Fast and Precise Worst-Case Interference Placement for Shared Cache Analysis

KARTIK NAGAR and Y. N. SRIKANT, Indian Institute of Science

Real-time systems require a safe and precise estimate of the worst-case execution time (WCET) of programs. In multicore architectures, the precision of a program's WCET estimate highly depends on the precision of its predicted shared cache behavior. Prediction of shared cache behavior is difficult due to the uncertain timing of interfering shared cache accesses made by programs running on other cores. Given the assignment of programs to cores, the worst-case interference placement (WCIP) technique tries to find the worst-case timing of interfering accesses, which would cause the maximum number of cache misses on the worst case path of the program, to determine its WCET. Although WCIP generates highly precise WCET estimates, the current ILP-based approach is also known to have very high analysis time. In this work, we investigate the WCIP problem in detail and determine its source of hardness. We show that performing WCIP is an NP-hard problem by reducing the 0-1 knapsack problem. We use this observation to make simplifying assumptions, which make the WCIP problem tractable, and we propose an approximate greedy technique for WCIP, whose time complexity is linear in the size of the program. We perform extensive experiments to show that the assumptions do not affect the precision of WCIP but result in significant reduction of analysis time.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Shared cache analysis, worst-case execution time estimation, multicore real-time systems

ACM Reference Format:
Kartik Nagar and Y. N. Srikant. 2016. Fast and precise worst-case interference placement for shared cache analysis. ACM Trans. Embed. Comput. Syst. 15, 3, Article 45 (March 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2854151

1. INTRODUCTION

Multicores are widespread in today's computing devices, from handheld mobiles to servers and workstations. Using multicores for real-time systems has proved difficult, because real-time systems require an estimate of the maximum execution time of programs (also referred to as the worst-case execution time (WCET)), and obtaining precise estimates of WCET on multicore architectures is not easy.

After decades of research, a standard framework for estimating the WCET of programs has emerged [Wilhelm et al. 2010]. Starting from the binary executable whose WCET is to be determined, first the control-flow graph (CFG) of the program is constructed and the loop bounds are estimated. Next, the microarchitectural analysis is

Kartik Nagar was supported by Microsoft Corporation and Microsoft Research India (under the Microsoft Research India Ph.D. Fellowship Award) and the IMPECS Project.
Authors' addresses: K. Nagar (corresponding author) and Y. N. Srikant, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India; emails: {kartik.nagar, srikant}@csa.iisc.ernet.in.

DOI: http://dx.doi.org/10.1145/2854151

carried out to determine the impact of architectural components, such as caches and pipelines, on the execution time of individual instructions. Finally, the program path with the maximum execution time, i.e., the worst-case program path, is determined, which realizes the WCET of the program.

Caches have a major impact on the execution time, and hence cache analysis is a crucial component of the WCET estimation framework. Multicore architectures typically have a cache hierarchy, with private caches assigned to each core and a shared cache that is accessed by all of the cores. Accesses that miss the shared cache have to go to the off-chip main memory, and due to the high difference between main memory latency and shared cache latency in current architectures, shared cache analysis becomes very important.

The purpose of shared cache analysis is to statically identify the shared cache hits experienced by a program so that the shared cache latency can be used for those accesses, thus lowering the estimated WCET. However, if we want to determine the shared cache behavior of a program running on one of the cores, then we must consider the effect of the shared cache accesses generated by other cores (henceforth called *interfering accesses* or *interferences*). This is because such interfering accesses can evict the cache blocks of the program under analysis, causing extra shared cache misses, which the program would not have suffered in isolation.

The shared cache misses caused due to interferences will cause an increase in the program's execution time, which must be accounted for while determining its WCET. The WCET of a program in a multicore environment should be greater than the actual execution time of the program in all its runs, i.e., irrespective of the program input, and the timing of the interferences from other cores. Although it would be safe to assume that all shared cache accesses cause a miss, this would introduce significant imprecision in the WCET estimate. Our goal is to safely predict as many shared cache hits as possible so that precise WCET estimates can be obtained.

Given the assignment of programs to cores, we can find the number of interferences generated by each core by performing standard cache analysis of the private caches. However, an interference to the shared cache can cause any number of misses between 0 and the cache associativity, depending on the timing of its arrival. Hence, the same number of interferences can cause different numbers of cache misses for the program under analysis. The worst-case arrival of interferences is the arrival that causes the maximum number of shared cache misses and therefore the maximum increase in execution time. The worst-case interference placement (WCIP) technique tries to find the worst-case interference arrival along the worst-case path in the program [Nagar and Srikant 2014].

WCIP is safe and theoretically is the most precise method to estimate the shared cache behavior, because one must consider the possibility of a program run, which will traverse the worst-case path and experience the worst-case arrival of interferences from other cores, and thus have an execution time equal to the WCET as calculated using WCIP.[1] In practice, the WCETs obtained using WCIP are much lower than other techniques used for shared cache analysis.

Previous approaches to shared cache analysis [Hardy et al. 2009] typically find the local worst-case arrival of interferences for every individual cache hit, whereas WCIP aims to find the global worst-case arrival for all accesses on the worst-case path. An important advantage of this strategy is that the number of estimated cache misses becomes directly proportional to the number of interferences. This is an important

[1]Note that because of infeasible paths, imprecision of private cache analysis, and so forth, it is possible that the actual WCET of the program may be lower than the WCET obtained using WCIP. However, this issue is orthogonal to WCIP, which itself will not introduce any imprecision.

property, because for cache hits inside loops, it is possible that the interferences from other cores may not be enough to cause misses in all iterations but may only cause misses in a subset of the iteration space. Previous approaches would not be able to handle such cases, but WCIP can identify them and accordingly find the maximum number of misses.

In Nagar and Srikant [2014], WCIP is performed by generating an integer linear program (ILP), whose optimal solution encodes the worst-case program path and the WCIP on this path. Solving an ILP is an NP-hard problem, and this is reflected in the fact that ILP-based WCIP fails to produce WCETs for some programs in a reasonable duration. This raises the challenge of finding efficient techniques to perform WCIP. In this work, we show that performing WCIP is a NP-hard problem. Specifically, we show that finding the worst-case path in a program, in the presence of interferences to the shared cache, is NP-hard by reducing the 0-1 knapsack problem (KP).

The difficulty in WCIP arises from the difference in the execution times and number of shared cache hits, along different program paths. A program path with high execution time may not have enough shared cache hits to "use" all of the interferences, whereas there may be program paths with a large number of shared cache hits but lower execution times. To bypass this problem, we assume that all shared cache hits are present on the worst-case path (calculated assuming no interferences). We then propose a simple greedy algorithm to perform WCIP, which simply picks those cache hits in the program that have the highest chance of becoming misses, as the cache blocks that they access are already close to eviction without interferences. The total time complexity of our approach is linear in the program size.

We have implemented our technique in the Chronos WCET analyzer and tested it on benchmarks from the Mälardalen, MiBench, and PapaBench benchmark suites. The results show that the approximate technique for WCIP is comparable to ILP-based WCIP in terms of precision of the WCET estimates. Both techniques are far superior than earlier approaches to shared cache analysis [Hardy et al. 2009], with an average precision improvement of 27.5% in the WCETs obtained using WCIP. The major advantage of approximate WCIP over ILP-based WCIP is in the analysis time, where ILP-based WCIP fails to compute the WCET for some benchmarks in any reasonable duration of time, whereas the approximate WCIP requires a maximum of 5 seconds across all benchmarks.

2. SHARED CACHE ANALYSIS

In this section, we provide a quick overview of the cache terminology and review some of the existing works in shared cache analysis. Caches store a small subset of the main memory closer to the processor and provide fast access to its contents. All transfer between the main memory and cache takes place in equal-sized chunks called *memory blocks* (or *cache blocks*). To enable fast lookup, caches are divided into cache sets. For an A-way set-associative cache, each cache set can contain a maximum of A cache blocks. When the processor performs a memory access, the cache subsystem first finds the unique cache set to which the accessed cache block is mapped, then searches for it among the (at most) A cache blocks in the cache set; if it is not present, it brings it from the main memory (or higher-level caches). A multilevel cache hierarchy has independent caches at different levels, generally with smaller caches being closer to the processor.

Since the total number of cache blocks mapped to a cache set will usually be much greater than the associativity (A), the cache replacement policy decides which cache block should be evicted if the cache set is full and a new cache block has to be brought in. The least recently used (LRU) policy orders all cache blocks in a cache set according to their most recent accesses and evicts the cache block that was accessed farthest in the

past. On a memory access, caches are searched in increasing order of cache levels, and the accessed memory block is brought into every cache level that has been searched.

Must analysis [Ferdinand and Wilhelm 1999] for private caches is an abstract interpretation (AI)-based technique, which finds those cache blocks that are guaranteed to be in the cache across all executions of the program. Accesses to such cache blocks can be safely predicted as cache hits. Most of the shared cache analysis techniques [Hardy et al. 2009; Chattopadhyay et al. 2012; Chattopadhyay and Roychoudhury 2011; Li et al. 2009] build on top of Must analysis and find shared cache accesses that are guaranteed to hit the cache irrespective of when the interferences arrive.

To do this, the shared cache states are first determined using Must analysis, assuming no interferences. Then, the shared cache state at each program point is modified by considering the effect of *all* interferences generated by interfering programs running on other cores, and the modified states are used for predicting cache hits. These approaches introduce a lot of imprecision and, in most cases, classify all shared cache accesses as misses.

Yan and Zhang [2008] introduced the "always-except-one" classification for instructions inside loops that access the shared cache if the interfering access is not inside a loop. However, their approach works only for direct-mapped (DM) caches, and they also assume the effect of interfering accesses at all program points. In a later work [Yan and Zhang 2009], they take into account the sequence of accesses to rule out certain misses arising due to infeasible interfering accesses. However, the feasible interferences could still occur anywhere in the program.

Hardware approaches [Suhendra and Mitra 2008; Paolieri et al. 2009; Ward et al. 2013] focus on making the multicore architecture prediction-friendly by using techniques such as cache locking and cache partitioning. Such techniques make it safe to assume that no interfering accesses arrive while performing the hit-miss analysis of the shared cache, thus making it as precise as private cache analysis. However, the restrictions imposed may result in a waste of resources and require support from the hardware.

