Shared Instruction Cache Analysis in Real-time Multi-core Systems

Kartik Nagar, Y. N. Srikant
IISc-CSA-TR-2015-1
January 2015

Abstract

Real-time systems require a safe and precise estimate of the Worst Case Execution Time (WCET) of programs. In multi-core architectures, the precision of a program's WCET estimate is highly dependent 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, to determine the WCET. In this work, we show that performing WCIP is an NP-Hard problem, by reducing the 0-1 Knapsack problem. We then propose an approximate technique to perform WCIP, which is fast and experimentally as precise as previously published ILP-based approach to WCIP. Shared cache analysis performed using WCIP is very precise as compared to other approaches, and we report an average precision improvement of 28.79 % in the WCETs obtained using approximate WCIP, over other approaches, with almost 100x reduction in average analysis time over ILP-based WCIP.

1 Introduction

Multi-cores are widespread in today's computing devices, from hand-held mobiles to servers and workstations. Using multi-cores for real time systems has proved difficult, because real time systems require an estimate of the maximum execution time of programs (also called the Worst Case Execution

Time), and obtaining precise estimates of WCET on multi-core architectures is not easy.

While predicting the exact WCET is undecidable in general, we would like to get as close as possible, without underestimating the actual value. After decades of research, a standard framework for estimating WCET of programs has emerged (see [1] for a comprehensive survey). 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 micro-architectural analysis is carried out to determine the impact of architectural components such as caches, pipelines, etc. 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. Multi-core architectures typically have a cache hierarchy, with private caches assigned to each core at lower levels, and a shared cache, which is shared between all the cores at the highest level. Accesses which miss the shared cache have to go to the main memory, and due to the huge 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 bringing down 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 have a destructive impact on the cache behavior of the program under analysis. In principle, these interferences can arrive at any time and 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 multi-core setting 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. While 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, it is straightforward to find the number of interferences (i.e. shared cache accesses) generated by each core. However, an interference can cause any number of shared cache misses between 0 and the cache associativity, depending on the timing of its arrival. Hence, the same number of interferences can cause different number of shared 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 hence 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 [2].

Theoretically, WCIP 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, which typically consider the effect of all interferences on every cache access [3].

In [2], WCIP is performed by generating an Integer Linear Program (ILP), whose optimal solution encodes the worst-case program path and the worst-case interference placement on this path. Solving an ILP is a NP-Hard problem, and this is reflected in the high analysis time required to solve the ILP for large programs. This raises the challenge of finding efficient techniques to perform WCIP. In this work, we show that performing WCIP is a NP-Hard problem, and hence there cannot exist an efficient algorithm. 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. This result shows that shared cache analysis is actually a combinatorial optimization problem, and is very different from normal private cache analysis.

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 the interferences, while there may be program paths with large number of shared cache hits but lower execution times. To bypass this problem, we propose an approximate technique for WCIP which assumes that all the shared cache hits are present on the worst-case path (calculated assuming

[1]Note that because of infeasible paths, imprecision of private cache analysis, etc., 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

no interferences). We then find an upper bound on the increase in execution time due to interferences across all program paths. We use an abstract interpretation based approach followed by a greedy algorithm to find this upper bound, whose time complexity depends only on the size of the shared cache.

We have implemented our technique in the Chronos WCET analyzer, and tested it on Mälardalen benchmarks. 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 [3], with an average precision improvement of 28.79% 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, while the approximate technique requires a maximum of 0.01 seconds across all benchmarks. We also compare the WCETs obtained using fixed cache partitioning techniques with the WCETs obtained using WCIP. Following are the contributions made by the paper:

- We formally prove that performing worst case interference placement in a program is an NP-hard problem, by showing a reduction from the 0-1 Knapsack problem (in Section 3).
- We propose an approximate technique to perform worst case interference placement, which is provably fast and is experimentally as precise as the ILP based approach (in Section 4).
- We provide experimental results comparing the WCETs obtained by ILP-based and approximate WCIP, with the WCETs obtained using previous approaches to shared cache analysis. We also compare the effectiveness of analysis-based WCIP with hardware-based approaches such as fixed cache partitioning (in Section 6).

2 Shared Cache analysis

In this section, we provide a quick overview of cache terminilogy, and review some of the existing works in shared cache analysis. Caches store a small subset of main memory closer to the processor, and provide fast access to its contents. All data 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 maximum of A cache blocks. Given an

access to a cache block, the cache subsystem first finds the unique cache set containing the accessed cache block, searches for it among the (at most) A cache blocks in the cache set, and if it is not present, brings it from the main memory (or higher level caches). A multi-level cache hierarchy has independent caches at different levels, generally with smaller caches 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 which 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. We assume a standard multi-core architecture, where each core has one (or more) private caches at lower levels and a shared cache (shared between all cores) at the highest level.

We now summarize existing techniques for cache analysis. Must Analysis [4] for private caches is an Abstract Interpretation based technique, which finds those cache blocks which are guaranteed to be in the actual 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 ([3], [5], [6], [7]) build on top of Must analysis, and find shared cache accesses which 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 states at each program point are modified by considering the effect of *all* interferences generated by 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.

Hardware approaches ([8], [9], [10], [11]) focus on making the multi-core architecture prediction-friendly by using techniques such as cache locking, cache partitioning, etc. 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 wastage of resources and require support from the hardware. Further, the schedulability analysis becomes complicated, and the constraints on task period and execution time imposed by the schedulability test become more stringent [12], which may prevent task sets from being scheduled.

There have also been efforts in shared cache interference-aware task map-

ping [13], which tries to find the optimal assignment of tasks to cores which minimizes shared cache interference. However, their technique is actually unsafe, because they assume that the worst-case path in the program does not change due to interferences, and only use the cache hits on the WC path to find the increase in execution time due to interferences. As we show in our work, other paths in the program (e.g. paths with more shared cache hits) could exhibit higher execution time in the presence of interferences. Finally, [14] contains a detailed discussion on issues related to timing analysis, caused due to interference to shared resources.

In [2], we proposed the WCIP approach for shared cache analysis (referred in [2] 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 in spite of the interferences from other cores. Ultimately, this results in much lower WCET estimates for multi-core architectures, which are nonetheless safe. For details on the ILP based approach for WCIP, we refer to [2].

In this work, we make the following assumptions. First, we assume that the maximum number of interferences that can be generated by all other cores, during a single execution instance of the program under analysis is known. This only requires the knowledge of the assignment of programs to cores, since the shared cache accesses made by a program 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. The cache replacement policy is assumed to be LRU and we assume a timing anomaly-free architecture. We also assume single-threaded and independent programs running on all the cores. We perform AI-based cache analysis at the private cache level to obtain an upper bound on the number of shared cache accesses, and we also perform the AI-based analysis at the shared cache level to obtain a lower bound on the number of shared cache misses.

3 Complexity of WCIP

Given a program and the set of interferences (coming from other cores) to each cache set, the WCIP problem is to (1) find the program path with

the maximum execution time in the presence of interferences and (2) to find a distribution of interferences, on this program path, which causes the maximum number of shared cache misses on the path. A distribution of interferences will assign disjoint subsets of interferences at each program point in the path. Note that in the absence of code/data sharing, cores will never access each other's cache blocks, and hence we would only be interested in the number of interferences assigned at each program point, since the cache blocks accessed by interferences will not matter.

The two sub-problems of finding the worst-case path and the worst-case distribution of interferences are inter-dependent on each other. To find the worst-case path in the presence of interferences, we must know the worst-case distribution which 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 naïve 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 interferences is not a difficult problem. The cache behavior can be estimated without interferences using the AI-based techniques, which would lead to a precise estimate of the WCET of each basic block in the program. Given the WCET of each basic block, and the program CFG, finding the WCET becomes equivalent to finding the longest path in the CFG, treating each basic block as a vertex and its WCET as its label. This can be accomplished in time polynomial in the number of basic blocks (for example, see [15]). 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 direct-mapped shared cache with a single cache set. Direct-mapped 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 our 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 direct-mapped 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 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. a_{i1}(respectively a_{i2}) is the execution time, without interferences, of the left (respectively right) branch of the ith segment. This execution time is obtained by performing normal cache analysis at all levels, assuming no interferences. b_{i1}(respectively b_{i2}) is the number of shared cache hits of the left (respectively right) branch of the ith segment. These are the accesses which are guaranteed to hit the shared cache, without any interferences. Assume WLOG that $\forall i, a_{i1} \geq a_{i2}$. Hence, the WCET without interferences would be $\sum_{i=1}^{n} a_{i1}$, obtained by taking the left branch of each segment (since there are no interferences, no additional cache misses will occur).

