Analysis of Shared Cache Interference in Multi-Core Systems using Event-Arrival Curves

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

Caches are used to bridge the gap between main memory and the significantly faster processor cores. In multi-core architectures, the last-level cache is often shared between cores. However, sharing a cache causes inter-core interference to emerge. Concurrently running tasks will experience additional cache misses as the competing tasks issue interfering accesses and trigger the eviction of data contained in the shared cache. Thus, to compute a task's worst-case execution time (WCET), a safe bound on the effects of inter-core cache interference has to be determined. In this paper, we propose a novel analysis approach for shared caches using the least recently used (LRU) replacement policy. The presented analysis leverages timing information to produce tight bounds on the worst-case interference. We describe how inter-core cache interference may be expressed as a function of time using event-arrival curves. Thus, by determining the maximal duration between subsequent accesses to a cache block, it is possible to bound the inter-core interference. This enables us to classify accesses as cache hits or potential misses. We implemented the analysis in a WCET analyzer and evaluated its performance for multi-core systems containing 2, 4, and 8 cores using shared caches from 4 KB to 32 KB. The analysis achieves significant improvements compared to a standard interference analysis with WCET reductions of up to 60%. The average WCET reduction is 9% for dual-core, 15% for quad-core, and 11% for octa-core systems. The analysis runtime overhead ranges from a factor of 4× to 7× compared to the baseline analysis.

CCS CONCEPTS

• **Computer systems organization → Real-time systems**; • **Software and its engineering → Formal software verification**.

KEYWORDS

shared cache, WCET analysis, multi-core, event-arrival curve

ACM Reference Format:
Thilo L. Fischer and Heiko Falk. 2023. Analysis of Shared Cache Interference in Multi-Core Systems using Event-Arrival Curves. In *The 31st International Conference on Real-Time Networks and Systems (RTNS 2023), June 7–8, 2023, Dortmund, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3575757.3593643

1 INTRODUCTION

In multi-core systems, the concurrent execution of multiple tasks influences the state of shared resources, such as shared busses and shared caches. A worst-case execution time (WCET) analysis has to take these effects into account. However, inter-core interference on shared caches is notoriously difficult to quantify. This unpredictability creates the potential for large over-estimation of the worst-case timing behavior. To avoid this over-estimation, a tight bound on the effects of inter-core cache interference is required.

To analyze 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. For private caches, methods [4] [16] based on the well established framework of abstract interpretation [1] exist to determine these access classifications. For shared caches however, the inter-core interference has to be included in the classification process. An intuitive approach to account for inter-core interference when deriving the CHMC of an access is to consider all potentially interfering cache blocks to actually interfere with the analyzed access. The number of interfering blocks is called the *cache block conflict number* (CCN).

This approach was presented by Hardy et al. in [6] and Liang et al. in [8]. 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.

Instead of classifying accesses to the shared cache individually, the authors of [11] and [17] focused on determining an upper bound for the additional execution time caused by inter-core interference. The upper bound for the additional execution time is termed *worst-case-extra-execution-time* (WCEET) in [17]. The actual WCET of a task in a multi-core environment is then given as the sum of the WCET without interference plus the extra execution time due to inter-core interference. 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 not vice versa.

In this paper, we propose a novel analysis approach to derive CHMCs for individual accesses while considering inter-core interference. The approach operates on set-associative shared caches using the least recently used (LRU) replacement policy. To quantify inter-core interference, we view cache accesses issued by each core as an event stream. These event streams can be characterized using event-arrival curves. Thus, we can examine the inter-arrival time between multiple cache access events. This perspective 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 in which interfering tasks can not issue a sufficient number of conflicting accesses to cause its eviction. Consequently, if a block is accessed before its TTL has expired, the access will result in a cache hit.

The key contributions of this paper are:

- We formulate an ILP model to derive event-arrival curves expressing inter-core cache interference.
- We provide a cache hit classification criterion for shared caches.
- We develop a data-flow analysis which leverages timing information to classify shared cache accesses as cache hits or potential misses.
- Our evaluation shows that the analysis is scalable and provides significant improvements upon the standard CCN approach.

In Section 2, related research is discussed. Section 3 gives an overview of the analysis workflow. We introduce the event-arrival perspective on cache interference in Section 4. In Section 5, a cache hit classification criterion is constructed. A data-flow analysis to discern definite hits from potential misses is formulated in Section 6. An evaluation using realistic workloads is presented in Section 7, while Section 8 concludes the paper.

2 RELATED WORK

The static analysis of worst-case behavior for complex systems is an active field of research. A survey of multi-core analysis techniques was published by Maiza et al. in [10], while analyses specifically focused on caches were surveyed by Lv et al. in [9].

Hardy et al. [6] analyzed 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). To classify an access to a cache block as a hit, the number of conflicting blocks has to be less than the associativity minus the block age. In [8], Liang et al. combine this information with 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 an interfering task is assumed to potentially issue all interfering accesses concurrently, at any point in time.

In a recent paper, Zhang et al. [17] aim to eliminate some of this pessimism 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. The approach is evaluated for dual-core systems and compared to the CCN approach. Using the MRTC benchmark suite [5], an average WCET reduction of 13% is reported. However, in the median only a 1% improvement is achieved.

A similar approach was used by the authors of [2] 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 et al. [12] [11] attempt to tackle the cache interference problem from a different perspective. Nagar developed a shared cache analysis by capturing inter-task 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. 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 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 in [17] and [11] 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. In this paper, we tackle the harder problem of classifying each individual access as a cache hit or potential miss.

In [14] [13], Oehlert et al. present 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 IPET ILP model is developed. The event-arrival curves are used to analyze, and subsequently improved, 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 accesses a given number of interfering cache blocks.