In Nagar and Srikant [2014], we proposed the WCIP approach for shared cache analysis (referred to in that paper as optimal interference placement), which tries to find an assignment of interferences to program points, which will cause the maximum number of shared cache misses on the worst-case path. Instead of considering the effect of all interferences on the shared cache state at every program point, WCIP only considers the effect of the interferences assigned at a program point to modify the shared cache state at that point. Hence, the effect of the same interference is not considered multiple times, and this results in a larger number of shared cache accesses classified as hits despite the interferences from other cores. Ultimately, this results in much lower WCET estimates for multicore architectures, which are nonetheless safe. For details on the ILP-based approach for WCIP, we refer the reader to Nagar and Srikant [2014].

The problem of determining the impact of co-running programs on the cache behavior has been studied earlier for a multitasking pre-emptive environment. For such cases, the cache-related pre-emption delay (CRPD) is calculated by determining the maximum number of cache misses caused by pre-empting tasks [Altmeyer et al. 2010]. However, this problem is fundamentally different from shared cache analysis in a multicore architecuture. In the CRPD problem, since all pre-empting tasks finish their execution completely, and only then the pre-empted task is allowed to run, CRPD calculation only needs to consider the impact of all interfering accesses at a single program point in the program under analysis, i.e., at the pre-emption point. On the other hand, in a multicore architecture, the interfering tasks are running simultaneously along with the program under analysis, and hence the impact of interfering accesses must

be considered at multiple program points. This raises issues such as considering the cumulative effect of interfering accesses at different program points and ensuring that interfering accesses are not counted more than once.

3. ASSUMPTIONS

In this work, we make the following assumptions. We assume a timing anomaly-free architecture [Lundqvist and Stenström 1999], as our approach is targeted toward maximizing the number of cache misses, and hence cache misses must cause the maximum increase in final execution time. The cache hierarchy is noninclusive, so evictions at different levels are independent of each other, with the cache replacement policy being LRU at all levels. Each core has one (or more) private caches and a shared cache (shared between all cores) closest to the main memory. We limit our attention to instruction caches and assume separate instruction caches at all levels. Even for private caches, data cache analysis is less precise than instruction cache analysis [Ramaprasad and Mueller 2005] due to the imprecision of address analysis, which results in a set of accessed cache blocks for each instruction. Although there have been some efforts at data cache analysis for shared caches [Lesage et al. 2010], and it may be possible to use WCIP for shared data cache analysis, it will require a nontrivial extension and is beyond the scope of this work.

Even though our approach can be applied to the shared cache at any level (with the restriction that all lower levels should be private), for simplicity we will assume a two-level cache hierarchy, with private L1 caches and a shared L2 cache. In the rest of the article, when we say *cache access*, we mean an access to the L2 cache, and when we say *cache hit*, we mean a memory access that hits the L2 cache. We assume a shared bus interconnecting all cores to the shared cache (and the main memory). We note that even though the shared bus can cause extra delays to any shared cache access, it will interfere neither with the internal workings of the cache nor with the memory accesses made to the shared cache.

Our technique requires knowledge about the maximum number of interferences that can be generated by all other cores during a single execution instance of the program under analysis. For this, we only need to know the mapping of tasks to cores, as the number of interferences generated by a task can be determined using AI-based analysis of the private caches. If programs are periodic, then the maximum number of instances of an interfering program, during a single instance of the program under analysis, may need to be determined.

4. COMPLEXITY OF WCIP

Given the WCET and shared cache behavior of the program in isolation, as well as the information about interferences to the shared cache generated by other cores, the WCIP problem is to find the maximum increase in the WCET due to the shared cache misses caused by interferences. This can be decomposed into two interdependent problems: (1) find the program path with the maximum execution time in the presence of interferences and (2) find a distribution of interferences on this program path that causes the maximum number of shared cache misses. A distribution of interferences will assign disjoint subsets of interferences at each program point in the path. It is only essential to keep track of the number of interferences assigned at a program point.

To find the worst-case path in the presence of interferences, we must know the worst-case distribution that will cause the maximum increase in the execution time of the path. However, to find this worst-case distribution, we have to know the entire path along which the interferences are to be distributed. A naive approach to WCIP would be take every complete program path from the beginning of the program to its end,

find the worst-case distribution and hence the WCET of the path in the presence of interferences, and then select the path with the maximum WCET.

Finding the worst-case path in a program without any shared cache interferences is not a difficult problem. The cache behavior can be estimated without interferences using the AI-based techniques, which would lead to an estimate of the WCET of each basic block in the program. Given the WCET of each basic block, and the program CFG, there are techniques that can find the WCET of the program in time polynomial in the size of the CFG by using a scoped version of Dijsktra's algorithm to find longest path in directed acyclic graphs [Althaus et al. 2011]. Interferences will cause shared cache misses and increase the WCET of basic blocks, and we will show that finding the worst-case path in the presence of interferences becomes NP-hard.

To focus on the problem of finding the worst-case path, we will make the worst-case distribution problem simpler by assuming a DM shared cache with a single cache set. DM caches contain a single cache block per cache set, and since we are assuming a single cache set, our entire cache will contain only one cache block, which will be the most recently accessed block. A cache hit will occur when the block present in the cache is accessed by the program. An interference from another core could evict the block in the cache and thus cause a cache miss if there is an access to the evicted cache block after the interference.

We define a straight-line program to be one without any loops or branches. For our purposes, a straight-line program simply consists of a sequence of accesses to the shared cache. For this simplified shared cache architecture, WCIP in a straight-line program becomes trivial.

LEMMA 1. *Given a straight-line program with H number of shared cache hits and B number of interferences coming from other cores, and assuming a DM shared cache with one cache set, the maximum number of shared cache misses caused due to interferences would be $min(H, B)$.*

Since at most one cache block will be present in the DM cache at a time, an interference can only affect the next access to this cache block. Hence, B interferences can cause at most B cache misses. If H is the number of shared cache hits in the program, and if $H \leq B$, then every cache hit will become a miss by assigning one interference before the access. On the other hand if $H > B$, then we can select any B cache hits and assign one interference before each selected cache hit, causing a total of B misses.

We now add one layer of complexity and consider programs with a single level of branching and no loops. An example of such a program is given in Figure 1. The program has n segments, where each segment is an if-then-else branch. For our purposes, each branch of a segment is just a sequence of shared cache accesses. t_i^l (t_i^r) is the execution time, without interferences, of the left (right) branch of the ith segment. h_i^l (h_i^r) is the number of shared cache hits in the left (right) branch of the ith segment. These are the accesses that are guaranteed to hit the shared cache without any interferences. Assume WLOG that $\forall i, t_i^l \geq t_i^r$. Hence, the WCET without interferences would be $\sum_{i=1}^{n} t_i^l$, obtained by taking the left branch of each segment.

For simplicity, assume a shared cache miss penalty of one cycle. If B interferences to the shared cache come from other cores, then they can cause at most B cache misses. Hence, if there are at least B cache hits among the left branches, then the WCET with interferences would be $\sum_{i=1}^{n} t_i^l + B$. However, if that is not the case, i.e., if $\sum_{i=1}^{n} h_i^l < B$, then the maximum execution time with interferences by picking the left-hand branch in each segment would be $\sum_{i=1}^{n}(t_i^l + h_i^l)$. If for some i, $h_i^r > h_i^l$, then by taking the right-hand branch, we would able to increase the execution time by $h_i^r - h_i^l - (t_i^l - t_i^r)$ by making use of $h_i^r - h_i^l$ extra interferences.

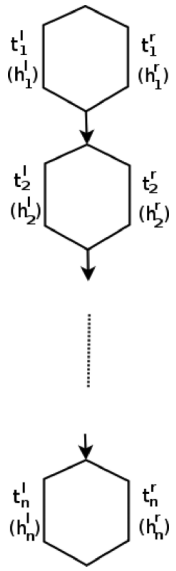

Fig. 1. A simple branched program.

Notice that the WCIP problem in this case boils down to finding the segments where the right-hand branch must be taken, i.e., finding the worst-case path in the program. Once the worst-case path is known, finding the distribution of interferences is trivial, as we can simply assign one interference before each cache hit on the path until we run out of interferences or cache hits.

We can now see the resemblance to the 0-1 KP, in which there are n objects each with value v_i and weight w_i and a total weight budget of W, and the problem is to select a subset of objects whose total weight is at most W and which maximizes the total value. Taking object i in KP as segment i in WCIP, selecting object i would be equivalent to selecting the right-hand branch of segment i, which would result in a "value" of $h_i^r - h_i^l - (t_i^l - t_i^r)$ with an associated "weight" of $h_i^r - h_i^l$. The total weight budget would be the extra interferences that were not used on the left-hand branches, i.e., $B - \sum_{i=1}^{n} h_i^l$.

The only issue with this reduction is that in WCIP, it is not necessary that all cache hits in a selected branch may be converted to cache misses due to interferences. In other words, $h_i^r - h_i^l - (t_i^l - t_i^r)$ is only the maximum value available by selecting the right-hand branch in segment i, and the worst-case distribution may result in a lower value, if it does not distribute interferences before all cache hits on the right-hand branch. On the other hand, in 0-1 KP, if an object i is selected, it is guaranteed to increase the total value by v_i. To solve this dilemma, we will use the fact that there always exists a worst-case distribution that will try to convert all cache hits to cache misses in a selected branch (Lemma 2) to formally show the reduction. Let us first define the decision version of the WCIP problem in the restricted setting.

WCIP Problem: Given a simple branched program $\mathcal{P}$ with n segments (as shown in Figure 1), an interference budget B and a target execution time T, the WCIP problem is to determine whether there exists a path in the program (represented by the function $\sigma : \{1, \ldots, n\} \rightarrow \{l, r\}$), and assigned interferences $b_1, \ldots, b_n$ on this path such that

$$\forall i, 1 \leq i \leq n, \quad b_i \leq h_i^{\sigma(i)} \tag{1}$$

$$\sum_{i=1}^{n} b_i \leq B \tag{2}$$

$$\sum_{i=1}^{n} t_i^{\sigma(i)} + b_i \geq T \tag{3}$$

An interference distribution that obeys Equations (1) and (2) is called a *valid distribution*. The following lemma states that given any path in the program and a valid interference distribution, there exists another path and a valid distribution, which will result in equal or higher execution time, and which will try to distribute interferences such that all cache hits in a selected segment will become misses.