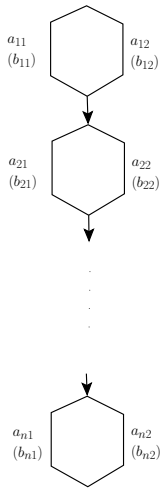


Figure 1:

Now suppose that B interferences come from other cores, causing some of the shared cache hits in the program to become misses. For simplicity, assume a shared cache miss penalty of 1 cycle. Since B interferences can cause at most B cache misses, if there are at least B cache hits among the left branches, then the WCET with interferences would be $\sum_{i=1}^{n} a_{i1} + B$.

However, if that is not the case, i.e. if $\sum_{i=1}^{n} b_{i1} < B$, then the maximum execution time with interferences by picking the left-hand branch in each segment would be $\sum_{i=1}^{n}(a_{i1} + b_{i1})$. We may be able to increase this execution time by taking the right-hand branch in some segment to use the extra unused interferences, if it has more shared cache hits than the left-hand branch. If

for some i, $b_{i2} > b_{i1}$, then by taking the right-hand branch, we would able to increase the execution time by $b_{i2} - b_{i1} - (a_{i1} - a_{i2})$, by making use of $b_{i2} - b_{i1}$ extra interferences.

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 Knapsack Problem (KP), in which there are n objects each with profit 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 profit. 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 'profit' of $b_{i2} - b_{i1} - (a_{i1} - a_{i2})$, with an associated 'weight' of $b_{i2} - b_{i1}$. The total weight budget would be the extra interferences not used on the left-hand branches, i.e. $B - \sum_{i=1}^{n} b_{i1}$.

The only issue is that in WCIP, it is not necessary that all the cache hits in a selected branch may be converted to cache misses due to interferences. In other words, $b_{i2} - b_{i1} - (a_{i1} - a_{i2})$ is only the maximum profit available by selecting the right-hand branch in segment i, and the worst-case distribution can choose a lower profit, 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 profit by v_i. To solve this dilemma, we will use the fact that there always exists a worst-case distribution which will try to convert all the 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 P with n segments (shown in Figure 1), an interference budget B and a target execution time T, does there exist a path selection function $S : \{1, \ldots, n\} \to \{1, 2\}$, and assigned interferences $b_1, \ldots, b_n$ such that

$$\forall i, 1 \leq i \leq n, \quad b_i \leq b_{iS(i)} \tag{1}$$

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

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

The path selection function will choose one of the two branches in each

segment, and thus reflects the worst-case path, while the number of interferences distributed in each segment will decide the number of shared cache misses caused in that segment. An interference distribution which obeys equations (1) and (2) is called a valid distribution. The following lemma states that given any path selection function and a valid interference distribution, there exists another selection function and 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 P with n segments $\{<(a_{i1}, b_{i1}), (a_{i2}, b_{i2})> | i = 1, \ldots, n\}$ $(\forall i, a_{i1} \geq a_{i2})$, an interference budget B, assume that $\sum_{i=1}^{n} b_{i1} \leq B$. Given a path selection function S and assigned interferences $b_1, \ldots, b_n$ which form a valid distribution, there exists another path selection function $\hat{S}$, and valid distribution $\hat{b}_1, \ldots, \hat{b}_n$, with the following properties:*

1. *If $\hat{S}(i) = 1$, then $\hat{b}_i = b_{i1}$.*
2. *There exists at most one j such that $\hat{S}(j) = 2$ and $\hat{b}_j < b_{j2}$.*
3. *$\sum_{i=1}^{n} a_{iS(i)} + b_i \leq \sum_{i=1}^{n} a_{i\hat{S}(i)} + \hat{b}_i$*

Lemma 3 *Given a program P with n segments $\{<(a_{i1}, b_{i1}), (a_{i2}, b_{i2})> | i = 1, \ldots, n\}$ $(\forall i, a_{i1} \geq a_{i2})$, an interference budget B, assume that $\sum_{i=1}^{n} b_{i1} \leq B$. Given a path selection function S and assigned interferences $b_1, \ldots, b_n$ which form a valid distribution, there exists another path selection function $\hat{S}$, and valid distribution $\hat{b}_1, \ldots, \hat{b}_n$, with the following properties:*

1. *If $\hat{S}(i) = 1$, then $\hat{b}_i = b_{i1}$.*
2. *There exists at most one j such that $\hat{S}(j) = 2$ and $\hat{b}_j < b_{j2}$.*
3. *$\sum_{i=1}^{n} a_{iS(i)} + b_i \leq \sum_{i=1}^{n} a_{i\hat{S}(i)} + \hat{b}_i$*

Proof:

We will perform a series of redistribution of interferences from the initial distribution to ensure that properties (1) and (2) are met. After each redistribution, we will also ensure that the total execution time either remains the same or it increases (i.e. property (3) remains true).

First, suppose $\exists i$, $S(i) = 1$ and $b_i < b_{i1}$. Let $b_{rem} = b_{i1} - b_i$.

If $\forall j, S(j) = 1$, then we can take $\hat{S}(j) = 1, \hat{b}_j = b_{j1}$ for all j. This satisfies property (1) and (2). Also, $\hat{b}_i = b_{i1} > b_i$, hence the total execution time has increased. Hence, $\hat{S}$ and $\hat{b}_i$ satisfy all the properties of the lemma and we are done.

Suppose $\exists j$, $S(j) = 2$. Then $b_j - b_{j1} > a_{j1} - a_{j2}$.

Let $b_j^{'} = b_j - b_{rem}$.

Case - 1: If $b_j^{'} > a_{j1} - a_{j2}$, consider the new distribution $\hat{S}(k) = S(k), \forall k$, $\hat{b}_i = b_{i1}$, $\hat{b}_j = b_j^{'}$, $\hat{b}_k = b_k$ for all other k.

$$\begin{aligned}\sum_{k=1}^{n} \hat{b}_k &= b_{i1} + b_j - b_{rem} + \sum_{k \neq i,j} b_k \\ &= b_i + b_j + \sum_{k \neq i,j} b_k \\ &= \sum_{k=1}^{n} b_k \leq B\end{aligned}$$

$$\begin{aligned}&\sum_{k=1}^{n} (a_{k\hat{S}(k)} + \hat{b}_k) - \sum_{k=1}^{n} (a_{kS(k)} + b_k) \\ &= a_{i1} + b_{i1} + a_{j2} + b_j - b_{rem} - a_{i1} - b_i - a_{j2} - b_j \\ &= 0\end{aligned}$$

In this case, we have redistributed the interferences such that $\hat{S}(i) = 1$ and $\hat{b}_i = b_{i1}$, while ensuring the total execution time remains the same.

Case - 2: If $b_j^{'} \leq a_{j1} - a_{j2}$, consider $\hat{S}(j) = 1$, $\hat{S}(k) = S(k)$ for all other k, $\hat{b}_j = b_{j1}$, $\hat{b}_i = b_i + (b_j - b_{j1})$, $\hat{b}_k = b_k$ for all other k. Again,

$$\begin{aligned}\sum_{k=1}^{n} \hat{b}_k &= b_i + b_j - b_{j1} + b_{j1} + \sum_{k \neq i,j} b_k \\ &= \sum_{k=1}^{n} b_k\end{aligned}$$

$$\begin{aligned}&\sum_{k=1}^{n} (a_{k\hat{S}(k)} + \hat{b}_k) - \sum_{k=1}^{n} (a_{kS(k)} + b_k) \\ &= a_{i1} + b_i + b_j - b_{j1} + a_{j1} + b_{j1} - a_{i1} - b_i - a_{j2} - b_j \\ &= a_{j1} - a_{j2} \geq 0\end{aligned}$$

In this case, by redistributing the interferences, we have $\hat{S}_j = 1, \hat{b}_j = b_{j1}$, while $\hat{b}_i > b_i$. By repeating the process with other j' such that $S(j') = 2$, we can continue to increase $\hat{b}_i$ until it reaches b_{i1}. If there is no such j', then we can simply set $\hat{b}_i$ to b_{i1}, since $\sum_{k=1}^{n} b_{k1} \leq B$.