3 ANALYSIS OVERVIEW

The system architecture considered in this paper consists of multiple cores with private caches, which are connected to a shared cache via a shared bus. We assume that each core processes a single task $\tau \in \mathrm{T}$. Consequently, inter-core and inter-task interference are synonymous in this context. The shared bus is managed using round-robin arbitration.

We analyze 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. While our analysis is general and applies to both data and instruction caches, we concentrate on instruction caches in this paper. We focus on instruction caches because the memory layout of the program code is known at compile time. This eliminates the need for a value analysis to determine the target address of data accesses.

In the remainder of this section, we provide an overview of the proposed shared cache analysis. The analysis consists of the following steps:

(1) Initial BCET and WCET analysis for each task
(2) Determining cache interference using event-arrival curves
(3) Data-flow analysis to compute cache hit classifications
(4) Final WCET analysis using the new hit classifications

Step 1. In order 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. Whereas the best-case execution time (BCET) is computed using a classical age-based abstract domain [4] without considering the impact of inter-core interference.

Step 2. Following the timing analysis, the event-arrival curves of cache access events originating from each task $\tau \in \mathrm{T}$ are derived. More precisely, it is determined how many distinct cache blocks may be accessed during a particular time frame by solving an ILP model. During this step, the BCET information is utilized to arrive at a safe upper bound. The total inter-task interference experienced by a specific task τ is then given as the sum of interference caused by all tasks running in parallel to τ.

Step 3. Based on this information, a backward data-flow analysis (DFA) is performed. The DFA investigates all accesses which potentially result in a cache hit. An access is considered as a *potential-hit*, if it would result in a cache hit without any inter-core interference. For each *potential-hit* access, 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 loading the block into the shared cache and subsequently accessing it. It also keeps track of any intra-task interference.

By evaluating the event-arrival curves of co-running tasks for the maximal load-access path duration, the inter-task interference can be safely bounded. Finally, potential-hits are classified as either definite cache hits or potential cache misses.

Step 4. The access classifications computed in the previous step can now be used in a WCET analysis to determine the final WCET estimate for each task.

4 EVENT-ARRIVAL CURVES FOR SHARED CACHE INTERFERENCE

In this section the cache access behavior of a task is examined and quantified. To this end, we introduce the concept of event-arrival curves for shared cache interference. The notation used in the following is shown in Table 1.

Under the LRU replacement policy, cache blocks are ordered based on which block was accessed last. Multiple accesses to the same cache block do not cause further aging of other data contained in the cache. Consequently, we define the notion of multiple events as accesses targeting a set of distinct cache blocks.

We denote the number of accesses to pair-wise different cache blocks issued from task $\tau \in \mathrm{T}$ during a time frame of Δt cycles by $\eta_\tau : \mathbb{N} \rightarrow \mathbb{N}$. A mapping $\eta_\tau(\Delta t) = n$ thus means that there exists a scenario in which τ will access n different cache blocks during a time frame of Δt cycles.

Table 1: Variables and Notation

Symbol	Meaning
T	Set of tasks
$\mathcal{A}$	Associativity of the shared cache
Acc	Set of all cache accesses
$\mathcal{B}$	Set of all cache blocks
$cb : Acc \rightarrow \mathcal{B}$	Mapping of accesses to the target block
$q_v \in \mathbb{N}$	Execution count of node v
$z_v \in \mathbb{N}$	Execution time contribution from v
$t_b \in \{0, 1\}$	Indicator whether $b \in \mathcal{B}$ is accessed

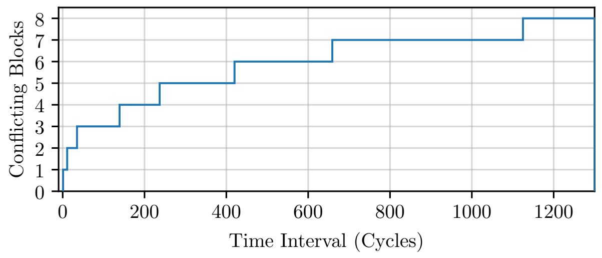


Figure 1: Example for an event-arrival curve expressing shared cache interference. Derived from the `canrdr01` benchmark of the EEMBC AutoBench suite [15].

In Figure 1 an example for such an event-arrival curve is shown. The time interval is shown on the x-axis, while the y-axis represents how many different cache blocks may be accessed. In this example, the interference grows quickly in the beginning. After a time frame of around 140 cycles, the task may have issued accesses to 4 different cache blocks. However, the time interval required to observe a larger amount of interference is significantly higher. To increase the observed interference from 6 to 8 conflicting blocks takes around 700 cycles. This clearly demonstrates the pessimism in the currently available analysis techniques [8] [11] [17]. 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 task to generate the traffic on the shared cache.

We will now show how such an event-arrival curve can be derived from a task's control-flow graph. To derive an event arrival curve from the CFG of a task τ, we have to associate cache accesses to the nodes in the CFG (V, E), where V contains the nodes in the graph and E are the edges connecting the nodes. 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. Thus, we have to consider the pipeline behavior.

We denote the set of all cache blocks by $\mathcal{B}$. Let $\rightsquigarrow \subset V \times \mathcal{B}$ be a relation that contains the pair (v, b) iff executing the node $v \in V$ may cause an access to the block $b \in \mathcal{B}$. A path π is a sequence of nodes $\pi = (v_1, \ldots, v_p)$ with $(v_i, v_{i+1}) \in E,\ i \in \{1, \ldots, p-1\}$. The execution of a path π in the CFG of τ causes other tasks to experience n interfering cache accesses, as given in (1).

$$n = \left| \bigcup_{v \in \pi} \{ b \in \mathcal{B} \mid v \rightsquigarrow b \} \right| \tag{1}$$

Note that η_τ is a step function and we are only interested in the arrival of the first $\mathcal{A}$ events (as after $\mathcal{A}$ events all 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 Figure 1.