LEMMA 2. *Given a program $\mathcal{P}$ with n segments (as shown in Figure 1) and an interference budget B, assume that $\sum_{i=1}^{n} h_i^l \leq B$. Given a path represented by σ and assigned interferences $b_1, \ldots, b_n$ that form a valid distribution, there exists another path $\hat{\sigma}$, and valid distribution $\hat{b}_1, \ldots, \hat{b}_n$, with the following properties:*

(1) If $\hat{\sigma}(i) = l$, then $\hat{b}_i = h_i^l$.
(2) There exists at most one j such that $\hat{\sigma}(j) = r$ and $\hat{b}_j < h_j^r$.
(3) $\sum_{i=1}^{n} t_i^{\sigma(i)} + b_i \leq \sum_{i=1}^{n} t_i^{\hat{\sigma}(i)} + \hat{b}_i$.

Property 1 guarantees that all cache hits in left-hand branches that are selected by $\hat{\sigma}$ will be converted to misses. It may not be possible to guarantee the same about the right-hand branches, because the number of cache hits on the right-hand branches can be arbitrarily high, but property 2 guarantees that there will be at most one right-hand branch, where all cache hits are not converted to misses. Property 3 shows that the new distribution will yield equal or higher execution time.

The proof is straightforward and uses a series of redistribution of interferences from the initial distribution to ensure that properties 1 and 2 are met while not decreasing the execution time. The main idea is that if all cache hits are not converted to misses in some selected segment, then interferences can be borrowed from some other selected segment, and this can be done until all hits becomes misses. We skip the proof here in the interest of space, but the complete proof is present in our technical report [Nagar and Srikant 2015b]. We can safely assume that any optimal distribution of interferences will follow the rules set down by Lemma 2. We now define the decision version of the 0-1 KP, which is known to be NP-hard [Martello and Toth 1990].

0-1 Knapsack Problem: Given a set of n items each with value v_i and weight w_i ($1 \leq i \leq n$), a weight budget W and a target value V, the 0-1 Knapsack problem is to determine whether there exists a subset P of items such that

$$\sum_{i \in P} w_i \leq W \tag{4}$$

$$\sum_{i \in P} v_i \geq V \tag{5}$$

We assume that the values and weights are positive integers. Given an instance of the KP, we first convert it to another KP where the weights are greater than the values for all items. Let v_m be the maximum value among all n items. In the new problem, item i will have the same value v_i and a new weight $w_i' = v_m w_i$. The weight budget will be $W' = v_m W$, whereas the target value remains V. Since $w_i > 0$, $v_m w_i \geq v_m \geq v_i$ for all i. It is easy to see that any solution of the original KP will also be a solution of the modified problem and vice versa.

We now construct a simple-branched program $\mathcal{P}_{KP}$ (Figure 2), along with the interference budget and target execution time, in such a way that the solution to the WCIP problem in this program corresponds to the solution of the modified KP. $\mathcal{P}_{KP}$ has n

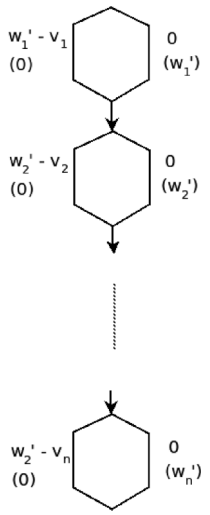

Fig. 2. Program $\mathcal{P}_{KP}$.

segments, with $t_i^l = w_i' - v_i$, $h_i^l = 0$, and $t_i^r = 0$, $h_i^r = w_i'$, for all i. The interference budget is $B = W'$, and the target execution time is $T = \sum_{i=1}^{n}(w_i' - v_i) + V$. Note that t_i^r can also be taken as any constant C, in which case t_i^l should be $w_i' - v_i + C$. The selection of execution times and shared cache hits should ensure that the "profit" of taking the right-side branch in segment i should be v_i.

Note that $\sum_{i=1}^{n}(w_i' - v_i)$ is the execution time of the program if the left branch is taken in all segments. However, no interferences can be used on the left branch, and taking the right branch in segment i will have a profit, i.e., increase in execution time, of v_i but associated weight, i.e., interferences used, of w_i'. Let us prove the reduction formally inthe following theorem.

THEOREM 1. *The WCIP problem in a simple-branched program, assuming a DM shared cache with single cache set, is NP-hard.*

PROOF. We need to show that $\exists P \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in P} w_i' \leq W'$ and $\sum_{i \in P} v_i \geq V \Leftrightarrow$. There exists a path σ and a valid interference distribution $b_1, \ldots, b_n$ such that $\mathcal{P}_{KP}$ achieves the target execution time $T = \sum_{i=1}^{n}(w_i' - v_i) + V$.

($\Rightarrow$) We are given a solution to KP, which achieves the target value. To obtain a solution to the WCIP problem, we simply pick the right-hand branch in those segments i where the corresponding item i has been selected in solution to KP. We will also assign interferences before all cache hits in the selected right-side branches. We define $\sigma : \{1, \ldots, n\} \rightarrow \{l, r\}$ and b_i as follows:

$$\sigma(i) = \begin{cases} r & \text{if } i \in P \\ l & \text{otherwise} \end{cases}$$

$$b_i = \begin{cases} w_i' & \text{if } i \in P \\ 0 & \text{otherwise.} \end{cases}$$

First, we need to show that this is a valid interference distribution. Equation (1) is trivially satisfied.

$$\sum_{i=1}^{n} b_i = \sum_{i \in P} w_i' \leq W'$$

This shows that Equation (2) is also satisfied (note that $B = W'$). We now show that this interference distribution achieves the target execution time T.

$$\begin{aligned}\sum_{i=1}^{n} t_i^{\sigma(i)} + b_i &= \sum_{i\in P} w_i' + \sum_{i\notin P} w_i' - v_i \\ &= \sum_{i\in P} (w_i' - v_i + v_i) + \sum_{i\notin P} w_i' - v_i \\ &\quad \text{(adding and subtracting } v_i,\ \forall i \in P) \\ &= \sum_{i=1}^{n} (w_i' - v_i) + \sum_{i\in P} v_i \\ &\quad \text{(rearranging terms)} \\ &\geq \sum_{i=1}^{n} (w_i' - v_i) + V = T\end{aligned}$$

Thus, $\mathcal{P}_{KP}$ achieves the target execution time under S and $b_1, \ldots, b_n$.

($\Leftarrow$) We are given σ, $b_1, \ldots, b_n$ such that the target execution time T is achieved and the interference distribution is valid, i.e., it satisfies Equations (1), (2), (3). Note that if $\sigma(i) = l$, then $b_i = 0$, and by Lemma 2, we can assume that there is at most one j such that $\sigma(j) = r$ and $b_j < w_j'$. To obtain the solution of KP, we will pick the item i if the right-hand branch has been taken in the corresponding segment i. Hence, $P = \{i | \sigma(i) = r\}$. First, we show that this selection of items meets the target value V.

$$\sum_{i=1}^{n} t_i^{\sigma(i)} + b_i \geq \sum_{i=1}^{n} (w_i' - v_i) + V$$

$$\Rightarrow \sum_{i:\sigma(i)=l} (w_i' - v_i) + \sum_{i:\sigma(i)=r} b_i \geq \sum_{i=1}^{n} (w_i' - v_i) + V$$

$$\Rightarrow \sum_{i:\sigma(i)=r} v_i \geq V + \sum_{i:\sigma(i)=r} (w_i' - b_i)$$

Since $\forall i, \sigma(i) = r \Rightarrow b_i \leq w_i'$, $\sum_{i:\sigma(i)=r} v_i \geq V$. Hence, we have shown that the target value is met. Next, we will show that $\sum_{i:S(i)=r} w_i' \leq W'$. We now consider two cases.

Case 1. If, $\forall i, \sigma(i) = r \Rightarrow b_i = w_i'$, then

$$\sum_{i:\sigma(i)=r} b_i \leq W' \Rightarrow \sum_{i:\sigma(i)=r} w_i' \leq W'.$$

Case 2. Suppose $\exists j$ such that $\sigma(j) = r$ and $b_j < w_j'$ (by Lemma 2, this means that $b_i = w_i'$ for all i such that $\sigma(i) = r$ and $i \neq j$). Note that in this case, $b_j > w_j' - v_j$. Otherwise, if $b_j \leq w_j' - v_j$, then we can modify the worst-case path to take the left-hand branch in segment j, i.e., $\sigma(j) = l$, which will only increase the execution time. Now, since for all other i, $\sigma(i) = r \Rightarrow b_i = w_i'$ implies the result (by case 1).

Hence, $b_j > w_j' - v_j \Rightarrow v_j > w_j' - b_j$. We know that $w_j' = v_m w_j$. We will now show that b_j should also be a multiple of v_m.

Again, in this case, $\sum_{i:\sigma(i)=r} b_i = W'$, because otherwise if $\sum_{i:\sigma(i)=r} b_i < W'$, we can increase b_j such that either b_j becomes w_j' or the sum becomes equal to W'. We will only

be increasing the execution time, and we have already proved the result if b_j becomes equal to w'_j (in case 1).

$$\sum_{i:\sigma(i)=r} b_i = W' \Rightarrow b_j = W' - \sum_{\substack{i:\sigma(i)=r \\ i\neq j}} b_i$$
$$\Rightarrow b_j = W' - \sum_{\substack{i:\sigma(i)=r \\ i\neq j}} w'_i$$
$$\Rightarrow b_j = v_m W - \sum_{\substack{i:\sigma(i)=r \\ i\neq j}} v_m w_i$$
$$\Rightarrow b_j = v_m p \quad (\text{where } p > 0)$$

However, $v_j > w'_j - b_j \Rightarrow v_j > v_m w_j - v_m p \Rightarrow v_j > v_m q$. This is a contradiction, as $q > 0$ and v_m is the maximum of all values. Therefore, there cannot exist j such that $\sigma(j) = r$ and $b_j < w'_j$. Hence, $\forall i, \sigma(i) = r \Rightarrow b_i = w'_i$. Hence, the set $P = \{i | \sigma(i) = r\}$ is a solution to KP. □

Although we have only looked at WCIP for simple-branched programs, assuming a DM cache with a single cache set, for the most general setting, we have to consider programs with nested branches and loops, and set-associative caches with multiple cache sets. These additions are only going to increase the complexity of the problem.