Thus, we have shown that we can always redistribute the interferences used on the right branch of some segment to the left branch of another segment. It is optimal to use as many interferences as possible on the left branches.

After performing the above transformations, we can now assume property 1. We will now give the redistribution strategies for proving property 2.

Suppose $\exists i, j$ such that $S(i) = 2, b_i < b_{i2}, S(j) = 2, b_j < b_{j2}$. Then $b_i - b_{i1} > a_{i1} - a_{i2}$ and $b_j - b_{j1} > a_{j1} - a_{j2}$. Let $b_i^{rem} = b_{i2} - b_i$ and $b_j^{rem} = b_{j2} - b_j$.

Case - 1 : If $b_i^{rem} \geq b_j - b_{j1}$, then let $\hat{S}(j) = 1$, $\hat{b}_j = b_{j1}$, $\hat{S}(i) = 2$, $\hat{b}_i = b_i + b_j - b_{j1} \leq b_i + b_i^{rem} = b_{i2}$, for all other k, $\hat{S}(k) = S(k)$ and $\hat{b}_k = b_k$.

$$\begin{aligned}\sum_{k=1}^{n} \hat{b}_k &= b_i + b_j - b_{j1} + b_{j1} + \sum_{k \neq i,j} b_k \\ &= \sum_{k=1}^{n} b_k \leq B\end{aligned}$$

$$\begin{aligned}&\sum_{k=1}^{n} (a_{k\hat{S}(k)} + \hat{b}_k) - \sum_{k=1}^{n} (a_{kS(k)} + b_k) \\ &= a_{i2} + b_i + b_j - b_{j1} + a_{j1} + b_{j1} - a_{i2} - b_i - a_{j2} - b_j \\ &= a_{j1} - a_{j2} \geq 0\end{aligned}$$

In this case, by redistribution, we have only one segment (i) where $\hat{S}(i) = 2$ and $\hat{b}_i < b_{i2}$. Similarly, if $b_j^{rem} \geq b_i - b_{i1}$, a similar proof will work.

Case - 2 : Now, suppose that neither of the two conditions are true. In particular, $b_i^{rem} < b_j - b_{j1}$. There are two subcases to consider:

Subcase - 1 : $b_i^{rem} \leq b_j - b_{j1} - (a_{j1} - a_{j2})$. Hence, $(b_j - b_i^{rem}) - b_{j1} > a_{j1} - a_{j2}$. Let $\hat{S}(i) = 2$, $\hat{b}_i = b_{i2}$, $\hat{S}(j) = 2$, $\hat{b}_j = b_j - b_i^{rem}$, and for all other k, $\hat{S}(k) = S(k)$, $\hat{b}_k = b_k$.

$$\begin{aligned}\sum_{k=1}^{n} \hat{b}_k &= b_{i2} + b_j - b_i^{rem} + \sum_{k \neq i,j} b_k \\ &= \sum_{k=1}^{n} b_k \leq B\end{aligned}$$

$$\sum_{k=1}^{n}(a_{k\hat{S}(k)} + \hat{b}_k) - \sum_{k=1}^{n}(a_{kS(k)} + b_k)$$
$$= a_{i2} + b_{i2} + a_{j2} + b_j - b_i^{rem} - a_{i2} - b_i - a_{j2} - b_j$$
$$= 0$$

Hence, by redistribution, we now have only one segment (j) where all cache hits are not converted to misses on the right branch.

Subcase - 2 : $b_i^{rem} > b_j - b_{j1} - (a_{j1} - a_{j2})$. Let $\hat{S}(i) = 2$, $\hat{b}_i = b_{i2}$, $\hat{S}(j) = 1$, $\hat{b}_j = b_{j1}$, and for all other k, $\hat{S}(k) = S(k)$, $\hat{b}_k = b_k$.

$$\sum_{k=1}^{n} b_k - \sum_{k=1}^{n} \hat{b}_k = b_i + b_j - b_{i2} - b_{j1}$$
$$= b_j - b_{j1} - b_i^{rem} > 0 \text{ (by assumption)}$$

Hence, $\sum_{k=1}^{n} \hat{b}_k < \sum_{k=1}^{n} b_k \leq B$.

$$\sum_{k=1}^{n}(a_{k\hat{S}(k)} + \hat{b}_k) - \sum_{k=1}^{n}(a_{kS(k)} + b_k)$$
$$= a_{i2} + b_{i2} + a_{j1} + b_{j1} - a_{i2} - b_i - a_{j2} - b_j$$
$$= b_i^{rem} - (b_j - b_{j1} - (a_{j1} - a_{j2})) > 0$$

Again, we have ensured that all cache hits are converted to misses in segments i and j. Thus, we have showed that by redistributing the interferences, we can always ensure that whenever the left branch is taken in a segment, all the cache hits are converted to misses, and there is at most one segment where the right branch is taken, but all cache hits are not converted to misses ■.

Property 1 guarantees that all cache hits in left-hand branches which are selected by $\hat{S}$ 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. We now define the decision version of the 0-1 Knapsack problem, which is known to be NP-Hard [16].

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, does there

exist 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 knapsack problem, we first convert it to another knapsack problem where the weights are greater than the values for all items. Let v_m be the maximum value among all the 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 $W' = v_m W$, while 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 knapsack problem will also be a solution of the modified problem, and vice versa.

We now construct a simple-branched program $\mathcal{P}$ (shown in Figure 2), along with the interference budget and target execution time, in such a way the solution to the WCIP problem in this program corresponds to the solution of the modified knapsack problem. $\mathcal{P}$ has n segments, with $a_{i1} = w_i^{'} - v_i$, $b_{i1} = 0$, and $a_{i2} = 0$, $b_{i2} = w_i^{'}$, 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 a_{i2} can also be taken as any constant C, in which case a_{i1} should be taken as $w_i^{'} - v_i + C$. All we want is that the 'profit' of taking the right-side branch in segment i should be v_i.

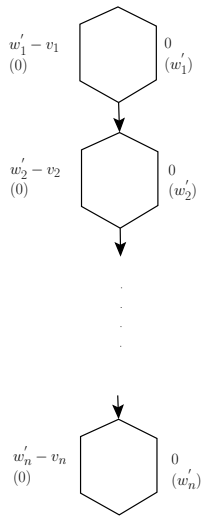


Figure 2: Program $\mathcal{P}$

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:

Theorem 1 *$\exists P \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in P} w_i^{'} \leq W'$ and $\sum_{i \in P} v_i \geq V \Leftrightarrow$ There exists a path selection function S and a valid interference distribution $b_1, \ldots, b_n$, such that $\mathcal{P}$ achieves the target execution time T.*

Proof:

$(\Rightarrow)$

We are given a solution to the Knapsack problem, 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 $S : \{1, \ldots, n\} \rightarrow \{1, 2\}$ and b_i as follows:

$$S(i) = \begin{cases} 2 & \text{if } i \in P \\ 1 & \text{otherwise} \end{cases}$$

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

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.

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

Thus, $\mathcal{P}$ achieves the target execution time under S and $b_1, \ldots, b_n$. We now prove the theorem in the other direction.

$$(\Leftarrow)$$

We are given S, $b_1, \ldots, b_n$ such that the target execution time is achieved and the interference distribution is valid (i.e. it satisfies equations (1),(2),(3)).

Note that if $S(i) = 1$, then $b_i = 0$, and by Lemma 2, we can assume there is at most one j such that $S(j) = 2$ 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 $P = \{i|S(i) = 2\}$. First, we show that this selection of items meets the target value V.

