Precise shared cache analysis using optimal interference placement

Kartik Nagar, Y.N. Srikant
Dept. of Computer Science and Automation,
Indian Institute of Science,
Bangalore, India.
Emails : {kartik.nagar, srikant}@csa.iisc.ernet.in

***Abstract*—Determining the Worst Case Execution Time (WCET) of programs running on a multi-core architecture is a challenging problem, that is hampering the use of multi-cores in real-time systems. The highly imprecise WCET estimates obtained using the current state-of-the-art analyses has prompted research in the direction of making the multi-core architecture itself more estimation-friendly, but there has been little effort to make the WCET analysis more precise. The main difficulty in analyzing programs running on multi-core architectures arises from the fact that interferences to shared resources (such as shared cache) from other cores can occur at any time. Hence, to perform safe micro-architectural analysis, current approaches assume that all interferences occur at all times, which results in significantly imprecise analysis WCET estimates. However, since we are interested in the WCET, we can instead assume that the interferences will come at the worst possible program points, causing maximum increase in the execution time. In our work, we formulate an ILP problem to determine these worst case interference points, from the perspective of a shared cache, and determine the WCET by assuming that the interferences come at those program points. Our approach provides an average precision improvement of 25.63% over earlier analysis for benchmarks which perform a reasonable number of accesses to the shared cache.**

I. Introduction

Multi-cores are widespread in today's computing devices, from hand-held mobiles to servers and workstations. The multi-core architecture allows one to leverage the increasing computing power of chips without increasing the complexity of their design. Multi-cores are here to stay, and it is expected that the number of cores will continue to increase, providing higher computational power. Using multi-cores for real time systems has proved difficult, because real time systems require guarantees on the execution time of programs, and obtaining precise estimates of WCET on multi-core architectures has not been easy.

Caches have a major impact on the execution time of programs, and a number of techniques have been developed to precisely capture this impact for different cache architectures in single-core machines. Extending these techniques to shared cache analysis in a multi-core architecture, however, has not worked. When a cache is analyzed from the point of view of a single core, the sequence of accesses to the cache can be determined with high precision, which results in highly precise cache hit-miss analysis. However, a shared cache in a multi-core architecture satisfies accesses originating from different cores, and the sequence of these accesses cannot be known statically, as all inter-leavings are possible at runtime.

If we assume that the tasks running on all cores are known, then the accesses made to the shared cache by each task can be precisely determined. The current state-of-the-art approaches to shared cache analysis [1] analyze the shared cache separately from each core's point of view. While performing the analysis from one core (say core i), we know the exact sequence of accesses from core i, and the set of accesses made by the rest of the cores, but we do not know when these accesses will occur (or their order). Hence, it is assumed that all accesses from the rest of the cores can occur between any two accesses from core i, and the shared cache is updated accordingly. This results in a highly pessimistic hit-miss analysis, since all the accesses to the shared cache by core i will be classified as misses if the total number of cache blocks accessed by other cores is greater than or equal to the cache associativity (which is usually a small number).

Even though we do not know the sequence of accesses to the shared cache from the other cores, we know the number of accesses, and we know that these accesses will arrive at some time during the execution of the task on core i. Let us call the accesses arriving from other cores to the shared cache as interfering accesses. An interfering access may or may not cause any extra misses to core i. For example, if the shared cache is empty (from the point of view of core i), or if none of the cache blocks in the shared cache are accessed later by core i, then an interfering access will not cause any miss. However, if all the cache blocks in the shared cache are accessed immediately after the interfering access, then it can cause many misses (at most the cache associativity). The damage done by an interfering access thus depends on the sequence of accesses from core i, and can be statically determined.

If we distribute the interfering accesses across the program (running on core i) such that the maximum damage is caused, then we can safely use the resulting hit-miss classification, since any other arrival of the interfering accesses is guaranteed to cause lesser damage. We propose an integer linear programming (ILP) based approach to determine the distribution of interfering accesses across the entire program, which will cause the maximum damage. The rationale behind using ILP is that we want all the interfering accesses to be distributed entirely on the worst case path (i.e. the path having the maximum execution time) in the program. Instead of distributing the interfering accesses on each program path and then finding the worst case path, we use the Implicit Path Enumeration

978-1-4799-4829-1/14/$31.00 ©2014 IEEE

Technique (IPET) [2] to find the worst case path. We encode the problem of finding the optimal distribution of interfering accesses as an ILP, and then combine this ILP with the ILP as proposed by IPET.