5. APPROXIMATE WCIP

5.1. Setup

We first perform AI-based multilevel Must and May cache analysis for the instruction cache hierarchy [Hardy and Puaut 2008] to obtain an initial cache access classification (CAC) and cache hit miss classification (CHMC) of all memory accesses at the shared cache level. The CAC can be one of Always, Uncertain, or Never. The CHMC can be Always Hit, Always Miss, or Uncertain. We consider all program accesses with a CAC of Always or Uncertain and a CHMC of Always Hit at the L2 cache.

Let Acc and Acc_H be the set of all instructions in the program that may access the L2 cache and are guaranteed to cause a L2 cache hit, respectively. Hence, for all $a \in Acc$, CAC(a) is Always or Uncertain, whereas for all $a \in Acc_H$, CHMC(a) is Always Hit. Clearly, $Acc_H \subseteq Acc$. Let $Age(a)$ be the age of the cache block accessed by a in the L2 Must cache at the program point just before the access. In an LRU cache, cache blocks in a cache set are given an age based on the timing of their last access. Hence, the most recently accessed cache block has an age of one, whereas the least recently accessed block will have an age of A (where A is the L2 cache associativity). We define the *eviction distance* of an access a to be $A - Age(a) + 1$. The eviction distance of an access is the minimum number of interferences required to evict the cache block just before the access. The concept of eviction distance is similar to the resilience of a cache block, as defined in Altmeyer et al. [2010]. For an access a referencing cache block m, if P_a is the program point just before the access a, and if $res_{P_a}(m)$ is the resilience of m at P_a, then the eviction distance of m would be $res_{P_a}(m) + 1$.

A cache hit path of an access is a program path along which the access will experience a cache hit [Nagar and Srikant 2014]. The cache hit path of an access a that references cache block m mapped to cache set s is a program path that begins with another instruction accessing m, ends with a, and accesses less than A distinct cache blocks mapped to s. For our purposes, the cache hit path π of an access will be represented by the set of shared cache accesses on the hit path (hence, $\pi \subseteq Acc$). Only those

interferences that occur along a cache hit path of an access need to be accounted for while determining the hit-miss classification of the access. If the total number of interferences occurring on a hit path of an access exceeds its eviction distance, then it will suffer a cache miss along that path.

Any access to the shared L2 cache made by a program will act as an interference to the program(s) running on other core(s). Hence, we count all actual accesses made by the interfering programs, i.e., the programs running on other cores, whose CAC at L2 is Always or Uncertain, to obtain the number of interfering accesses suffered by the program under analysis. If the access is inside a loop of the interfering program, then we use the loop bound to count the interferences caused by the access. In this way, we obtain the number of interferences, B_s and the number of interfering cache block B_s^{cb} to every cache set s. Let H be the actual number of cache hits of the program under analysis (obtained by counting every access in Acc_H, considering its loop bound if the access is inside a loop).

5.2. Motivation

The source of hardness for the WCIP problem (as shown in Section 4) lies in finding the worst-case path in the presence of interferences. One way to bypass this issue is to first find the worst-case path assuming no interferences (say π_{wc}) and then determine an upper bound on the increase in execution time due to interferences across all paths. Interferences will convert some of the shared cache hits into misses, but we cannot simply consider the shared cache hits present on π_{wc} to calculate the upper bound. Instead, a safe option would be to consider all shared cache hits present in the program and then find the maximum number of misses generated by interferences among them.

Let π_{wc}^{Int} be the worst-case path in the program in the presence of interferences, and let T_{wc}^{Int} and I^{Int} be its execution time without interferences and the maximum increase in its execution time due to interferences, respectively. Hence, $T_{wc}^{Int} + I^{Int}$ is the WCET of π_{wc}^{Int} (also the WCET of the program) in the presence of interferences. As mentioned earlier, π_{wc} is the worst-case path in the program in the absence of interferences, and let T_{wc} be its execution time.

Then, $T_{wc} \geq T_{wc}^{Int}$. Our strategy is to find the maximum number of cache misses due to interferences among all shared cache hits in the program and determine the resulting increase in execution time, I. Since this will also include all shared cache hits present on π_{wc}^{Int}, clearly $I \geq I^{Int}$. Hence, $T_{wc} + I \geq T_{wc}^{Int} + I^{Int}$.

The worst-case path and its WCET without interferences (T_{wc}) can be easily determined, and therefore our objective now is to calculate I, for which we need to determine the maximum number of cache misses that interferences can cause among Acc_H. We know that the eviction distance of a cache hit is the minimum number of interferences required to convert it to a cache miss. Hence, if the number of interferences assigned just before a cache hit is equal to its eviction distance, then the access can miss the cache.

It is easy to see that the optimal strategy to maximize the number of cache misses would be the greedy strategy of selecting cache hits in increasing order of their eviction distances. If cache hits with lower eviction distance are selected first, and interferences are assigned before them, then this would ensure that more interferences are available for later cache hits, thus maximizing the impact of every interference. However, before using this strategy, we must account for the overlapping effect of interferences.

5.3. The Overlapping Effect

We say that an interference affects a cache hit a when it increases the age of the cache block m that will eventually be accessed by a, without any intervening accesses to m

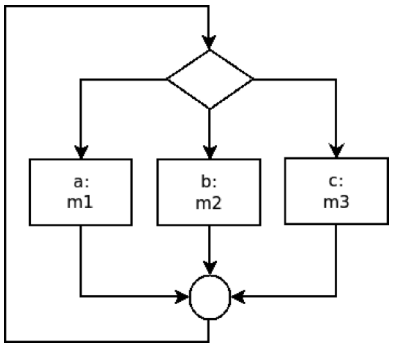

Fig. 3. Example to illustrate the overlapping effect.

between the interference and a. Obviously, any interference that occurs just before the cache hit a will affect it. However, any interference that occurs on a cache hit path of a will also affect a. Since the cache hit path can contain other cache hits, interferences that occur before these hits can also affect the access a. The implication is that the greedy strategy will not work, because it only considers the impact of those interferences that are assigned just before the cache hit.

As an example, consider the program fragment shown in Figure 3, containing instructions a, b, c that access cache blocks m_1, m_2, m_3, respectively, all mapping to the same cache set and hitting the cache. Assume that the cache associativity is 4. Since the age of all three cache blocks in the Must cache will be 3, their eviction distance will be 2. Now, any interference that occurs just before the access a is going to affect accesses b and c as well. If the sequence of accesses made by the program is $b-a-c-b-a-c-b-\ldots$, then assigning two interferences just before the access a will result in a cache miss for the second access to b. This shows that it is not enough to simply consider the interferences assigned just before the cache hit, but we must also consider the impact of interferences assigned before other cache hits.

The *overlapping factor* (OF) of a cache hit a is defined as the maximum number of cache hits that a single interference just before a can affect. An interference before a will affect the next access to any cache block that is present in the cache set just before a. If just before the cache hit a the cache set is full, i.e., it contains A cache blocks, then an interference before a can affect the next access to each of the A cache blocks. In the example program shown in Figure 3, the OF of each of the three cache hits $a, b,$ and c is 3. This is because an interference occurring before any of the three cache hits will affect all three.

We can use cache hit paths to calculate the OF. Only those interferences that occur within a cache hit path of an access can affect that access. Hence, if a cache hit h_1 is present in a hit path of another cache hit h_2, then any interference occurring before h_1 will affect h_2. The OF of cache hit h will be the number of cache hits that have a cache hit path that contains h. In our example program, a is present in the hit path $a-a$ of cache hit a, $b-a-b$ of b, and $c-a-c$ of c. Similar observations can be made about b and c.

Different values of OF and the eviction distance complicate worst-case distribution of interferences because we cannot directly use the greedy strategy of considering cache hits in increasing order of eviction distances. For example, as shown in Figure 4, consider the cache hits $h_1, h_2,$ and h_3 with eviction distances 2, 3, and 1, respectively. The cache hit paths of h_1 and h_2, h_2 and h_3 overlap. Both h_1 and h_2 have an OF of 2, whereas the OF of h_3 is 1. With an interference budget of 3, all three cache hits can be converted to misses by assigning two interferences before h_1 and one interference

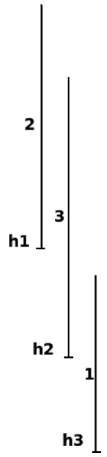

Fig. 4. Example showing that the greedy strategy of selecting cache hits in increasing order of eviction distance is not optimal.

before h_2. However, if we assign interferences based on increasing eviction distances, we will first assign one interference before h_3 and then two interferences before h_1, thus using up the interference budget and obtaining only two cache misses.

Other selection strategies, such as decreasing order of OFs or increasing order of eviction distance of overlapped cache hits, also do not work. This is where we make our second approximation, by removing the overlapping effect through an increase in the number of interferences. If a cache hit a has an OF of o, and if z interferences occur before a, then they will affect all o cache hits, and the effect is equivalent to having oz interferences and assigning z interferences individually before each cache hit.

Remember that B_s is the interference budget for cache set s. We find the maximum OF among all cache hits in the program mapped to s. If o_s is the maximum OF, then we take $o_s B_s$ to be the new interference budget for cache set s. We can now safely assume no overlapping while using the new interference budget. In other words, we can now assume that only those interferences that occur just before a cache hit will affect it. In the example of Figure 4, the maximum OF is 2, and hence the interference budget would become 6. Now, the eviction distance of each cache hit can be met, thus resulting in three cache misses.