To compute the time required for a particular number of accesses, we utilize an IPET model [7] [14]. As the basic construction of such a model is out of the scope of this paper, we focus only on the additional constraints required to model cache interference.

For each cache block, we introduce a binary decision variable to indicate whether this cache block is accessed on the considered path. The variables are denoted by t_b for $b \in \mathcal{B}$. The indicator variables t_b are constrained by (2) and (3), where q_v is the variable containing the execution count of v.

$$0 \leq t_b \leq 1 \tag{2}$$

$$t_b \leq \sum_{v \rightsquigarrow b} q_v \tag{3}$$

Thus, if any node is executed which may access the block b, the indicator variable t_b may take the value 1, otherwise it is set to 0.

To determine the minimal number of cycles during which n cache blocks may be accessed, only paths containing accesses to at least n cache blocks are considered (4). In the model, n is a parameter which may be set to values $1,\ \ldots,\ \mathcal{A}$ to determine the time span required for different levels of interference.

$$n \leq \sum_{b \in \mathcal{B}} t_b \tag{4}$$

Consequently, the objective of the ILP is to minimize the number of cycles required to cause accesses to n distinct cache blocks (5). The path duration is computed as the sum of the execution time contributions z_v of each node $v \in V$.

$$\min \sum_{v \in V} z_v \tag{5}$$

The execution time contribution z_v depends on the best-case execution time of the node v and the execution count q_v. However, 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 is reduced to a single cycle. Solving this ILP for all parameter values $1 \leq n \leq \mathcal{A}$ then allows us to construct the event-arrival curve for each task. As noted before, cache sets are analyzed independently of each other, thus event-arrival curves need to be derived for each cache set.

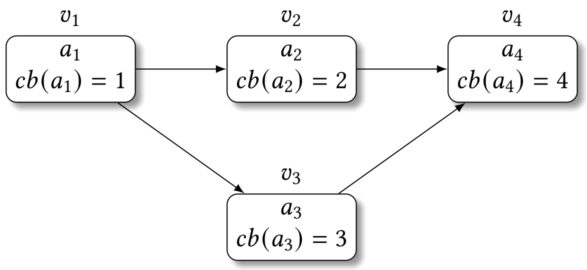


Figure 2: Example CFG containing four accesses targeting different cache blocks. The accesses a_2 and a_3 are mutually exclusive.

Note that, by determining interference on the basis of paths in the CFG, we implicitly eliminate any mutually exclusive accesses from the interference computation. For example, consider the control-flow graph shown in Figure 2. The nodes $v_1, \ldots, v_4$ each contain an access targeting a different cache block. The IPET model will consider the two paths (v_1, v_2, v_4) and (v_1, v_3, v_4). Hence, the maximal interference caused by this CFG is 3 blocks, either the set $\{1, 2, 4\}$ or $\{1, 3, 4\}$ is accessed. The blocks 2 and 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.

The cumulative interference experienced by the task τ can be safely estimated by the addition of the curves η_ϕ from co-running tasks $\phi \in \mathrm{T} \setminus \{\tau\}$. These considerations yield (6) as a safe approximation of the inter-task interference depending on the duration l between subsequent accesses to a particular cache block.

$$\gamma_\tau(l) = \sum_{\phi \neq \tau} \eta_\phi(l) \tag{6}$$

Using the cumulative interference function γ_τ, the time-to-live of a cache block $b \in \mathcal{B}$ can then be expressed as a function of its age. Here, we use the word *age* to refer to the number of conflicting blocks which have been accessed by τ since the last access to b, without considering the inter-task interference. We denote this function as $\mathrm{TTL}_\tau : \{0, \ldots, \mathcal{A} - 1\} \rightarrow \mathbb{N}_0 \cup \{\infty\}$.

$$\mathrm{TTL}_\tau(age) = \sup_{0 \leq l} \{ l \mid \gamma_\tau(l) < \mathcal{A} - age \} \tag{7}$$

Note that the TTL of a block may be $+\infty$ cycles in case the interfering tasks never issue enough interfering accesses to trigger the eviction. We utilize the notion of cumulative interference in the next section to derive cache hit classifications.

5 CACHE HIT CLASSIFICATION

In this section, we construct a cache hit classification criterion based on the duration between accesses to the same cache block. We call the set of cache accesses contained in the programs *Acc*. The *cache-access-classification* (CAC) is denoted by the mapping $\mathrm{CAC} : Acc \rightarrow \{A, N, U\}$. The CAC signifies whether the access will reach the shared cache always (A), never (N), or whether the behavior is uncertain (U).

We will now examine the intra-task interference between two accesses to the same cache block. For this purpose, we can abstract a path in the CFG to a sequence of accesses $(a_i)_{i=1}^m$.

Definition 1. *The intra-task interference on a path between two accesses a_1 and a_m to the same cache block $cb(a_1) = cb(a_m)$ is given by:*

$$int((a_i)_{i=1}^m) = \left\{ cb(a_s) \;\middle|\; \begin{array}{l} 1 \leq s \leq m : \\ cb(a_s) \neq cb(a_m) \wedge \\ CAC(a_s) \neq N \end{array} \right\} \tag{8}$$

The set 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 2. *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 which could lead to a cache miss for a_m as given in (9).*

$$\xi((a_i)_{i=1}^m) = \mathcal{A} - \left| int((a_i)_{i=1}^m) \right| \tag{9}$$

As mentioned in Section 3, we want to analyze paths which 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 3. *A path with associated cache access sequence $(a_i)_{i=1}^m$ is called a potential-hit path for the access a_m iff:*

$$CAC(a_m) \neq N \tag{10a}$$

$$cb(a_1) = cb(a_m) \wedge CAC(a_1) = A \tag{10b}$$

$$0 < \xi((a_i)_{i=1}^m) \tag{10c}$$

Equation (10) 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 will 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-task interference is less than the eviction distance. As seen in (6), using event-arrival curves, the inter-task interference may be quantified as a function of time. By evaluating γ_τ for the WCET of the considered path, a safe estimate of the inter-task interference can be made. We can thus construct a cache hit classification which depends on the temporal reuse distance of the cache-block and the interference function derived from the event-arrival curves discussed in Section 4.