While generating the ILP, we maintain the property that if any arrival of interfering accesses can cause an instruction to experience cache miss during actual execution, then the same distribution of accesses in our ILP will also cause a cache miss for the instruction. This ensures that our approach will generate a safe WCET in the presence of interfering accesses, since the objective function of our ILP maximizes the execution time while considering all feasible misses caused due to interferences. We have implemented our approach for shared L2 instruction cache in a 2-core system as an extension of the timing analyzer Chronos [3]. Almost all benchmarks that make a reasonable number of accesses to the L2 cache benefit from our approach, and in many cases, the precision improvement is very high.

Using ILPs raises the issue of scalability, and we found in our experiments that determining the optimal distribution of all interfering accesses may not be feasible for all programs. However, in such cases, depending on the time budget allocated to the WCET estimation phase, we can select a subset of interfering accesses to be distributed optimally, or limit optimal interference placement to a selected program segment. We found substantial improvement in precision even in the case of such selective optimization.

The rest of the paper is organized as follows: Section 2 discusses previous work in shared cache analysis. Section 3 contains details of the targeted multi-core architecture and other assumptions. Section 4 highlights the imprecision of the current approach using a simple example. Section 5 provides a step-by-step construction of our ILP for optimal interference placement, along with the proof of safety. Section 6 contains experimental results, and also discusses the scalability of our approach. We conclude in Section 7.

II. Related Work

All variants of shared cache analysis in the literature assume that all interferences occur at all program points. Yan and Zhang [4] introduced the 'always-except-one' classification for instructions inside loops which access the shared cache, if the interfering access is not inside a loop. However, their approach works only for direct-mapped caches, and they assume that the interfering access can occur anywhere in the program. In a later work [5], they take into account the sequence of accesses to rule out certain misses arising due to infeasible interfering accesses. However, the feasible interferences could still occur anywhere in the program.

Hardy et al. [1] proposed a shared cache analysis for set-associative caches, where they first perform the cache analysis of the shared cache separately for each core, assuming no interference. Then, they change the shared cache state at each program point, assuming all interfering accesses happen just before every access in the program. They also propose a hardware mechanism that forces certain accesses to bypass the shared cache, thus reducing the number of interfering accesses.

Yan Li et al. [6] proposed a timing analysis for message-passing programs, where they concentrate on the question of whether two tasks can run concurrently, and if so, for how long. Again, during the time that tasks are running concurrently, they assume that all interferences can come at any program point. The shared bus is another important hardware resource which introduces unpredictability in timing analysis, and a number of works (e.g. [7]) have looked at precise analysis of shared bus, and integrated analysis of shared bus and cache. For shared cache analysis, these works use the same approach proposed by [1]. Model checking has also been used for shared cache analysis in [8] to find out accesses which occur on infeasible paths and hence can be safely ignored while identifying interfering accesses.

Hardware approaches ([9], [10], [11], [12]) 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 single-core 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 [13], which may prevent task sets to be scheduled.

III. Setup

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. For the discussion in this paper and as well as our experiments, we have only considered a shared instruction cache, but our analysis can be directly applied to data cache or unified cache without any changes. We assume that the cache replacement policy is LRU (Least Recently Used). We also assume a timing anomaly free architecture.

We perform the standard Must and May analysis [14] at the private lower levels and the shared level separately for each core. As a result, for each access in the programs (at all the cores), we have a Cache Access Classification (CAC) and Cache Hit Miss Classification (CHMC) at the shared cache level. The CAC determines whether an instruction will access a cache level, and can be Always (A), Uncertain (U) or Never (N). In the first two cases, the access has to be considered at the shared cache, and in the last case, the access can be ignored. Note that the CAC at the shared level depends only on the analysis of caches at the lower levels, which are all assumed to be private. Hence, interferences from other cores will not affect the CAC.