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

$$\Rightarrow \sum_{i:S(i)=1} (w_i^{'} - v_i) + \sum_{i:S(i)=2} b_i \geq \sum_{i=1}^{n} (w_i^{'} - v_i) + V$$

$$\Rightarrow \sum_{i:S(i)=2} b_i \geq \sum_{i:S(i)=2} (w_i^{'} - v_i) + V$$

$$\Rightarrow \sum_{i:S(i)=2} v_i \geq V + \sum_{i:S(i)=2} (w_i^{'} - b_i)$$

Since $\forall i, S(i) = 2 \Rightarrow b_i \leq w_i^{'}$, $\sum_{i:S(i)=2} v_i \geq V$. Hence, we have shown that the target value is met. Now, we will show that $\sum_{i:S(i)=2} w_i^{'} \leq W'$. We consider two cases.

Case 1: If, $\forall i$, $S(i) = 2 \Rightarrow b_i = w_i^{'}$, then

$$\sum_{i:S(i)=2} b_i \leq W' \Rightarrow \sum_{i:S(i)=2} w_i^{'} \leq W'.$$

Case 2: If $\exists j$, such that $S(j) = 2$ and $b_j < w_j^{'}$ ($b_i = w_i^{'}$, for all i such that $S(i) = 2$ 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 selection function to take the left-hand branch, i.e. $S(j) = 1$, which will only increase the execution time. Now, since for all other i, $S(i) = 2 \Rightarrow b_i = w_i^{'} \Rightarrow \sum_{i:S(i)=2} v_i \geq T$ and $\sum_{i:S(i)=2} w_i^{'} \leq W'$.

Hence, $b_j > w_j^{'} - v_j \Rightarrow v_j > w_j^{'} - b_j$. We know that $w_j^{'} = v_m w_j$. If we can show that b_j is also a multiple of v_m, then we would have a contradiction, because v_j would become strictly greater than some multiple of v_m.

Again, in this case, $\sum_{i:S(i)=2} b_i = W'$, because otherwise if $\sum_{i:S(i)=2} b_i < W'$, 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).

$$\sum_{i:S(i)=2} b_i = W' \Rightarrow b_j = W' - \sum_{i:S(i)=2, i\neq j} b_i$$
$$\Rightarrow b_j = W' - \sum_{i:S(i)=2, i\neq j} w_i^{'}$$
$$(\text{because } \forall i, S(i) = 2, i \neq j \Rightarrow b_i = w_i^{'})$$
$$\Rightarrow b_j = v_m W - \sum_{i:S(i)=2, i\neq j} v_m w_i$$
$$(W' = v_m W, \text{ and } w_i^{'} = v_m w_i)$$
$$\Rightarrow b_j = v_m p \quad (\text{where } p > 0)$$

But, $v_j > w_j^{'} - b_j \Rightarrow v_j > v_m w_j - v_m p \Rightarrow v_j > v_m q$. This is a contradiction, as $q > 0$ and v_m is the maximum of all values. Hence there cannot exist j such that $S(j) = 2$ and $b_j < w_j^{'}$. Hence, $\forall i$, $S(i) = 2 \Rightarrow b_i = w_i^{'}$. Hence, the set $P = \{i | S(i) = 2\}$ is a solution to the Knapsack problem ■.

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.

4 Approximate WCIP

4.1 Setup

Let $G = (V, E)$ be the control flow graph (CFG) of the program under analysis. $V = \{b_1, \ldots, b_n\}$ is the set of basic blocks, E is the set of edges which represent the control transfer among the basic blocks. We perform AI-based multi-level Must and May cache analysis [17] to obtain an initial cache access classification (CAC) and cache hit miss classification (CHMC) of all memory accesses. 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 shared cache level. While 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 2-level cache hierarchy, with private L1 caches and a shared L2 cache. In the rest of the paper, when we say a cache access, we mean an access to the L2 cache, and when we use cache hit, we mean a memory access which hits the L2 cache.

Let $\{a_{i1}, a_{i2}, \ldots, a_{ik_i}\}$ be the set of memory accesses of b_i which hit the L2 cache (i.e. whose CHMC is Always Hit). Let Acc and Acc_H be the set of all L2 cache accesses and L2 cache hits in the program, respectively. Hence, $Acc_H \subseteq Acc$ and $Acc_H = \cup_{i=1}^{n} \cup_{j=1}^{k_i} a_{ij}$. 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 a LRU cache, cache block 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 1, while 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 [18].

A cache hit path of an access is a program path along which the access will experience a cache hit [2]. The cache hit path of an access a which references cache block m mapped to cache set s, is a program path which 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 occuring 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 the 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 maximum 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 maximum loop bound if the access is in a loop).

4.2 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 the shared cache hits present in the program, and then find the maximum number of misses generated by interferences.

Note that in WCIP, we find the maximum number of misses caused by interferences among the shared cache hits present on the worst-case path calculated in the presence of interferences. In the above approximation strategy, we are effectively assuming that all the shared cache hits in the program are present on π_{wc}. Remember that H is the total number of shared cache hits in the entire program. Let π_{wc}^{Int} be the worst-case path in the presence of interferences, and let T_{wc}^{Int} and H^{Int} be the WCET and number of shared cache hits of π_{wc}^{Int} respectively without considering the effect of interferences. Let T_{wc} be the WCET of π_{wc}, again without considering effect of interferences.

Then, $T_{wc} \geq T_{wc}^{Int}$, since π_{wc} is the worst case path in the program. Also, $H^{Int} \leq H$. Hence, the maximum number of shared cache misses caused by interferences among H^{Int} will be less than or equal to the maximum number of shared cache misses caused by interferences among H. If I^{Int} is the maximum increase in execution time due to interferences among cache hits in H^{Int} and I is the maximum increase among H, then $I \geq I^{Int}$. Hence, $T_{wc} + I \geq T_{wc}^{Int} + I^{Int}$. Note that $T_{wc}^{Int} + I^{Int}$ is the WCET obtained using WCIP. For approximate WCIP, we concentrate on finding $T_{wc} + I$.

The worst case path and its WCET without interferences (T_{wc}) can be easily determined, and hence our objective now is to calculate I, for which we need to determine the maximum number of cache misses that interferences can cause among 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.

4.3 The overlapping effect

We say that an interference affects a cache hit a, when it increases the age of the cache block m which will eventually be accessed by a, without any intervening accesses to m between the interference and a. Obviously, any interference which occurs just before the cache hit a will affect it. However, any interference which occurs on a cache hit path of a will also affect a. Since the cache hit path can contain other cache hits, interferences which 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 which are assigned just before the cache hit.

As an example, consider the program fragment shown in Figure 3, containing instructions a, b, c which access cache blocks m_1, m_2, m_3 respectively, all mapping to the same cache set, and hitting the cache. Assume the cache associativity is 4. Since the age of all the three cache blocks in the Must cache will be 3, their eviction distance will be 2. Now, any interference which occurs just before the access a is going to affect the 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 2 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.

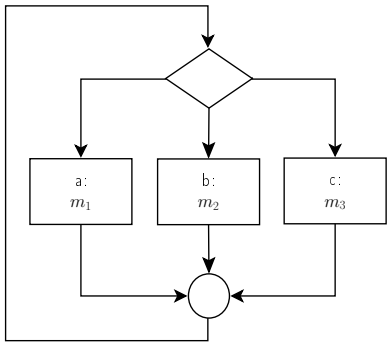


Figure 3: Example to illustrate the overlapping effect

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 the three.

We can use cache hit paths to calculate the OF. Only those interferences which 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 overlapping factor of cache hit h will be the number of cache hits who have a cache hit path which 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.

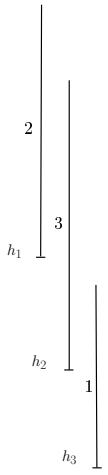


Figure 4:

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 OF 2, while the OF of h_3 is 1. With an interference budget of 3, all the three cache hits can be converted to misses by assigning 2 interferences before h_1 and 1 interference before h_2. However, if we assign interferences based on increasing eviction distances, we will first assign 1 interference before h_3 and then 2 interferences before h_1, thus using up the interference budget and obtaining only 2 cache misses.

