. of RTNS*, 2023, p. 23–33.
[8] S. Chattopadhyay and A. Roychoudhury, "Cache-Related Preemption Delay Analysis for Multilevel Noninclusive Caches," *ACM TECS*, vol. 13, no. 5s, pp. 1–29, 2014.
[9] Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra, "Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores," *Real-Time Systems*, vol. 48, no. 6, pp. 638–680, 2012.
[10] P. P. P. Dharishini and P. V. R. Murthy, "Precise Shared Instruction Cache Analysis to Estimate WCET of Multi-threaded Programs," in *Proc. of INDICON*, 2021, pp. 1–7.
[11] J. Xiao, Y. Shen, and A. D. Pimentel, "Cache Interference-aware Task Partitioning for Non-preemptive Real-time Multi-core Systems," *ACM TECS*, vol. 21, no. 3, pp. 1–28, 2022.
[12] D. Oehlert, S. Saidi, and H. Falk, "Compiler-Based Extraction of Event Arrival Functions for Real-Time Systems Analysis," in *Proc. of ECRTS*, 2018, pp. 4:1–4:22.
[13] S. A. Rashid, G. Nelissen, and E. Tovar, "Tightening the CRPD bound for multilevel non-inclusive caches," *Journal of Systems Architecture*, vol. 122, 2022.
[14] S. Altmeyer, "Analysis of Preemptively Scheduled Hard Real-time Systems," Ph.D. dissertation, Universität des Saarlandes, 2012.
[15] J. Liebeherr, "Duality of the Max-Plus and Min-Plus Network Calculus," *Foundations and Trends® in Networking*, vol. 11, no. 3-4, pp. 139–282, 2017.
[16] H. Falk and P. Lokuciejewski, "A Compiler Framework for the Reduction of Worst-Case Execution Times," *Real-Time Systems*, vol. 46, no. 2, pp. 251–300, 2010.
[17] J. A. Poovey, T. M. Conte, M. Levy, and S. Gal-On, "A Benchmark Characterization of the EEMBC Benchmark Suite," *IEEE Micro*, vol. 29, no. 5, pp. 18–29, 2009.