In general, this is safe because any distribution of interferences with the original budget can be converted into a new distribution with the new budget and assuming no overlap. Hence, for the worst-case distribution with the original budget and overlap, there will also exist a distribution with the new budget, which will not use any overlap. We will find the worst-case distribution with the new budget and assuming no overlap.

Algorithm 1 is used to find the maximum OF for each cache set. It goes through every hit path of every cache hit and finds the OF of each cache hit, which is then used to find the maximum OF for the cache set. Since the size and number of hit paths are bounded, the inner for loop (lines 7 through 9) will run for a constant number of iterations. Hence, the algorithm has a complexity of $O(|Acc_H|)$, which will be linear in the code size. To find cache hit paths, we use a modified version of the AI-based analysis used to find cache miss paths, proposed in Nagar and Srikant [2015a]. The AI-based approach performs a constant number of traversals of the program CFG (upper bounded by the number of basic blocks) to find the fixpoint. Hence, the complexity of the approach is also linear in the size of the program.

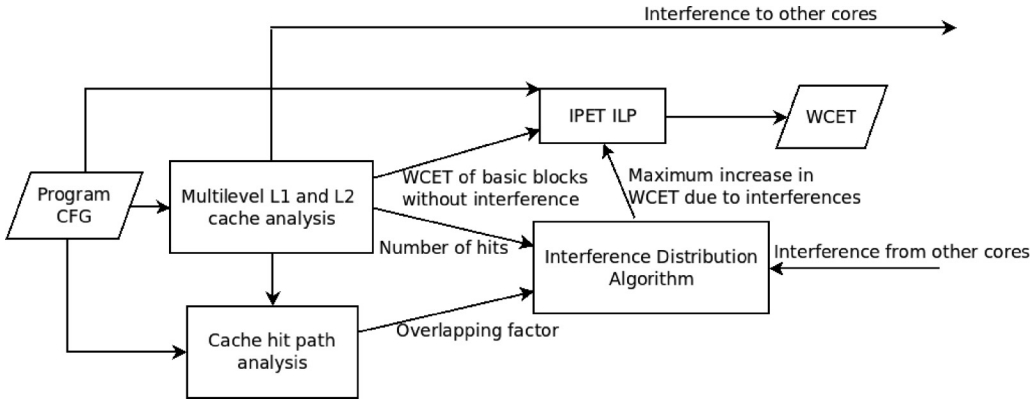

Fig. 5. Approximate WCIP.

ALGORITHM 1: Algorithm to Find the Maximum Overlapping Factor

Input: Cache hits mapped to each cache set s, hit paths of every cache hit
Output: Maximum OF o_s for every cache set s

```
1  for every cache set s do
2  |  o_s ← 0 ;
3  |  for every cache hit a ∈ Acc_H mapped to s do
4  |  |  OF_a ← 0
5  |  end
6  |  for every cache hit a ∈ Acc_H mapped to s do
7  |  |  for every hit a' ∈ Acc_H present in a hit path of a do
8  |  |  |  OF_a' ← OF_a' + 1
9  |  |  end
10 |  end
11 |  for every cache hit a mapped to s do
12 |  |  if OF_a > o_s then
13 |  |  |  o_s ← OF_a
14 |  |  end
15 |  end
16 end
```

5.4. Interference Distribution Algorithm

Since we are assuming that there is no overlapping effect, the optimal interference distribution strategy is to select cache hits in the increasing order of their eviction distances. Figure 5 shows the overall scheme for approximate WCIP. Given a program, we first perform the multilevel cache analysis and calculate the WCET of each basic block in the program, assuming no interferences. By performing the cache analysis, we can determine all shared cache hits in the program, as well as their eviction distances. We also obtain information about all shared cache accesses made by the program, which will act as interferences to programs running on other cores.

We find the cache hit paths of all cache hits and then find the maximum OF o_s for each cache set s using Algorithm 1. The cache hit information, OF, and the interference information is used by the interference distribution algorithm (explained next) to obtain the maximum increase in WCET due to cache misses caused by interferences. This is simply added to the WCET obtained by the implicit path enumeration technique (IPET) ILP to obtain the final WCET.

We call a cache hit having an eviction distance of k a $k-interference$ cache hit ($1 \leq k \leq A$). We count all $k-interference$ cache hits in the program using the L2 Must cache and loop bounds. Algorithm 2 shows our interference distribution strategy.

ALGORITHM 2: Interference Distribution Algorithm

Input: Number of interferences B_s, number of interfering cache blocks B_s^{cb} for each cache set s, number of k-interference cache hits $numhits_s^k$ in the program, for each cache set s ($1 \leq k \leq A$), OF o_s for each cache set, shared cache miss penalty c_p
Output: Maximum increase in WCET due to interferences, I

```
1  I ← 0;
2  for every cache set s do
3  |  B_s ← o_s B_s;
4  |  for k ← 1 to A do
5  |  |  if B_s^cb ≥ k then
6  |  |  |  if B_s ≤ k × numhits_s^k then
7  |  |  |  |  I ← I + (⌈B_s/k⌉ × c_p);
8  |  |  |  |  B_s ← 0;
9  |  |  |  else
10 |  |  |  |  I ← I + (numhits_s^k × c_p);
11 |  |  |  |  B_s ← B_s − (numhits_s^k × k);
12 |  |  |  end
13 |  |  end
14 |  end
15 end
```

The algorithm assigns interferences to cache hits in increasing order of their eviction distances for each cache set. First, we multiply the interference budget with the OF to get the new budget (line 3). Next, we check if the number of distinct cache blocks accessed by interferences is greater than k (line 5). If this is not the case, then no cache misses can caused by interferences, because $k-interference$ cache hits require interferences to at least k distinct blocks to become misses.

Then, we check whether there are enough $k-interference$ cache hits to use all interferences (line 6). If yes, then all interferences are assigned (k interferences before each cache hit), resulting in a maximum of $\lceil \frac{B_s}{k} \rceil$ misses. The cache miss penalty is added to the WCET for each of these misses, and the interference budget is updated to 0 (lines 7 and 8). If there are not enough $k-interference$ cache hits to use all interferences, then the cache miss penalty is added for all $k-interference$ cache hits and the interference budget is decreased (lines 10 and 11), and we continue the interference distribution in the next iteration with $(k+1)-interference$ cache hits and the remaining interferences.

5.5. Algorithm Analysis

For a shared cache with associativity A and number of cache sets S, Algorithm 2 has a time complexity of $O(SA)$. Combined with Algorithm 1, the total complexity of our approach is linear in the program size and the shared cache size. The algorithm introduces imprecision on account of the two approximations made to simplify the analysis. First, it assumes that all cache hits of the program are on the worst-case path without interferences. However, it is possible that this worst-case path may have very few cache hits, and non–worst-case paths with many shared cache hits may have small execution time without interferences. Second, we multiplied the original interference budget with the maximum OF o_s, with the inherent assumption that every interference to set s affects o_s cache hits, which may not be true.

In general, the algorithm will compute a maximum WCET increase of $(\sum_{\text{all sets s}} o_s B_s)c_p$ (if there are enough cache hits to use all interferences). An important property of the algorithm is that the maximum increase in execution time due to interferences is directly proportional to the number of interferences. This ensures that if the cache interference is low, then the increase in WCET due to cache interference will also be small. All previous approaches to shared cache analysis do not have this property. In addition, the increase in execution time due to interferences is a multidimensional, piecewise linear function of the number of interferences. Specifically, for some cache set s, if we plot the increase in WCET versus the number of interferences, then we will have a line with slope c_p until $numhits_s^1$ interferences, then a line with slope $\frac{c_p}{2}$ until $numhits_s^1 + 2numhits_s^2$ interferences, and so on.

Since we cannot find the worst-case path in the presence of interferences efficiently, we must make the worst-case assumptions about it, which translates to the worst-case assumptions about the number of shared cache hits, maximum OF, and eviction distance. As far as the approximations regarding the maximum OF, this value can be expected to be small (generally less than the cache associativity), because an interference at a program point will only affect the accesses to the cache blocks that are guaranteed to be present at that program point. In our experiments, the OF for most of the benchmarks was 1, and it never exceeded the cache associativity.

Instead of considering all shared cache hits in the program, we could also find the maximum number of cache hits that could happen on a program path. For this, we can use the IPET ILP, modifying the objective function to consider the number of cache hits in a basic block, instead of its execution time. In our experiments, this did not have any impact on the precision of WCIP. Note that to find the maximum OF, we must still consider all cache hits in the program.

Handling code sharing. We can simply ignore the effect of code sharing during our analysis, as this only affects the precision of the analysis. In the presence of sharing, interfering cache blocks may already have been brought into the cache by the program under analysis, in which case they may not cause eviction of other cache blocks. Consider instruction a, which accesses cache block m mapped to cache set s, and let π be a cache hit path of the instruction. Let M^π be the set of cache blocks accessed in π. Let M_s be the set of cache blocks accessed by the interfering program and mapped to cache set s (hence, $B_s^{cb} = |M_c|$). Interfering accesses that access cache blocks in $M_s \cap M^\pi$ will never cause eviction of m, because these cache blocks will also be accessed by instructions on the hit path, and hence their impact on m would have already been considered (during private cache analysis). Hence, we calculate the set $M_s \setminus M^\pi$, and we ignore the hit path π of m if $|M_s \setminus M^\pi|$ is less than the eviction distance of m. If this happens for all hit paths of an access, then we can safely conclude that the access will never experience a miss due to interferences. Otherwise, the hit paths will be ignored while determining the OF (in Algorithm 1).

6. INTERACTION WITH THE SHARED BUS

In most multicore architectures, accesses to the shared cache have to go through the shared bus, which collects access requests from all cores and sends them to the shared cache. If requests from two different cores arrive at the same time, then one core must wait, as the shared cache can only fulfill one access request at a time. For predictability, it is desirable that the delay suffered due to this interference be bounded statically. One of the most commonly used arbitration policies to ensure bounded delays is the time division multiple access (TDMA)-based round-robin policy. In this policy, each core is assigned a fixed slot of time, and all access requests arriving during this slot will be immediately forwarded to the shared cache (provided those requests can be fulfilled in the

same slot). All slots are arranged in a fixed, static schedule that repeats itself. If a core generates an access request outside of its slot, then it must wait for its next slot. Since the size of the slots and the schedule are known, the delay can be accurately bounded.