Theorem 1. *An access $a \in Acc$ will always result 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:*

$$\gamma_\tau(WCET(\pi)) < \xi((a_i)_{i=1}^m) \tag{11}$$

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-task 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, $\gamma_\tau(\mathrm{WCET}(\pi))$ is a safe upper bound on the number of aging events due to interfering accesses from competing tasks. As (11) requires that the eviction distance is strictly greater than the interference, the cache block $cb(a)$ will not be evicted due to inter-task interference. Thus, the access will always result in a cache hit. □

Note that the condition given in (11) can be written equivalently using the time-to-live function:

$$\mathrm{WCET}(\pi) \leq \mathrm{TTL}_\tau(|int((a_i)_{i=1}^m)|) \tag{12}$$

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. What is missing now is an algorithmic description on how we can efficiently use these two components to derive a classification for every access.

6 PATH ANALYSIS

In this section, we utilize Theorem 1 to determine whether accesses to the shared cache definitively result in a cache hit. To this end, we perform a data-flow analysis. In the DFA, we examine accesses which would result in a hit provided no inter-core interference is present. Checking the condition given in (11) for a particular access $a \in Acc$ 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 set of potentially accessed conflicting blocks.

Thus, path information for access classification can be represented in an abstract domain using a semi-lattice $D = (\mathbb{N} \times 2^{\mathcal{B}}) \cup \{\bot, \top\}$. Elements $(l, C) \in \mathbb{N} \times 2^{\mathcal{B}}$ represent a maximal path duration of l cycles and intra-task interference $C \subseteq \mathcal{B}$. $\top$ corresponds to a potential cache miss, while $\bot$ is used to signal 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 intuitively ordered $(l_1, C_1) \leq (l_2, C_2) \iff l_1 \leq l_2 \wedge C_1 \subseteq C_2$, while $\top$ is the greatest element and $\bot$ is the least element. Joining of two elements is performed by the $\sqcup$ operator as defined in (13).

$$(l_1, C_1) \sqcup (l_2, C_2) = (\max(l_1, l_2), C_1 \cup C_2) \tag{13a}$$

$$d \sqcup \top = \top,\ d \sqcup \bot = d,\ d \in D \tag{13b}$$

To compute the data-flow information for all nodes in the control-flow graph, we conduct a data-flow analysis in the backward direction. To formalize this DFA, we specify the data-flow information as the mappings $in[v] : Acc \to D$ and $out[v] : Acc \to D$ for $v \in V$. 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 $v \in V$ 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 (14).

$$f_i^v(a,(l,C)) = \begin{cases} (a,\top) & \text{if } \gamma_\tau(l) \geq \mathcal{A} - |C'| \\ (a,\bot) & \text{else if } \ \mathrm{CAC}(a_i^v) = A \wedge \\ & \qquad cb(a) = cb(a_i^v) \\ (a,(l,C')) & \text{else} \end{cases} \tag{14a}$$

$$f_i^v(a,\top) = (a,\top),\ f_i^v(a,\bot) = (a,\bot) \tag{14b}$$

where

$$C' := C \ \cup \ \left\{ cb(a_i^v) \ \middle| \ cb(a) \neq cb(a_i^v) \wedge \mathrm{CAC}(a_i^v) \neq N \right\} \tag{15}$$

The first case in (14a) covers the situation in which the interference on the shared cache is too high to guarantee a cache hit. 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. In the final case, the data-flow information is propagated further with the updated set of conflicting cache blocks C' given in (15). 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 $G[v]$ as defined in (16).

$$G[v] = \left\{ f_{1,\dots,(t-1)}^v(a_t^v, (\mathrm{WCET}(v), \emptyset)) \mid 1 \leq t \leq m \right\} \tag{16}$$

The initial path duration is approximated by WCET(v). Starting from the empty set, the blocks conflicting with a_t^v are derived from the event sequence $(a_i^v)_{i=1}^{t-1}$ by using the propagation function $f_{1,\dots,(t-1)}^v$.

Data-flow information arriving at v from successor nodes is contained in $out[v]$. This information is propagated backwards along v by $P[v]$ as shown in (17).

$$P[v] = \left\{ f_{1,\dots,m}^v(a,(l + \mathrm{WCET}(v), C)) \mid (a,(l,C)) \in out[v] \right\} \tag{17}$$

The propagation function $f_{1,\dots,m}^v$ is applied to the accesses contained in $out[v]$ which are not mapped to $\top$ or $\bot$. The path duration l is increased to $l + \mathrm{WCET}(v)$ to reflect the fact that it may increase by up to WCET(v) cycles by extending the path over v.

The incoming data-flow information for node v is the combination of $G[v]$ and $P[v]$ (18), whereas the outgoing data-flow information consists of the merged information from all successors w (19).

$$in[v] = G[v] \sqcup P[v] \tag{18}$$

$$out[v] = \bigsqcup_{(v,w)\in E} in[w] \tag{19}$$

Here, the join operation $\sqcup$ is extended to data-flow mappings by pair-wise joining of elements associated to the same access event. The values of $in[v]$ and $out[v]$ are computed iteratively until they stabilize. Then, an access a can be safely classified as a cache hit if no node v exists with $(a,\top) \in in[v]$.

6.1 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. It is not guaranteed that the data-flow information converges after a finite number of iterations.

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.