Other selection strategies such as decreasing order of overlapping factors, 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 the 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 overlapping factor 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 which occur just before a cache hit will affect it. In the example of Figure 4, the maximum OF is 2, hence the interference budget would become 6. Now, the eviction distance of each cache hit can be met, thus resulting in 3 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

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 overlapping factor for every cache set

```
1  for every cache set s do
2  |  o_s ← 0 ;
3  |  for every cache hit a mapped to s do
4  |  |  OF_a ← 0
5  |  end
6  |  for every cache hit a mapped to s do
7  |  |  for every hit a' 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
```

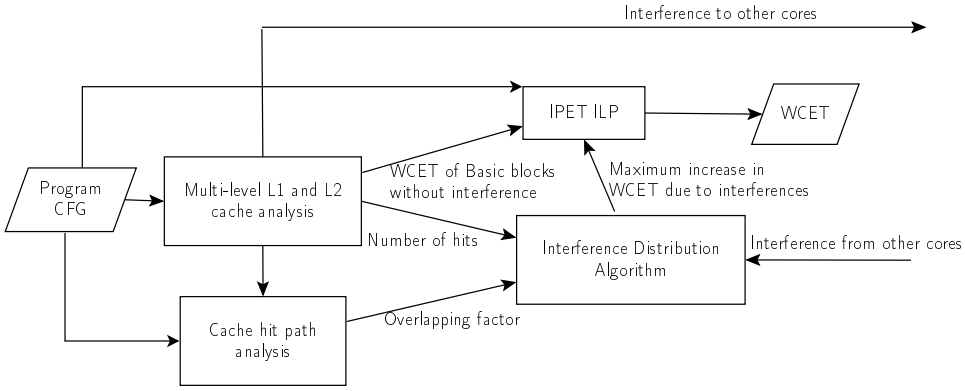


Figure 5: Approximate WCIP

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 overlapping factor for each cache set. It goes through every hit path of every cache hit, and finds the overlapping factor 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 7-9) 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 Abstract Interpretation based analysis used to find cache miss paths, proposed in [19]. The AI-based approach performs a constant number of traversals of the program CFG (this constant depends on the maximum size of the cache hit paths, which we set to twice the cache associativity) to find the fixpoint. Hence, the complexity of the approach is also linear in the size of the program.

4.4 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 multi-level 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 the shared cache hits in the program, as well as their eviction distances. We also obtain information about all the 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 the cache hits, and then find the maximum overlapping factor o_s for each cache set s, using Algorithm 1. The cache hit information, overlapping factor, and the interference information is used by the inteference distribution algorithm (explained below), to obtain the maximum increase in WCET due to cache misses caused by interferences. This is simply added to the WCET obtained by the IPET (Implicit Path Enumeration Technique) ILP to obtain the final WCET.

We call a cache hit having an eviction distance of k as a $k-interference$ cache hit ($1 \leq k \leq A$). We count all the $k-interference$ 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 ($1 \leq k \leq A$), Overlapping factor 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 overlapping factor 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 7-8). If there aren't 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 10-11), and we continue the interference distribution in the next iteration with $(k+1) - interference$ cache hits and the remaining interferences.

4.5 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 overlapping factor 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. Also, the increase in execution time due to interferences is a multi-dimensional, 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 trans-

lates 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 which 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 the 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 overlapping factor, we must still consider all the cache hits in the program.

Handling Code Sharing : We can simply ignore the effect of code sharing during our analysis, since 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 which 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 overlapping factor (in Algorithm 1).

4.6 Finding Cache hit paths

Apart from their use in calculating the Overlapping factor, Cache hit paths are also extensively used in the ILP-based approach for WCIP. Finding cache hit paths of accesses outside any loop is straightforward, as we can simply traverse the program in the reverse direction starting from the access, until we reach an access to the same cache block. However, for accesses inside loop, we may have to take the back edge and traverse the loop multiple times, and in general, it is not clear how many times one should do that, to ensure that

all cache hit paths have been discovered. Hit paths for accesses inside loops can span multiple iterations of the loop.

For example, consider again the program fragment in Figure 3. To determine HP (1), we only have to traverse the back edge once, but to determine HPs (2) and (3), we have to traverse it twice, and for HPs (4) and (5), three times. Hence, we need a systematic method to find all the hit paths of a cache access.

We propose an Abstract Interpretation based approach, which builds the hit paths of all shared cache hits in the program until a fix-point is reached. This program analysis is carried out in the reverse direction. Since we only need HPs of those accesses which are guaranteed shared cache hits (without interferences), we will only concentrate on such accesses (which can be determined using normal AI-based cache analysis). The general idea is that we traverse backward in the CFG starting from the access and keep track of the cache blocks encountered along different paths. Eventually, another instruction accessing the same cache block will be encountered, and the hit paths can be completed. Since we only consider guaranteed cache hits, all the program paths leading to the access must end with cache hit paths of the access.

Let Acc_H be the set of all guaranteed shared cache hits in the program, and let Acc be the set of all shared cache accesses ($Acc_H \subseteq Acc$). While the cache hit path was defined to be a sequence of accesses following program order, the sequence itself is not important when hit paths are used for WCIP. Hence, we will only maintain the set of accesses in every hit path. We also use a special symbol $\dashv$ to indicate that the hit path has been completed, i.e., the start instruction of the hit path has been encountered. Thus, for every hit path π, $\pi \in 2^{Acc \cup \{\dashv\}}$.

A cache hit can have multiple hit paths, and hence we maintain a set of hit paths for each cache hit. Our abstract lattice is the set of all functions $F = \{f | f : Acc_H \rightarrow 2^{2^{Acc \cup \{\dashv\}}}\}$. For $f_1, f_2 \in F$, we say that $f_1 \preceq f_2$ iff $\forall h \in Acc_H, f_1(h) \subseteq f_2(h)$. This is the standard power-set formulation of abstract lattice, with the join being defined as the point-wise union. Hence, $(f_1 \sqcup f_2)(h) = f_1(h) \cup f_2(h)$.

We now define the transfer function for a shared cache access made by the program. The transfer function for the rest of the instructions will be the identity function. Let $cb(a)$ and $cs(a)$ denote the cache block and the cache set accessed by a, respectively. $CAC(a)$ denotes the cache access classification of a at the shared cache level, and can be Always (if the access is guaranteed to reach the shared cache), Uncertain or Never. As shown in Figure 6, suppose shared cache access a accesses

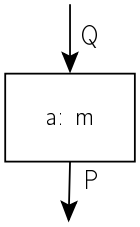


Figure 6:

cache block m mapped to cache set s. For $f_P \in F$, the transfer function T_{PQ} for this access is defined as $T_{PQ}(f_P) = f_Q$, where

$$f_Q(h) = \begin{cases} \{\{a\}\} \cup \{\pi \cup \{\dashv\} | \pi \in f_P(h)\} \\ \qquad \text{if } a = h, \\ \{\pi \cup \{a, \dashv\} | \pi \in f_P(h) \wedge \dashv \notin \pi\} \\ \cup \{\pi | \pi \in f_P(h) \wedge \dashv \in \pi\} \\ \qquad \text{if } cb(h) = cb(a) \text{ and } CAC(a) = A, \\ \{\pi \cup \{a\} | \pi \in f_P(h) \wedge \dashv \notin \pi\} \\ \cup \{\pi | \pi \in f_P(h) \wedge \dashv \in \pi\} \\ \qquad \text{if } cb(h) \neq cb(a) \text{ and } cs(h) = cs(a) \\ \qquad \text{and } CAC(a) = A \text{ or } U, \\ f_P(h) \\ \qquad \text{otherwise} \end{cases}$$

The transfer function operates separately on each hit path of every cache hit. The presence of a path π in $f_P(h)$ indicates that π is a path from the access a to cache hit h, and it is a sub path of some cache hit path of h.

First, we consider the hit paths of the access a itself (if $a \in Acc_H$). We add the path $\{a\}$, to begin the collection of hit paths of a, while all existing paths of a are completed by adding $\dashv$. An existing path of a will be present when a is inside a loop, and it has already been encountered once during an earlier AI iteration.