An obvious bound on the maximum bus delay would be the maximum time between two slots assigned to the same core, obtained by assuming that the access request arrives just after the slot assigned to the core has finished. Using this upper bound for every shared cache access, however, could result in overapproximation, and hence previous works [Kelter et al. 2014; Chattopadhyay et al. 2010] have proposed a more precise timing analysis by using accurate bounds on the exact timing of each shared cache access.

The time at which a shared cache access happens depends on the hit-miss behavior of previous accesses, and hence approaches for TDMA-based shared bus analysis require the safe hit-miss classification for every individual access to the shared cache. In our approach, we do not provide a safe hit-miss classification for every access to the shared cache, but instead only provide upper bound on the number of shared cache misses. Providing guarantees for every shared cache access is very difficult, as that would require considering the worst-case interference arrival individually for every access, resulting in high overapproximation of the number of misses. By considering the global worst-case interference arrival, we can guarantee substantially higher number of cache hits. However, we cannot exactly pinpoint where the hits and misses are going to happen during the program execution.

In this section, we show that knowing the maximum number of shared cache misses caused due to interferences is enough to find the maximum shared bus delay that these misses will cause. Hence, our approach for shared cache analysis can be safely integrated with TDMA-based shared bus analysis techniques to accurately bound the shared bus delay. Note that we only concentrate on shared instruction cache accesses and assume separate busses for instruction and data traffic. Initially, the shared cache behavior of a program in isolation (which will provide precise hit-miss classification for individual accesses) would be used to find the WCET, taking into account the shared bus delays, e.g., using the techniques described in Kelter et al. [2014] and Chattopadhyay et al. [2010]. Then, we find the maximum number of shared cache misses caused due to interferences using approximate WCIP. We can show that every shared cache miss can only cause a maximum bus delay equal to the twice the TDMA period (which is the sum of the length of slots assigned to each core). Hence, the maximum bus delay caused due to interferences would also be directly proportional to the number of shared cache misses and can be found without pinpointing where the misses occur during execution.

Let π_{wc}^{Int} be the worst-case path in the presence of interferences. Let I be the maximum number of shared cache misses in the entire program caused by interferences (this number can determined using approximate WCIP). If I^{Int} is the maximum number of shared cache misses caused by interferences on the path π_{wc}^{Int}, then clearly $I^{Int} \leq I$. Let B^{Int} be the increase in the shared bus delay on the path π_{wc}^{Int} that happens due to the shared cache misses caused by interferences.

Let there be n_c cores, and let s_l be the slot length of each core in the TDMA schedule. For simplicity, we assume that the length of each slot is the same, and each core is assigned exactly one slot in the schedule. Hence, the TDMA period will be $n_c s_l$. We will show that $B^{Int} \leq 2n_c s_l I^{Int}$. Since $I^{Int} \leq I$, $2n_c s_l I$ would be a safe upper bound on the maximum increase in shared bus delay. Let $s_1, s_2, \ldots s_N$ be the sequence of shared cache accesses made during the execution of the worst-case path π_{wc}^{Int}. Moreover, let $s_{i_1}, s_{i_2}, \ldots s_{i_l}$ be the accesses in this sequence, which were initially shared cache hits but became misses due to interferences. Note that $l = I^{Int}$, the maximum number of misses on the worst-case path.

Up to s_{i_1}, there are no shared cache misses caused by interferences, and hence no extra bus delay will be caused. The access s_{i_1} is the first access to experience a miss due to interferences, which will result in an access to the main memory and therefore extra cache miss penalty c_p. Let β_{i_1} be the actual time at which the access s_{i_1} takes place, and let $o_{i_1} = \beta_{i_1} mod(n_c s_l)$ be the offset in the TDMA period.

In the worst case, this offset can occur just before the slot assigned to the core finishes, in such a way that the original cache hit could be served within the slot, but the cache miss cannot be served within the same slot. Formally, if $[s_l(p-1), s_l p)$ was the slot of the core issuing the request and γ_{i_1} was the original time required for the cache hit ($\gamma_{i_1} + c_p$ is the new time for the cache miss), then the worst case happens when $s_l p - o_{i_1} \geq \gamma_{i_1}$, but $s_l p - o_{i_1} < \gamma_{i_1} + c_p$. In this case, the core must wait for the next slot assigned to it, resulting in a bus delay of at most $n_c s_l$, which would not have been encountered in the run without interferences. Note that for all other offsets of the access s_{i_1}, no extra bus delay would happen due to the cache miss, and the maximum difference between the execution times would be only the cache miss penalty.

Because of the cache miss suffered by s_{i_1}, the time of the next shared bus access s_{i_1+1} will also change. Here, we use the offset relocation lemma proposed in Kelter et al. [2014], which states that if there are two executions of the same path, with one execution starting at offset o and another starting at a different offset o' in the TDMA period, then assuming identical behavior for every other microarchitectural component except the shared bus, the two executions will differ by at most $n_c s_l$ cycles. The reasoning is that the worst-case scenario would be where the new offset would occur just after the assigned slot, whereas the old offset may be at the beginning of the slot, resulting in the maximum delay of one TDMA period. In our case, the offset of s_{i_1+1} will change because of the miss to s_{i_1}, from the original offset when s_{i_1} was a hit. However, this will cause a maximum increase of $n_c s_l$ in the execution time of the path starting from s_{i_1+1}.

Hence, the cache miss to s_{i_1} can cause a maximum of increase of $2n_c s_l$ due to shared bus delays. Each miss s_{i_j} will cause a similar increase, and therefore the total increase in shared bus delay will be upper bounded by $2n_c s_l I^{Int}$.

7. EXPERIMENTAL EVALUATION

7.1. Setup

We use the WCET analyzer Chronos [Li et al. 2007] for our experiments. We experimented on 27 benchmarks from the Mälardalen WCET benchmark suite.[2] Since most of these benchmarks are fairly small, we also experimented on the benchmark *susan* from the MiBench suite[3] and the *autopilot* module from the PapaBench suite [Nemer et al. 2006]. We used *lp_solve* to solve the generated ILPs, and our experiments were performed on a four-core Intel i5 CPU with 4GB memory.

Unless stated otherwise, we assume a two-core architecture with a two-level cache hierarchy. Since we are focusing on the impact of shared instruction caches on the WCET, we assume a perfect data cache. We also assume a fixed cache miss penalty for every shared cache miss caused due to interferences. We ignore the effect of the shared bus to avoid relying on any specific arbitration policy and instead assume a fixed bus latency for every shared cache access.

For shared instruction cache analysis, we implemented three different techniques: (1) the approach of Hardy et al. [2009], which considers the effect of all interferences on all shared cache hits (this technique is also used for shared cache analysis in multicore

[2]WCET Projects/Benchmarks: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
[3]MiBench Version 1.0: http://wwweb.eecs.umich.edu/mibench/.

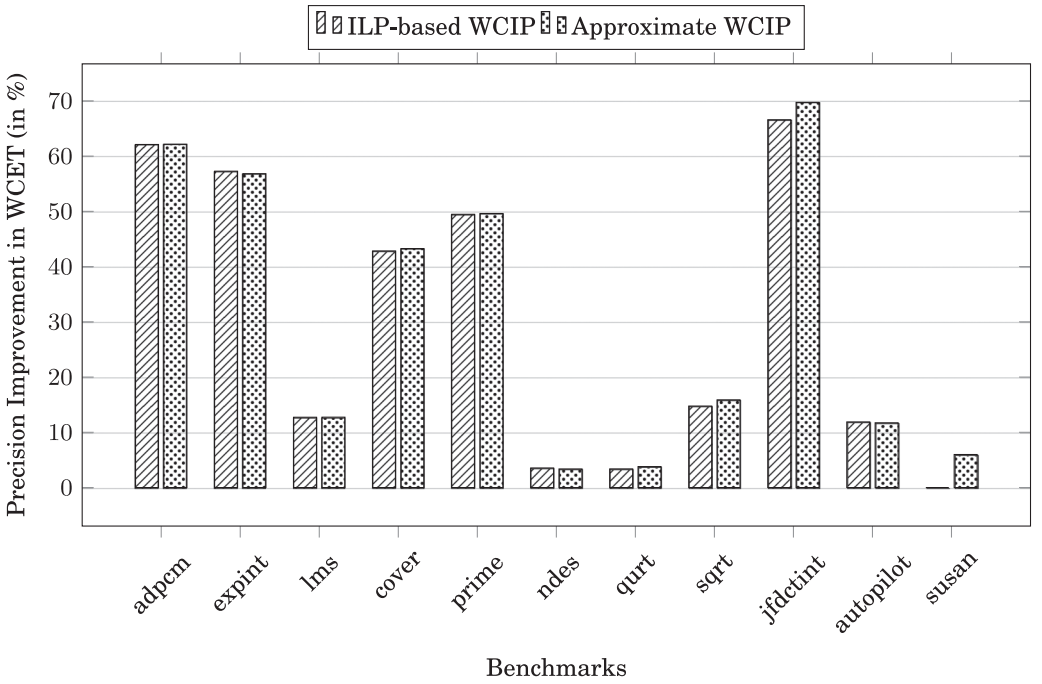

Fig. 6. Precision improvement of WCET obtained using ILP-based WCIP and approximate WCIP over the approach of Hardy et al. [2009].

Chronos [Chattopadhyay et al. 2012]) (2) the ILP-based approach for WCIP, as proposed in Nagar and Srikant [2014]; and (3) the approximate technique for WCIP, which is proposed in this article. We compare the precision of WCET obtained using all three techniques.

The code size of the Mälardalen benchmarks ranges from 0.2KB to 60KB, with an average size of 23.5KB. For these benchmarks, we assume a 1KB four-way L1 I-cache and a 4KB eight-way L2 I-cache, with block size of 32 bytes. However, both *susan* and *autopilot* are fairly large (130KB and 110KB code size, respectively) and show almost zero L2 cache hits for the preceding cache architecture, even without interferences. Hence, we use a 16KB eight-way L2 cache while experimenting on these benchmarks. We assume an L1 cache miss latency of 6 cycles and L2 cache miss latency of 30 cycles. To compute WCET of a benchmark on a two-core architecture, we assume that the benchmark runs on one core and the benchmark *nsichneu* runs on the other core. For all benchmarks except $jfdctint$, *nsichneu* is the worst-case adversary, i.e., the benchmark that causes the maximum shared cache interference.