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 $l \in \mathbb{N}$, only a limited interval $[1,\ L_\tau] \subset \mathbb{N}$ can be permitted. A natural choice for L_τ is the smallest duration to experience the maximal interference as given in (20).

$$L_\tau = \min_{l\in\mathbb{N}} \{ l \mid \gamma_\tau(l) = \gamma_\tau(\mathrm{WCET}(\tau)) \} \tag{20}$$

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.

While termination is now ensured, in practice, the number of iterations required for termination may still be prohibitively large. To solve this problem, we widen an abstracted path (l,C) 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 Section 7.3.

6.2 Relative Precision

In this section, we make a few notes about the relative precision of the presented analysis compared to the baseline analysis [8]. We use the abbreviation CCN to refer to the baseline method which uses the cache block conflict number to derive cache hit classifications. For brevity, we denote the method presented in this paper as the EAC method.

It is clear that the quantification of interference over time using event-arrival curves can yield more precise results than assuming that every conflicting block causes interference at all points in time. However, the EAC classification approach presented in this paper is not strictly more precise than the CCN method.

Consider the control-flow graph in Figure 3 as an example. The graph contains three nodes x, y, z. They contain accesses a and b in the nodes x and z respectively.

In this scenario, the access a causes the block $cb(a)$ to be loaded into the shared cache. The node y does not contain any accesses to the shared cache. Thus, when performing the access b to $cb(b) = cb(a)$, no further blocks were loaded into the cache by the analyzed task. Thus, the block age from the isolated perspective is 0. The CCN analysis can now determine whether a hit classification is appropriate using the maximal number of interfering cache blocks.

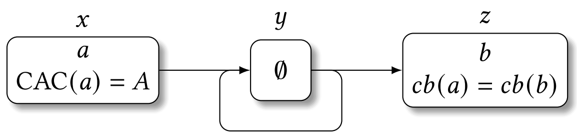


Figure 3: Example CFG containing two accesses targeting the same cache block.

The DFA utilized in the EAC approach, however, has to process the loop which allows the node y to be executed multiple times between x and z. The path information originating from b can potentially be propagated many times along the loop y. Depending on the propagation threshold, the DFA may stop and widen the result of b to a potential miss. Thus, the EAC technique is not strictly more precise compared to CCN.

However, it is possible to combine the two techniques to create a more precise method. To consider a cache access as a hit, it is sufficient that either the CCN analysis or the EAC analysis produces a hit classification. To combine the two classifications approaches, first, the EAC classification is determined. If the result is not conclusive the information of the CCN analysis can be utilized. We refer to the combination of the EAC and CCN classifications as EAC+. We evaluate the relative precision of the EAC and CCN analyses in practice in Section 7.4.

6.3 Iterative Application

The DFA described in this section 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 will result in a cache hit. However, after classifying some accesses as definitive cache hits, these WCET values may be reduced. The DFA can then be performed a second time using the tighter WCET values. Accesses which were 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.

7 EVALUATION

To evaluate the performance of the novel classification approach, we implemented it as a module in the WCC compiler [3]. The target architecture consisted of multiple ARM7TDMI cores with private L1 caches connected to a shared L2 cache using a round-robin arbitrated bus. We considered systems containing 2, 4, and 8 cores. For all core counts, we analyzed 10 systems. For each system, we randomly assigned a task to each core. The tasks were taken from the EEMBC AutoBench 1.1 benchmark suite [15] to create a realistic workload. Virtual inlining and unrolling was limited at a depth of 3, respectively.

For the private L1 caches, we set the cache size to 256 bytes, direct-mapping and a cache block size of 16 bytes. Shared cache sizes of 4 KB to 32 KB were evaluated, with 8-way associativity, and cache block size of 64 bytes. The caches used the LRU replacement policy. 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, excluding potential bus access delays. As we focus on instruction caches in this evaluation, we assume that each data access takes 3 cycles.

Table 2: 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
8	290	320	0.91

An instruction access may be stalled for up to $(|T| - 1) \cdot 40$ cycles at the shared bus due to accesses from other cores. The worst-case access timing thus consists of the sum of the bus access delay and the L2 access time. Table 2 shows the different timing values including the worst-case bus stall time. It can be seen, that the difference between a cache hit and cache miss shrinks for higher core counts, as the worst-case access delay is predominantly determined by the delay to access the shared bus. Thus, improvements in the cache hit classifications may have a smaller than expected impact on the WCET of a task, as the WCET is also impacted by other factors such as the bus access delay.

7.1 Evaluation Results

In this subsection, we evaluate the performance of the presented EAC+ analysis. As the reference point, we also analyze the systems using the method described in [8]. This baseline is abbreviated as the CCN analysis.

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 used in this evaluation is the relative WCET achieved by the EAC+ analysis compared to the CCN approach 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 use box plots to visualize the results. The line in the middle of each box represents the median value, while 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 dash. The metrics are evaluated for each task and grouped for each system configuration. The configuration groups are named using the number of cores (2, 4, or 8) and the size of the shared cache (4, 8, 16, or 32 KB).

7.1.1 Dual-Core Systems. Figure 4 shows the hit ratio for dual-core systems. For the 4 KB cache, the median hit ratio is 0% for the baseline CCN analysis. Here, the EAC+ analysis achieves significant improvements with a median hit ratio of 76%. For the 8 KB cache, the

CCN analysis is able to achieve a hit ratio of 74%. Again, the EAC+ analysis is able to outperform the CCN method and yields a median hit ratio of 88%. In the largest cache configuration for dual-core systems, the EAC+ approach achieves only a small improvement over the CCN approach. Both classification methods achieve a hit ratio of around 90%. This result indicates that a 16 KB cache is large enough for the considered dual-cores systems relative to the contained code size so that inter-core interference does not lead to a substantial degradation of the worst-case timing behavior.