In the second case, we consider the paths of those instructions h which access the same cache block $cb(a)$. If the CAC of a is Always, then a is the instruction guaranteed to bring $cb(a)$ into the cache, which will eventually by accessed by h, causing h to be a cache hit. Hence, any existing paths of h which do not contain $\dashv$ are completed, while existing paths of h which have already been completed are retained.

In the third case, we consider the paths of those instructions h which access a different cache block $cb(h)$, mapped to the same cache set $cs(a)$. In this case, cache block $cb(a)$ will conflict with $cb(h)$ and therefore the access a is added to any existing path of h, which has not been completed, while all completed hit paths are retained. Finally, paths of instructions which do not access the cache set $cs(a)$ are not modified.

It is easy to see that the transfer function is monotonic, since it operates individually on each hit path of every cache hit. It either adds a new path, or adds new accesses to an existing path, but this depends solely on the properties of the access or the path itself. Formally, if $f_1 \preceq f_2$, then $f_1(h) \subseteq$

$f_2(h)$. Since all hit paths in $f_1(h)$ are also present in $f_2(h)$, after applying the transfer function, the transformed hit paths in $T_{PQ}(f_1)(h)$ will also be present in $T_{PQ}(f_2)(h)$. Moreover, the abstract lattice F is finite, hence, Kildall's algorithm can be used to determine the fixpoint of the analysis. All the completed cache hit paths of accesses in H will be gathered at the start of the program in the fixpoint solution.

5 Interaction with the shared bus

In most multi-core 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, because 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 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 which 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 as well as the schedule are known, the delay can be accurately bounded.

Assuming that the shared cache can only fulfill one access request at a time, and that access requests cannot be prempted (i.e. once a memory block is requested, the cache must satisfy the request before working on the next request), the length of a slot assigned to a core must be greater than the main memory latency, so that a request can be fulfilled in the same slot. 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 over-approximation, and hence, previous works ([20, 21]) 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, since that would require considering the worst-case interference arrival individually for every access, resulting in high over-approximation 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 ([20, 21]) to accurately bound the shared bus delay. 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 (for example, using the techniques described in [20, 21]). 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} which 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.

Upto 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 hence extra cache miss penalty c_p. Let β_{i_1} be the actual time at which the access s_{i_1} takes place, and $o_{i_1} = \beta_{i_1} mod(n_c s_l)$ 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 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 just 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 [20], which states 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 micro-architectural component except the shared bus, the two executions will differ by at most $n_c s_l$ cycles. 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 hence the total increase in shared bus delay will be upper bounded by $2n_c s_l I^{Int}$.

6 Experimental Evaluation

We use the open-source WCET analyzer Chronos [22] for our experiments. Chronos is built on top of the SimpleScalar simulation framework, and requires benchmarks compiled for the simplescalar PISA architecture. Since we are focusing on the impact of shared instruction caches on the WCET, we assume a perfect data cache and also ignore the effect of the processor pipeline or the shared bus. We use 27 benchmarks from the Mälardalen

WCET benchmark suite[2]. We use *lp_solve* to solve the generated ILPs, and our experiments were performed on a 4-core Intel i5 CPU with 4 GB memory. We assume a 2-core architecture with a 2-level cache hierarchy.

For shared instruction cache analysis, we implemented three different techniques : (1) [3]'s approach, which considers the effect of all interferences on all shared cache hits, (this technique is also used for shared cache analysis in multi-core Chronos [5]),(2) ILP-based approach for WCIP, as proposed in [2] and (3) the approximate technique for WCIP, proposed in this paper. We compare the precision of WCET obtained using all three techniques.

The code size of the benchmarks ranges from 0.2 KB to 60 KB, with an average size of 23.5 KB. For the initial set of experiments, we assume a 1 KB 4-way L1 I-cache, and a 4 KB 8-way L2 I-cache, with block size of 32 bytes. Later, we will also show the impact of changing the L2 cache size on the precision of the estimated WCET. To compute WCET of a benchmark on a 2-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 which causes the maximum shared cache interference.

With the above assumptions, we calculated the WCET using all the three techniques, and found that WCIP (both the ILP-based and approximate approach) gave lower WCETs for 11 of the 27 benchmarks as compared to Hardy et. al.'s approach. The WCETs were same for the rest of the benchmarks. Figure 7 shows the precision improvement of WCET (in %) obtained using the two WCIP approaches over Hardy et. al.'s approach. (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).

The precision improvement in WCET using the ILP-based WCIP and approximate WCIP is equal for almost all benchmarks. Moreover, the precision improvement using approximate WCIP is actually slightly higher in some benchmarks. The reason is that the ILP itself introduces some imprecision while handling shared cache hits inside loops. The average precision improvement over the 11 benchmarks for ILP-based WCIP is 28.6 %, and for approximate WCIP, it is 29 %.

The real advantage of approximate WCIP over ILP-based WCIP is in the analysis time required to obtain the WCET. The complexity of the ILP increases with the number of shared cache hits in the program, while the complexity of approximate WCIP is independent of the number of cache hits. The average analysis time for ILP-based WCIP for the 11 benchmarks was

[2]WCET Projects/Benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

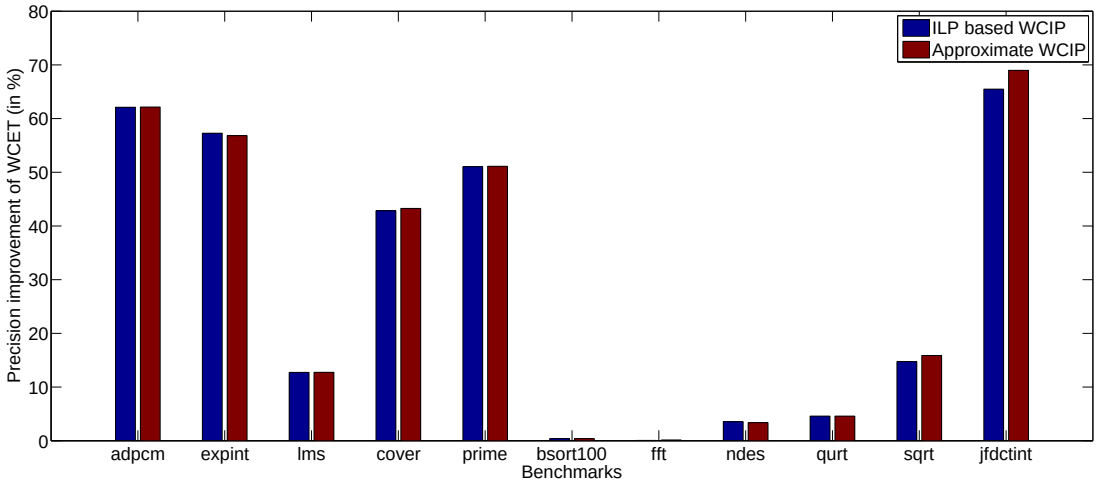


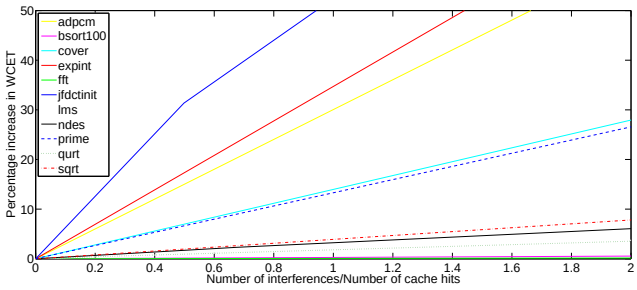
Figure 7: Precision improvement of WCET obtained using (1) ILP-based WCIP and (2) Approximate WCIP over Hardy et al.'s approach [3] for a 4 KB L2 cache

0.82 seconds (with maximum of 4.58s for *ndes*), while the average analysis time for approximate WCIP was 0.002 seconds.

6.1 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 the 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 minimum of 70 % and maximum of 100 %). This shows that the first assumption will likely not have a major impact on the precision of approximate WCIP.