7.2. Results

With the preceding assumptions, we calculated the WCET using all three techniques and found that WCIP gave lower WCETs for 9 of the 27 Mälardalen benchmarks, as well as both *susan* and *autopilot* as compared to Hardy et al.'s approach. ILP-based WCIP failed to provide the final WCET for *susan*. The WCETs were same for the rest of the benchmarks. Figure 6 shows the precision improvement of WCET (in percentages) obtained using the two WCIP approaches over the approach of Hardy et al. (The precision improvement is calculated as $\frac{WCET_H - WCET_O}{WCET_H}$, where $WCET_H$ is obtained using Hardy et al.'s approach, and $WCET_O$ is obtained using WCIP.)

We note that the Mälardalen benchmarks that did not show any precision improvement all had very few L2 cache hits even without interferences. For all benchmarks, the approach of Hardy et al. was not able to guarantee a single shared cache hit after considering the effect of interferences. The precision improvement in WCET using the ILP-based WCIP and approximate WCIP is equal for almost all benchmarks. In some benchmarks, the precision improvement using approximate WCIP is slightly higher due to the way in which the ILP-based approach counts misses due to interferences for cache hits inside loops. Specifically, multiple hit paths could be on the worst-case path, and hence the same miss could be counted multiple times. More explanation is present (along with an example) in Section 5E in Nagar and Srikant [2014]. The average precision improvement over 10 benchmarks for ILP-based WCIP was 27.5%, and for approximate WCIP (over 11 benchmarks) it was 26%. Note that over the same 10 benchmarks as ILP-based WCIP, the precision improvement of approximate WCIP was also 27.5%.

The complexity of the ILP increases with the number of shared cache hits in the program, because the ILP-based approach essentially searches among all distributions of interferences to shared cache hits to find the maximum WCET. On the other hand, the complexity of approximate WCIP is independent of the number of cache hits. This is demonstrated by the large benchmark *susan*, for which the ILP-based approach fails to provide the final WCET because the solver is unable to solve the ILP (with a timeout of 24 hours). On the other hand, approximate WCIP requires only 2.5 seconds to give the final WCET, with a precision improvement of 6% over Hardy et al.'s approach. The analysis times for the rest of the benchmarks were similar for both ILP-based and approximate WCIP, on average 0.82 seconds (with maximum of 4.5 seconds). The reason that ILP-based approach fails for *susan* is because the number of instructions causing L2 cache hits (without interferences) in *susan* were 325, whereas the maximum number of instructions causing L2 cache hits across all other benchmarks was 15 (for *autopilot*).

7.3. Hits on the WC Path and the Maximum OF

To understand why approximate WCIP performs so well, we measured the impact of the two assumptions made by approximate WCIP. The first assumption is that all shared cache hits of the program are present on the worst-case path. We measured the number of shared cache hits on the worst-case path (obtained assuming no interferences) and the total number of shared cache hits in the program. We found that among the 11 benchmarks, an average of 95.6% of the total number of shared cache hits were present on the WC path (with a minimum of 70% and a maximum of 100%). This shows that the first assumption will likely not have a major impact on the precision of approximate WCIP.

The average eviction distance (across all shared cache hits) across all benchmarks was 7.6, (ranging from 6.1 to 8). The second assumption made by approximate WCIP is to remove the overlapping effect by multiplying the original budget of interferences with the maximum OF. We found that the maximum OF across all cache sets was 1 for 9 out of the 11 benchmarks (it was 4 for *ndes* and 7 for *jfdctint*). This shows that removing the overlapping effect does not cause a huge increase in the number of interferences.

Another advantage of approximate WCIP is that we can express the increase in WCET due to interferences as a piecewise-linear function of the number of interferences. Figure 7 shows the increase in WCET (as compared to WCET obtained assuming no interferences) corresponding to different number of interferences (expressed as the ratio of the number of interferences to number of cache hits in the program). Although there would be a separate graph for each cache set, here we take the total number of interferences across all cache sets on the x-axis. The figure shows that the increase in

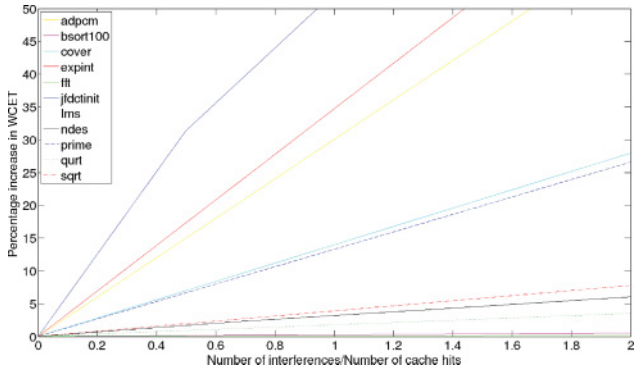

Fig. 7. Graph showing the relation between WCET obtained using WCIP and the amount of shared cache interference.

WCET is less than 50% for all benchmarks if the number of interferences do not exceed the number of cache hits. Moreover, for majority of the benchmarks, the increase in WCET is within 30% even when the number of interferences are twice the number of shared cache hits.

Changing the number of cores. We also experimented with a four-core architecture with the same cache configurations, except the L2 cache is now shared among all the four cores. We assume different instances of the same benchmark *nsichneu* running on three cores. For this architecture, approximate WCIP gave lower WCETs for nine benchmarks over the approach of Hardy et al., with average precision improvement of 25.5% (benchmarks *qurt* and *sqrt* did not show any improvement). A higher number of cores results in higher number of interferences to be distributed, causing more cache misses, but the increase in WCET computed using approximate WCIP continues to remain much smaller as compared to Hardy et al.'s approach.

7.4. Changing the Cache Size

We also experimented with two other L2-cache configurations, half the original size and double the original size. On halving the L2 cache, 7 (out of the original 11) benchmarks showed precision improvement. Again, both the ILP-based and approximate WCIP gave similar results, with the ILP-based approach again failing to provide the final WCET for *susan*. The average precision improvement over the 7 benchmarks was 33%. Note that the average precision improvement for the same 7 benchmarks for the original L2 cache was 37.6%. Hence, the precision improvement has decreased, which is as expected, since the smaller L2 cache will result in lower number of cache hits, thus decreasing the reliance of the WCET on L2 cache analysis.

On doubling the L2 cache, all benchmarks (except *sqrt* and *autopilot*) showed higher precision improvement for WCIP, and the average precision improvement for approximate WCIP over the 11 benchmarks was 35.7%. However, for ILP-based WCIP, the ILP solver was not able to solve the generated ILPs for three more benchmarks (*ndes*, *qurt*, and *jfdctint*), in addition to *susan*, within 24 hours. This has happened because of the higher number of L2 cache hits and the subsequent increase in the complexity of the ILP. On the other hand, approximate WCIP shows a precision improvement of 47% for *susan*, as the number of shared cache hits have almost doubled because of the larger shared cache. For all benchmarks, approximate WCIP took less than 5 seconds to find the WCET, with high precision improvement in the three benchmarks for which ILP-based WCIP fails. This illustrates the advantage of using approximate WCIP over ILP-based WCIP.

7.5. Comparison with Simulated WCET

We also compared the estimated WCETs obtained using the three techniques to the WCET obtained using simulation. To obtain the simulated WCET, we used the modified version of the Simplescalar framework, used for validation in multicore Chronos. The modified version supports simulation of shared cache and shared bus, among other architectural components. For our purposes, we only simulated the effect of the shared cache and assume constant shared bus delay.

We calculated the WCET estimation ratio, defined as $\frac{Estimated\ WCET}{Simulated\ WCET}$, for 11 benchmarks. We first found the simulated and estimated WCETs assuming a private L2 cache to determine the impact of infeasible paths, private cache analysis, and so forth, on the overestimation of the estimated WCET. Then, we assumed the same L2 cache shared between two cores, with the benchmark *nsichneu* running on the other core, and found the simulated and estimated WCETs, as well as the overapproximation ratio. The estimated WCETs were determined using the three shared cache analysis techniques.

We found that for majority of the benchmarks, the overestimation ratio in the case where the shared L2 cache is analyzed using WCIP is almost the same as the overestimation ratio for the private L2 cache. The average overestimation ratio for the private L2 cache was 1.82, whereas for the shared L2 cache analysis using ILP and approximate WCIP it was 2.1. On the other hand, Hardy et al.'s analysis introduces large amounts of imprecision, and the average overestimation ratio was 3.4. This shows that shared cache analysis using WCIP does not introduce large amounts of imprecision in the estimated WCETs. For detailed results, we refer the reader to our technical report [Nagar and Srikant 2015b].

7.6. Cache Partitioning

Cache partitioning is a hardware-based approach to simplify shared cache analysis in multicore architectures. In cache partitioning, each core is assigned a private portion of the shared cache, which will not be accessed by any other core. This ensures that there will be no interferences to account for during shared cache analysis, and hence private cache analysis techniques can be directly applied for the shared cache. The disadvantage is that a core will not be able to use the entire shared cache, and therefore it may suffer more shared cache misses (both capacity and conflict misses).

Here, we limit our attention to fixed partitioning and compare the WCETs obtained by using partitioned shared cache to the WCETs obtained using WCIP in an un-partitioned shared cache. There are two ways in which fixed partitioning can be implemented: (1) vertical partitioning, where each core is assigned a subset of ways in all cache sets, and (2) horizontal partitioning, where each core is assigned a subset of cache sets. For our two-core architecture, we divided the cache equally between both the cores.

For vertical partitioning experiments, we decreased the shared cache associativity from 8 to 4, whereas for horizontal partitioning experiments, we decreased the number of cache sets from 16 to 8. Figure 8 shows the percentage increase in the WCET obtained using approximate WCIP horizontal cache partitioning and assuming that all shared cache accesses as misses, as compared to WCET of the benchmarks running in isolation with an unpartitioned shared L2 cache.