In Figure 5, the WCET of the EAC+ analysis is compared to the CCN analysis. The first three box-plots show the results for dual-core systems. For the smallest cache size of 4 KB, the EAC+ analysis yields an average WCET improvement of 14.4% (11.2% median). Using a larger cache, this gap shrinks as the CCN analysis is able to classify more accesses as cache hits. For an 8 KB shared cache, the average WCET reduction is 9.8% (2.8% median). In the largest cache configuration of 16 KB, no median WCET reduction is achieved. Again, this is due to the fact that the 16 KB cache is large enough to completely contain the code for most dual-core systems.

However, there is still an outlier which experienced a WCET reduction of 60%. This outlier occurred in a system containing the tasks `aiifft01` and `bitmnp01`. The code size of `bitmnp01` is around 27 KB. Thus, the CCN approach is unable to classify any accesses belonging to `aiifft01` as a cache hit. However, the EAC analysis was able to classify 95% of accesses contained in `aiifft01` as cache hits. This discrepancy leads to the large reduction in WCET.

7.1.2 Quad-Core Systems. A different landscape unfolds for systems containing four cores. As can be seen in Figure 6, the cumulative code size of the tasks contained in the systems is so large that it prohibits the CCN analysis from making any useful conclusions about cache hits. The median hit ratio is 0% even for the largest 16 KB cache. In contrast to this, the EAC+ analysis presented in this paper is still able to classify many accesses as cache hits. The median hit ratio values for 4 KB caches is 41%; for 8 KB it is 71%; and for 16 KB it is 78%. This substantial improvement in access classification is also reflected in the relative WCET values, shown in Figure 5. The three box plots on the right show the WCET reduction for quad-core systems. The EAC+ analysis reduces the WCET by 10.2% on average (4.0% median) for 4 KB caches. The performance gap increases to an average 17.5% (11.5% median) for 8 KB caches. For 16 KB caches, the average relative WCET reduction is 16.7% (12.2% median).

These results demonstrate the necessity of a precise analysis of the inter-task cache interference. Without precise information on the potential cache interference, the cache becomes useless in the worst-case as almost no access can be classified as a cache hit using the standard classification method.

7.1.3 Scalability to Octa-Core Systems. To explore the scalability of the analysis, we also performed an evaluation of octa-core systems. The size of the shared cache was increased to 32 KB for this evaluation. All other parameters remained unchanged. We evaluated 10 task sets. The results are shown in Figure 7.

As was the case for quad-core systems, the CCN analysis was not able to classify any significant amount of accesses as cache hits. The hit ratio for almost all tasks is 0%. Again, the EAC+ analysis is able to perform substantially better with a median hit ratio of 49.6%.

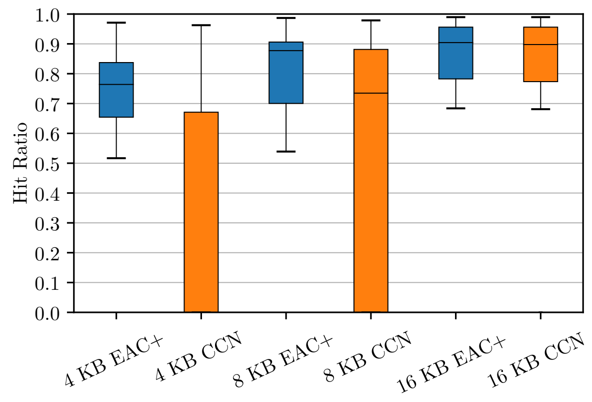


Figure 4: Dual-core cache hit ratio of the EAC+ (blue) and CCN (orange) analyses for different cache configurations.

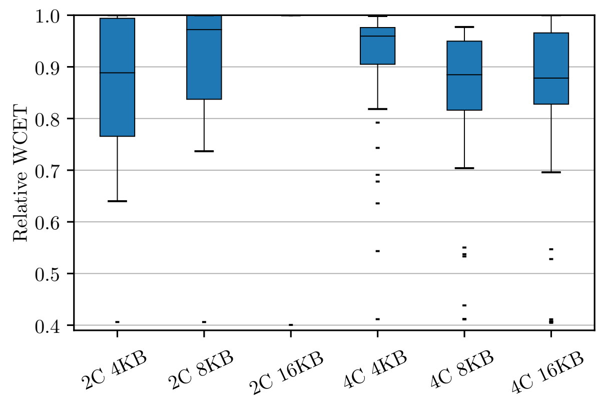


Figure 5: Relative WCET values derived using the EAC+ analysis in relation to the CCN analysis.

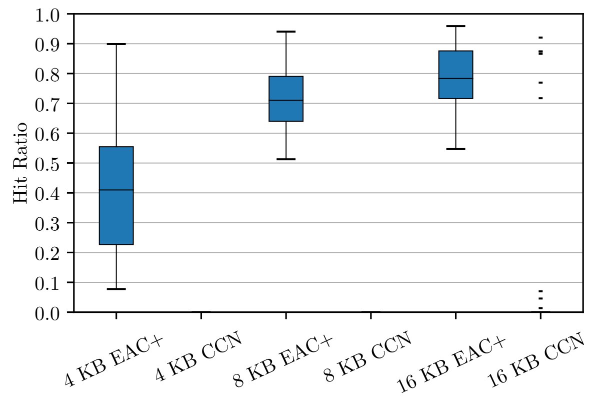
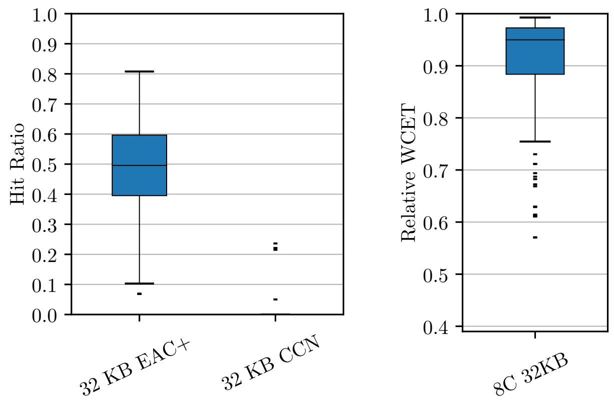


Figure 6: Quad-core cache hit ratio of the EAC+ (blue) and CCN (orange) analyses for different cache configurations.