The number of interferences caused by *nsichneu* expressed as a percentage of the total number of cache hits in a benchmark was 56.5 %, averaged across all benchmarks (ranging from 1.6 %



for $jfdctint$ to more than
100 % for $fft, ndes, qurt$).
The average eviction distance (across all shared cache hits) across all benchmarks was 7.6, (ranging from 6.1 to 8), which means that an average of 7.6 interferences were required to cause a single cache miss across all benchmarks. The second assumption made by approximate WCIP is to remove the overlapping effect, by multiplying the original budget of interferences with the maximum Overlapping Factor (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 6.1 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), for 11 benchmarks. While 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 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 small for higher number of interferences as well.

6.2 Changing the cache size

We also experimented with two other L2-cache configurations, a 2 KB L2 cache (half the original size) and a 8 KB L2 cache (double the original size). Figure 8 shows the precision improvement in WCET obtained using ILP-based and approximate WCIP, over Hardy et al.'s approach, for both the cache configurations.

For the 2 KB L2 cache, 7 out of the 11 benchmarks showed precision improvement. Again, both the ILP-based and approximate WCIP give similar results, and the average precision improvement over the 7 benchmarks for both the approaches is 36.6 %. Note that the average precision improvement for the same 7 benchmarks for a 4 KB L2 cache was 42.6 %. 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.

For the 8 KB L2 cache, all benchmarks (except *sqrt*) showed slighly higher precision improvement (than the 4 KB L2 cache), and the average

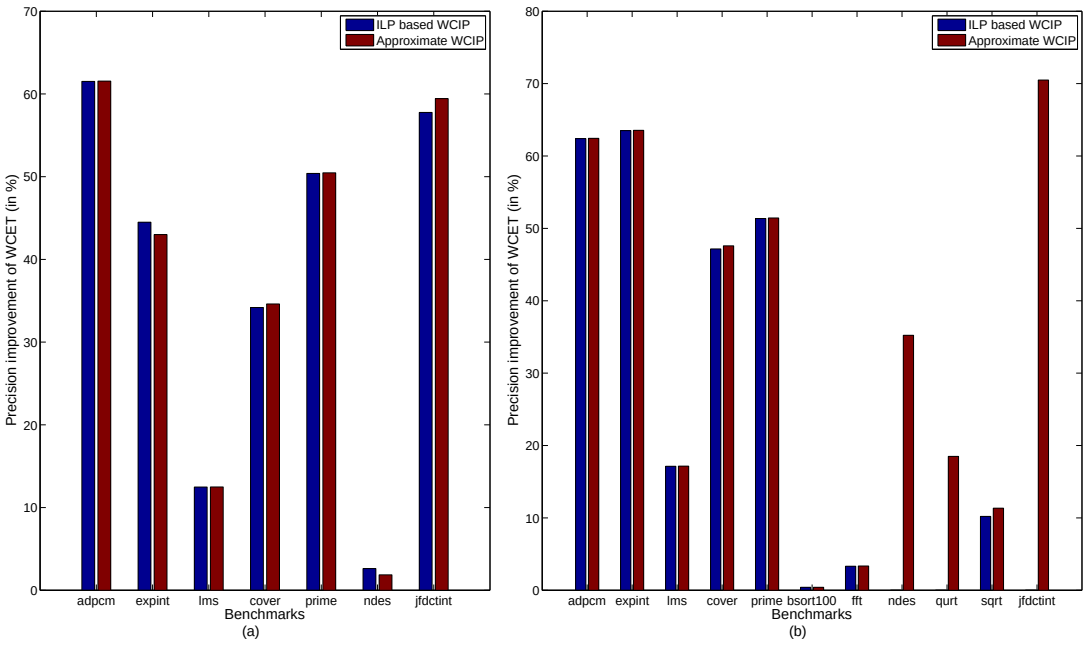


Figure 8: Precision improvement of the WCET for two different cache configurations (a) 2 KB L2 cache (b) 8 KB L2 cache

precision improvement for approximate WCIP over the 11 benchmarks was 36.29 %. However, for ILP-based WCIP, the ILP solver was not able to solve the generated ILPs for 3 benchmarks (*ndes*, *qurt* and $jfdctint$) 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. For all benchmarks, approximate WCIP took less than 1 second to find the WCET, with high precision improvement in the 3 benchmarks for which ILP-based WCIP fails. This illustrates the advantage of using approximate WCIP over ILP-based WCIP.

6.3 Comparison with Simulated WCET

We also compare the estimated WCETs obtained using the three techniques, with the WCET obtained using simulation. To obtain the simulated WCET, we use the modified version of Simplescalar framework [23], used for validation in Multi-core chronos [5]. The modified version supports simulation of shared cache and shared bus, among other architectural components. For our purposes, we only simulate the effect of the shared cache, and assume zero shared bus delay and no pipeline hazards.

As has been noted by [5], it is difficult to simulate the exact interleaving for accesses which will result in the worst-case scenario for shared caches. Hence, the simulated WCET may highly under-estimate the actual WCET of the program. It is also difficult to obtain the exact worst-case input for some of the benchmarks, such as *qurt* and *ndes*, which involve branching based on complex mathematical calculations.

Figure 9 depicts the WCET estimation ratio, calculated as $\frac{Estimated\ WCET}{Simulated\ WCET}$, for 11 benchmarks. We assume a 1 KB L1 cache and 4 KB L2 cache (same as in the first experiment). We first find the simulated and estimated WCETs assuming a private L2 cache, to determine the impact of infeasible paths, private cache analysis, etc. on the overestimation of the estimated WCET. Then, we assume the same L2 cache shared between two cores, with the benchmark *nsichneu* running on the other core, and find the simulated and estimated WCETs, and the overapproximation ratio. The estimated WCETs are determined using the three shared cache analysis techniques. Figure 9 shows the WCET overestimation ratio of the 11 benchmarks for all the four cases.

Except for *expint*, the overestimation ratio for all the other benchmarks, in the case where the shared L2 cache is analyzed using WCIP, is almost the same as the overestimation ratio for private L2 cache. This shows that shared cache analysis using WCIP does not introduce large amounts of imprecision in the estimated WCETs. The average overestimation ratio for private L2 cache

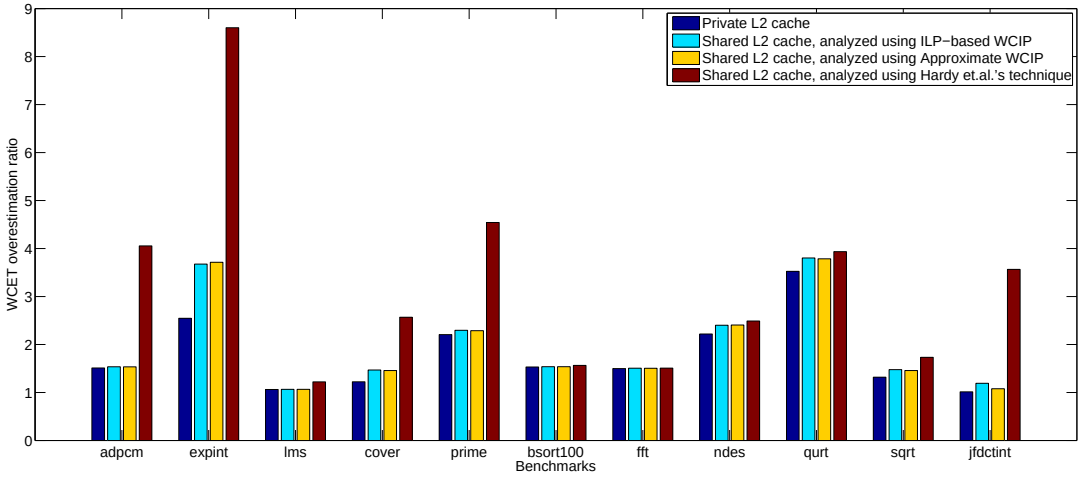


Figure 9: WCET overestimation ratio as compared with simulated WCET, for (1) private L2 cache, (2) shared L2 cache analyzed using ILP-based WCIP, (3) shared L2 cache analyzed using approximate WCIP and (4) shared L2 cache analyzed using Hardy et. al.'s approach

is 1.78, while for shared L2 cache analysis using ILP and approximate WCIP, it is 1.99 and 1.98 respectively. On the other hand, Hardy et. al.'s analysis introduces large amounts of imprecision, and the average overestimation ratio is 3.25.