The percentage increase in WCET is calculated as $\frac{WCET_{shared} - WCET_{orig}}{WCET_{orig}}$, where $WCET_{orig}$ is the WCET of the benchmark running on single-core architecture with the same (unpartitioned) cache hierarchy, whereas $WCET_{shared}$ is obtained using either approximate WCIP or the two cache partitioning techniques or assuming that all shared cache

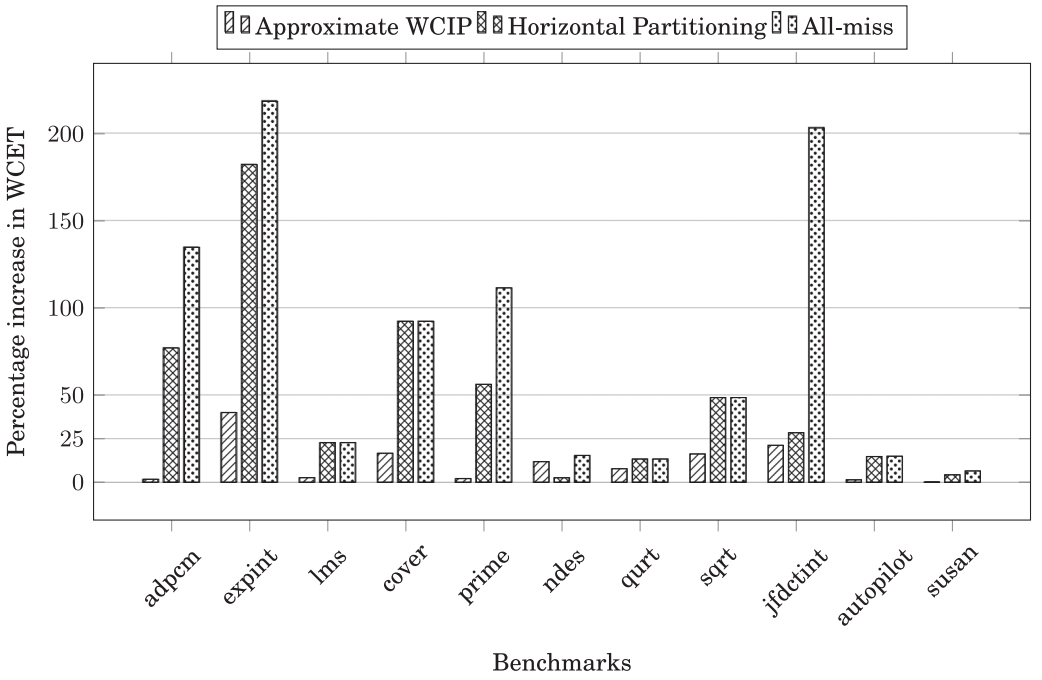

Fig. 8. Graph showing the percentage increase in WCET obtained using approximate WCIP and horizontal cache partitioning, and assuming all shared cache accesses as misses.

accesses miss the cache. The point of comparing with $WCET_{orig}$ is that the lower the percentage increase, the lower the imprecision introduced by shared cache analysis to the WCET.

First, note that using vertical cache partitioning does not result in any increase in the WCET for all benchmarks except *jfdctint*. The reason is that most of these benchmarks do not access more than four cache blocks per cache set in the shared cache, and hence their cache performance is not affected by decreasing the cache associativity to 4. *jfdctint* accesses at least six cache blocks in every cache set, and the percentage increase in its WCET due to vertical partitioning was 28.4%, which is greater than approximate WCIP (21.1%).

Horizontal partitioning, on the other hand, is highly ineffective, as it introduces greater imprecision than both of the other techniques for all benchmarks except *ndes*. The average increase in the WCET across the 11 benchmarks for approximate WCIP is 11%, whereas for horizontal partitioning it is 49.3%. In addition, note the high increase in WCET for most benchmarks when it is assumed that all shared cache accesses miss the cache. The average increase in WCET in this case is 80.1%, which highlights the importance of shared cache analysis in accurate WCET estimation.

From the preceding results, it would seem that using normal cache analysis with vertical partitioning is more effective than using WCIP with an unpartitioned shared cache. However, note that with WCIP, the WCET also depends on amount of interference caused by programs running on other cores, and for the preceding experiment, we used the worst-case adversary. By selecting programs that generate less shared cache interference, the WCET can be controlled using WCIP, but with cache partitioning, there will be no impact on the WCET.

7.7. Interaction with TDMA-Based Shared Bus

As shown in Section 6, our technique can also be integrated with precise shared bus analysis techniques, which provide better bounds on the shared bus delay. To find the impact on precision of the WCET, we experimented with a shared bus architecture that uses a TDMA-based round-robin arbitration policy. We assume the slot length of each core to be 50 cycles in the TDMA schedule. We use the global convergence analysis proposed in Kelter et al. [2014] to accurately bound the shared bus delay for each basic block. This analysis essentially finds all offsets in the TDMA period that can occur at the start of a basic block, which are then used to bound the bus delay experienced by the shared cache accesses present in the basic block.

We used Hardy et al.'s approach for shared cache analysis, in conjunction with the global convergence analysis for shared bus, to find $WCET_H$. We used the shared cache behavior in isolation, in conjunction with the global convergence analysis to find $WCET_I$. Then, we used approximate WCIP to find the maximum increase in WCET due to interferences, added this increase to $WCET_I$, along with $200(= 2n_c s_l)$ cycles for each shared cache miss, to find $WCET_O$. As shown in Section 6, this is a safe overapproximation of the extra bus delay caused by the shared cache misses. We found that approximate WCIP continued to provide lower WCET estimates for all 11 benchmarks, with average precision improvement of $WCET_O$ over $WCET_H$ being 21.4%.

8. CONCLUSION AND FUTURE WORK

Shared cache analysis plays a crucial role in obtaining precise WCET estimates on multicores, and WCIP is one of the most precise techniques used to perform shared cache analysis. In this work, we show that performing WCIP is NP-hard and propose an approximate greedy technique for WCIP that bypasses the hard problem of finding the worst-case path in the presence of interferences. Whereas ILP-based WCIP is NP-hard, approximate WCIP has a time complexity linear in the size of the shared cache and the number of cache hits, thus guaranteeing fast analysis time. Experimentally, we show that approximate WCIP is as precise as ILP-based WCIP across all benchmarks, with a substantial reduction in analysis time.

Precise and fast shared cache analysis opens up several interesting avenues for future work. For example, the increase in WCET due to shared cache interferences is a piecewise linear function of the number of interferences (after removing the overlapping effect), and this gives a good handle on controlling the WCET while maximizing utilization during scheduling. Moreover, the WCET of a program, in the presence of shared caches, can be determined independently of the programs running on other cores by assuming upper bounds on the amount of interference.

REFERENCES

Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. 2011. Precise and efficient parametric path analysis. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*.

Sebastian Altmeyer, Claire Maiza, and Jan Reineke. 2010. Resilience analysis: Tightening the CRPD bound for set-associative caches. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 153–162.

Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudhury, Timon Kelter, Marwedel Peter, and Falk Heiko. 2012. A unified WCET analysis framework for multi-core platforms. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*.

Sudipta Chattopadhyay and Abhik Roychoudhury. 2011. Scalable and precise refinement of cache timing analysis via model checking. In *Proceedings of the Real-Time Systems Symposium*.

Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. 2010. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*.

Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems* 17, 2–3, 131–181.

Damien Hardy, Thomas Piquet, and Isabelle Puaut. 2009. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the Real-Time Systems Symposium*.

Damien Hardy and Isabelle Puaut. 2008. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the Real-Time Systems Symposium*.

Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems* 50, 2, 185–229. DOI:http://dx.doi.org/10.1007/s11241-013-9189-x

Benjamin Lesage, Damien Hardy, and Isabelle Puaut. 2010. Shared data cache conflicts reduction for WCET computation in multi-core architectures. In *Proceedings of the International Conference on Real-Time Networks and Systems*.

Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudury. 2007. Chronos: A timing analyzer for embedded software. *Science of Computer Programming* 69, 1–3, 56–67. http://www.comp.nus.edu.sg/∼rpembed/chronos.

Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2009. Timing analysis of concurrent programs running on shared cache multi-cores. In *Proceedings of the Real-Time Systems Symposium*.

Thomas Lundqvist and Per Stenström. 1999. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*.

Silvano Martello and Paolo Toth. 1990. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons.

K. Nagar and Y. N. Srikant. 2015a. Path sensitive cache analysis using cache miss paths. In *Verification, Model Checking, and Abstract Interpretation*. Springer, Berlin, 43–60.

K. Nagar and Y. N. Srikant. 2014. Precise shared cache analysis using optimal interference placement. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*.

K. Nagar and Y. N. Srikant. 2015b. *Shared Instruction Cache Analysis in Real-Time Multi-Core Systems*. Technical Report. http://www.csa.iisc.ernet.in/TR/2015/1/tech-report.pdf.

Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. 2006. PapaBench: A free real-time benchmark. In *Proceedings of the Workshop on Worst-Case Execution Time Analysis (WCET'06)*.

Marco Paolieri, Eduardo Quiñones, Franciso J. Cazorla, Guillem Bernat, and Mateo Valero. 2009. Hardware support for WCET analysis of hard real-time multicore systems. In *Proceedings of the International Symposium on Computer Architecture*.

Harini Ramaprasad and Frank Mueller. 2005. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*.

Vivy Suhendra and Tulika Mitra. 2008. Exploring locking and partitioning for predictable shared caches on multi-cores. In *Proceedings of the Design Automation Conference*.

Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. 2013. Making shared caches more predictable on multicore platforms. In *Proceedings of the Euromicro Conference on Real-Time Systems*.

Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguire, Daniel Grund, Jrg Herter, Jan Reineke, Bjrn Wachter, and Stephan Wilhelm. 2010. Static timing analysis for hard real-time systems. In *Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, Vol. 5944. Springer, 3–22.

Jun Yan and Wei Zhang. 2008. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*.

Jun Yan and Wei Zhang. 2009. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications*.

Received March 2015; revised October 2015; accepted November 2015