(a) Hit ratio **(b) Relative WCET**

Figure 7: Hit ratio of the EAC+ (blue) and CCN (orange) analysis, as well as the relative WCET for octa-core systems.

This increase in hit classifications resulted in an average relative WCET reduction of 11.3% (5.1% median), while the maximal WCET reduction for a task was 43%.

Hence, we conclude that the proposed analysis is also applicable to multi-core systems with a high core count and yields notable improvements over the baseline analysis.

7.2 Runtime Evaluation

In this subsection, we take a look at the runtime overhead that is required by the proposed analysis. The evaluations were conducted on an Intel Xeon Server containing 46 cores running at 3.2GHz. All analyses were configured to use only a single processor core.

The runtime of the EAC+ 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 event-arrival curves, *(c)* the DFA to classify accesses, *(d)* the final WCET analysis. The runtime of the CCN method includes determining the cache block conflict number for every cache set and a WCET analysis. A table containing the measured runtimes is shown in Table 3.

As the analysis is based upon an isolated per-core WCET analysis it is expected that the runtime scales linearly in the number of cores. Additionally, the size of the shared cache also has a linear impact on the number of ILPs that potentially need to be solved to determine the event-arrival curves. These expectations are matched in the measured runtimes. The time required on average for an analysis of a dual-core system ranges from 3.5 to 4.9 minutes. We observed a small decrease in the runtime of the 16 KB configuration compared to the 8 KB configuration. This runtime decrease occurred during the derivation of the event-arrival curves. As the cache size increased, the traffic on each individual cache set decreased. This resulted in a more efficient event-arrival curve derivation process and caused a small drop in the required analysis time. For quad-core systems the runtime increased to 7.9 to 11.9 minutes. The average runtime of an 8-core system was 24.7 minutes. Note that the 8-core systems were also equipped with a larger 32 KB shared cache.

Table 3: Average analysis runtime in minutes.

Cores	Cache Size	EAC+	CCN
2	4 KB	3.5	0.5
	8 KB	4.9	0.8
	16 KB	4.4	0.9
4	4 KB	7.9	1.2
	8 KB	11.8	1.9
	16 KB	11.9	2.2
8	32 KB	24.7	6.1

In the EAC+ analysis, a large proportion of the time was spent to solve the ILPs formulated in Section 4 to derive the event-arrival curves. As the ILPs are independent of each other, this step could be parallelized to reduce the time requirement. The data-flow analysis presented in Section 6 was very quick, with an average runtime of less than one second per system.

The average EAC+ analysis runtime is larger than the time required by the CCN method by a factor of 4× to 7×. This increase in runtime is however justifiable, as the CCN method did not produce any useful cache hit classifications for 5 out of the 7 system configurations. Using the CCN Classifications, the median hit ratio was higher than 0% only for dual-core systems with a cache size of 8 KB and 16 KB. The presented analysis technique, however, achieved significant improvements in the number of hit classifications and the WCET estimate.

7.3 Propagation Limit Sensitivity

To ensure that the data-flow analysis presented in Section 6 converges quickly, the propagation of abstracted path information over nodes in the CFG is limited by a threshold value as described in Section 6.1. When the number of nodes a path is propagated over exceeds the threshold value, the path is considered to potentially cause a cache miss. In this subsection, we explore whether this cut off threshold is necessary and how it affects the analysis precision.

To achieve the results shown in the previous sections, we utilized a threshold value of 30. This means, that potential-hit paths containing up to 30 nodes may be recognized by the DFA as resulting in a cache-hit. Paths were not investigated beyond this threshold and were considered a potential miss instead. To explore the sensitivity of the analysis to this parameter, we also performed the analysis of dual-core and quad-core systems with different threshold values.

We first decreased the propagation limit 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. For quad-core systems with a 16 KB shared cache, the largest average hit ratio reduction occurred. The average hit ratio reduced by 1.5% from 78.3% to 76.8%. Thus, a lower threshold only has a minor impact on the precision.

To check whether a higher propagation limit is useful, we tried increasing the limit to 150. The biggest increase in the average hit ratio was observed for dual-core systems with a 4 KB cache.

The hit ratio increased by 0.5% from 74.9% to 75.4%. These results suggest that accesses to the shared cache exhibited highly localized behavior.

The average runtime of the data-flow analysis for quad-core systems was 1.2 seconds with the propagation limit set to 30. Decreasing the limit to 5 reduced the runtime to 0.24 seconds. While increasing the limit to 150 caused the analysis to require 9.6 seconds on average. Removing the propagation limit altogether caused 8 of the 30 analyzed quad-core systems to not terminate after 2 hours. Thus, we conclude that the propagation limit is necessary, but the analysis precision itself is not very sensitive to the choice of the parameter value.

7.4 Relative Precision Evaluation

In Section 6.2, we note that the precision of the EAC and the CCN analysis techniques are not comparable as there are situations in which either of the two analyses is more precise than the other one. To evaluate whether this theoretical consideration manifests itself in practice, we also performed the evaluation of dual-core and quad-core system using only the classifications derived by the DFA without using the CCN classification as the fallback.