6.4 Cache partitioning

Cache partitioning is a hardware-based approach to simplify shared cache analysis in multi-core 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 hence 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 (also called *columnization* [9]), where each core is assigned a subset of ways in all cache sets, and (2) Horizontal partitioning (also called *bankization*), where each core is

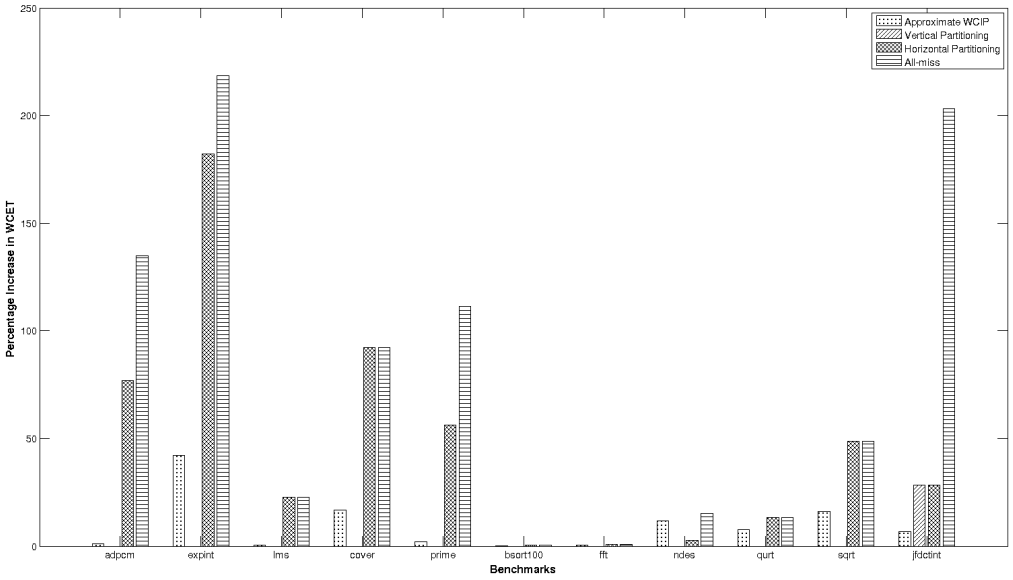


Figure 10: Graph showing the percentage increase in WCET obtained using (1) Approximate WCIP, (2) Vertical cache partitioning, (3) Horizontal cache partitioning and (4) All shared cache accesses as misses

assigned a subset of cache sets. For our 2-core architecture, we divided the cache equally between both the cores.

For vertical partitioning experiments, we decrease the shared cache associativity from 8 to 4, while for horizontal partitioning experiments, we decrease the number of cache sets from 16 to 8. For approximate WCIP, we set the adversary program as *nsichneu*. Figure 10 shows the percentage increase in the WCET for 11 benchmarks obtained using approximate WCIP and the two cache partitioning techniques, and assuming that all shared cache accesses as misses, as compared with WCET of the benchmarks running in isolation with an un-partitioned 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, while $WCET_{shared}$ is obtained using either approximate WCIP or the two cache partitioning techniques or assuming that all shared cache 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, notice 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 4 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 6 cache blocks in every cache set, and the percentage increase in its WCET due to vertical partitioning is greater than approximate WCIP.

Horizontal partitioning, on the other hand, is highly ineffective, as it introduces greater imprecision than both the other techniques for all benchmarks except *ndes*. The average increase in the WCET across the 11 benchmarks for approximate WCIP is 9.6 %, while for horizontal partitioning, it is 47.7 %. Also, 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 78.3 %, which highlights the importance of shared cache analysis in accurate WCET estimation.

From the above 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 above experiment, we used the worst case adversary. By selecting programs which generate less shared cache interference, the WCET can be controlled using WCIP, but with cache partitioning, there will be no impact on the WCET. With vertical partitioning, we are effectively assuming that there is an adversary program which generates a constant amount of shared cache interference (in this case, 4 interferences for every shared cache access in the program under analysis). Moreover, in the instance where the number of cache blocks accessed per cache set goes beyond 4 (for $jfdctint$), vertical partitioning introduces greater imprecision than WCIP.

So far, cache partitioning/locking have been used aggressively to guarantee *zero* shared cache interference, so that private cache analysis techniques can be directly applied to find the WCET. However, this requires providing guaranteed cache space (generally equal to the working set) to a task before it starts execution (for example, as proposed in [12]), and locking this cache space for the entire execution, even though the task may not need it. We have shown that cache analysis can, in fact, cope with limited amounts of cache interference and can still be used to estimate fairly accurate WCETs. In general, we believe that a relaxed form of cache locking/partitioning could be used (for example, by allowing tasks to share partitions) to provide non-zero but small guarantees on the shared cache interference, and these guarantees can then be used to obtain precise WCET estimates.

7 Conclusion and Future Work

Estimating the WCET of programs running on multi-core architectures is an important step towards using multi-cores in real-time systems. Shared cache analysis plays a crucial role in obtaining precise WCET estimates on multi-cores, and Worst case interference placement (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, even for simple programs without branches or loops and direct-mapped shared caches. We also propose an approximate technique for WCIP that bypasses the hard problem of finding the worst case path in the presence of interferences. While ILP-based WCIP is NP-Hard, approximate WCIP has a time complexity linear in the size of the shared cache, thus guaranteeing fast analysis time irrespective of program size.

Experimentally, we find that approximate WCIP is as precise as ILP-based WCIP across all benchmarks, with a substantial reduction in analysis time. We also compare the effectiveness of shared cache analysis using WCIP against fixed cache partitioning, and find that while horizontal partitioning results in very high WCET estimates, vertical partitioning can actually perform better than WCIP, but WCIP gives more control on the WCET by selecting appropriate programs to run on other cores.

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. It does not matter which programs run on the other cores, as long as they do not exceed the upper bound. Finding appropriate upper bounds on interference to maximize utilization and schedulability of task sets is another interesting problem.

References

[1] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguire, Daniel Grund, Jrg Herter, Jan Reineke, Bjrn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In *Verification, Model Checking, and Abstract Interpretation*, pages 3–22. 2010.

[2] Kartik Nagar and Y N Srikant. Precise shared cache analysis using optimal interference placement. In *Real Time and Embedded Technology and Applications Symposium*, 2014.

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

[4] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Syst.*, 17(2-3):131–181, December 1999.

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

[6] Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *Real Time Systems Symposium*, 2011.

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

[8] Vivy Suhendra and Tulika Mitra. Exploring locking and partitioning for predictable shared caches on multi-cores. In *Design Automation Conference*, 2008.

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

[10] Man-ki Yoon, Jung-Eun Kim, and Sha Lui. Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems. In *Real Time Systems Symposium*, 2011.

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

[12] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *EMSOFT*, 2009.

[13] Huping Ding, Yun Liang, and T. Mitra. Shared cache aware task mapping for wcrt minimization. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 735–740, Jan 2013.

[14] Gabriel Fernandez et al. Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art. In *International Workshop on Worst-Case Execution Time Analysis*, pages 31–42, 2014.

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

[16] Silvano Martello and Paolo Toth. *Knapsack Problems : Algorithms and Computer Implementations*. John Wiley and Sons, 1990.

[17] Damien Hardy and Isabelle Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *Real-Time Systems Symposium*, 2008.

[18] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. 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*, pages 153–162, 2010.

[19] Kartik Nagar and Y.N. Srikant. Path sensitive cache analysis using cache miss paths. In *Verification, Model Checking, and Abstract Interpretation*, pages 43–60. Springer Berlin Heidelberg, 2015.

[20] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Static analysis of multi-core tdma resource arbitration delays. *Real-Time Systems*, 50(2):185–229, 2014.

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

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

[23] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002.