For dual-core systems, we observed a difference in the analysis precision between the different classification strategies. The median hit ratio of the EAC classification for 4 KB caches was 73%. This is similar to the 76% which were achieved by EAC+. The median hit ratio of CCN for this configuration was 0%. For 8 KB caches, the median hit ratio using EAC was 77%. Recall that the CCN approach yielded a 74% median hit ratio, while the combination of the two classification methods yielded a median hit ratio of 88%. The combination of the two hit classification methods thus performed better than each method did when applied separately.

For the largest cache configuration of 16 KB, the CCN analysis classified more accesses as cache hits than EAC. The median hit ratio for EAC was 82%, while both EAC+ and CCN achieved around 90%. Thus, EAC performed notably better than CCN for the smallest cache size, while CCN had higher performance than EAC for the 16 KB cache.

For quad-core systems, however, we observed that there was no significant difference between the performance of the EAC and EAC+ classifications. The reason for this is that while the CCN analysis may be more precise in some situations in theory, for the evaluation setup shown in this paper, the CCN analysis was unable to generate hit classifications in almost all situations (see Figure 6). Thus, for quad-core systems, solely using the EAC classifications instead of the EAC+ combination, which also makes use of the CCN classifications, did not result in decreased analysis precision.

These results demonstrate that there are situations in which either the EAC or CCN technique performs better, while the combination of both classification approaches EAC+ performs the best.

8 CONCLUSION

In this paper, we presented a novel analysis perspective for shared caches accessed via a round-robin arbitrated bus. In the analysis, the inter-task interference between tasks running on different cores is expressed using event-arrival curves. These curves quantify how much time will pass between accesses to multiple conflicting cache blocks. This perspective enables us to view inter-task cache interference as a function of time. Furthermore, we presented a data-flow analysis with which the temporal reuse distance of a cache block can be determined. These two components, the event-arrival curves and the temporal reuse distance, allow the analysis 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 2, 4, and 8 cores and shared caches ranging from 4 KB to 32 KB. The presented analysis significantly outperforms the baseline analysis [8] in most situations. The baseline analysis collects all potentially accessed cache blocks and assumes that these blocks may be accessed at any time. This pessimistic assumption caused the analysis to not produce any substantial amount of hit classifications in 5 out of the 7 considered system configurations. Compared to this standard analysis, the presented analysis performed particularly well in systems with a small cache size relative to the total program size.

Further research could expand the work presented in this paper by analyzing systems containing more than one task per core. Another avenue for future research is the applicability of event-arrival curves to replacement policies other than LRU. To reduce the required runtime of the analysis, it is conceivable to compute safe approximations of the event-arrival curves instead of precise curves. This trade off between runtime and precision could be explored in future work.

REFERENCES

[1] Patrick Cousot and Radhia Cousot. 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 1977)*. 238–252. https://doi.org/10.1145/512950.512973

[2] P. Padma Priya Dharishini and P. V. R. Murthy. 2021. Precise Shared Instruction Cache Analysis to Estimate WCET of Multi-threaded Programs. In *2021 IEEE 18th India Council International Conference (INDICON)*. 1–7. https://doi.org/10.1109/INDICON52576.2021.9691620

[3] Heiko Falk and Paul Lokuciejewski. 2010. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems* 46, 2 (2010), 251–300. https://doi.org/10.1007/s11241-010-9101-x

[4] Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems* 17, 2 (1999), 131–181. https://doi.org/10.1023/A:1008186323068

[5] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET Benchmarks – Past, Present and Future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, Björn Lisper (Ed.). OCG, Brussels, Belgium, 136–146. https://doi.org/10.4230/OASIcs.WCET.2010.136

[6] Damien Hardy, Thomas Piquet, and Isabelle Puaut. 2009. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *30th IEEE Real-Time Systems Symposium (RTSS)*. 68–77. https://doi.org/10.1109/RTSS.2009.34

[7] Yau-Tsun Steven Li and Sharad Malik. 1997. Performance Analysis of Embedded Software Using Implicit Path Enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 16, 12 (1997), 1477–1487. https://doi.org/10.1109/43.664229

[8] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. 2012. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. *Real-Time Systems* 48, 6 (2012), 638–680. https://doi.org/10.1007/s11241-012-9160-2

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

[10] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. 2019. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys (CSUR)* 52, 3 (2019), 1–38.

https://doi.org/10.1145/3323212

[11] Kartik Nagar. 2016. *Precise analysis of Private and Shared Caches for tight WCET Estimates.* Ph. D. Dissertation. Indian Institute of Science Bangalore.

[12] Kartik Nagar and Y. N. Srikant. 2014. Precise Shared Cache Analysis using Optimal Interference Placement. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 125–134. https://doi.org/10.1109/RTAS.2014.6925996

[13] Dominic Oehlert. 2021. *Worst Case Execution Time Oriented Code Optimization of Hard Real-Time Multicore Systems.* Ph. D. Dissertation. Technische Universität Hamburg.

[14] Dominic Oehlert, Selma Saidi, and Heiko Falk. 2018. Compiler-Based Extraction of Event Arrival Functions for Real-Time Systems Analysis. In *30th Euromicro Conference on Real-Time Systems (ECRTS)*. 4:1–4:22. https://doi.org/10.4230/LIPIcs.ECRTS.2018.4

[15] The Embedded Microprocessor Benchmark Consortium. 2023. *About the EEMBC AutoBench™ Performance Benchmark Suite.* EEMBC. Retrieved 2023-01-20 from https://www.eembc.org/autobench/

[16] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2019. Fast and Exact Analysis for LRU Caches. *Proceedings of the ACM on Programming Languages (POPL)* 3 (2019), 54:1–54:29. https://doi.org/10.1145/3290367

[17] Wei Zhang, Mingsong Lv, Wanli Chang, and Lei Ju. 2022. Precise and Scalable Shared Cache Contention Analysis for WCET Estimation. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*. 1267–1272. https://doi.org/10.1145/3489517.3530613