The CHMC at the shared cache can be Always-Hit (AH), Always-Miss (AM) or Uncertain (U). The CHMC will be determined without considering interfering accesses from other cores. Accesses classified as AM or U will not be affected by interferences, since they are already counted as misses, and interfering accesses will only increase the number of misses. Hence, we will concentrate on the accesses which are classified as AH at the shared level. Note that we could also consider the accesses classified as First-Miss (FM) using persistence analysis at the shared cache level.

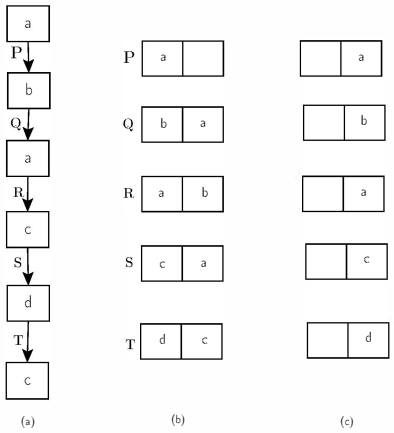

Fig. 1. Example to show the imprecision of current approach. (a) Program fragment (b) Shared cache states (c) Shared cache states assuming 1 interfering access

The state-of-the-art method [1] to deal with interfering accesses is to find the number of interfering cache blocks for each cache set and then update the shared cache state obtained after the must analysis at each program point. Assuming no code-sharing across cores, the update simply increases the age of all cache blocks in the shared cache by the number of interfering cache blocks. Since it is not known when the interfering accesses will come, it is assumed that all interfering accesses can come between any two accesses of the program being analyzed. The updated shared cache states are then used to obtain the new (and safe) CHMC at the shared cache level. However, this is a highly pessimistic approach, as we show using a simple example in the next section.

IV. Example

Consider the program fragment in Figure 1a. Assume that a, b, c, d are cache blocks at the shared cache level mapped to the same cache set, and all the accesses reach the shared cache (i.e. CAC is A or U). Further, assume that the associativity of the shared cache is 2. Figure 1b shows the shared cache states obtained after must analysis, assuming no interferences. Using these cache states, the CHMC of the second access to a and the second access to c will be AH (since both a and c are present in the cache just before their accesses). Now, suppose the program fragment is running in a multi-core environment, and there is one interfering access coming from other cores. Figure 1c shows the updated cache states, obtained by increasing the age of each cache block in the original cache states by 1. Both the second accesses to a and c will now be classified as Miss (U), since they are not present in the updated cache state just before their accesses.

However, notice that if the interfering access comes at program points P or Q, it will only affect the access to a. Similarly, if it arrives at S or T, it will only affect the access to c, while if it arrives at R, neither of the two accesses will be affected. Hence, one interfering access can cause at most one miss in the program, and one of the two accesses is guaranteed to remain a cache hit. Note that this example can be easily expanded to contain more accesses of the form $m - m' - m$, so that the second access to m would be a cache hit. The state-of-the-art shared cache analysis will still report each such access to be a cache miss with just one interfering access, while during actual execution, at most one cache miss would be caused due to interferences. Similarly, if the entire program fragment is enclosed in a loop, then accesses in all iterations will reported as Miss by the state-of-the-art cache analysis. During actual execution, the interfering access will come during one iteration, causing at most one cache miss.

Hence, it is clear that assuming all interferences occur at all program points results in a highly pessimistic analysis and can blow up the WCET estimate. While it is true that the interfering accesses can arrive at any program point during actual execution, and thus no access can be safely classified as AH, we are not really interested in which accesses are guaranteed to be hits. We are actually interested in the maximum possible execution time, taking into account the interfering accesses, which can obtained by estimating the maximum number of misses caused by interferences. We have just seen that interfering accesses arriving at different program points can cause different number of cache misses, and the damage caused depends only on the sequence of accesses in the program being analyzed. If we can statically distribute the interfering accesses across the program such that they cause the maximum possible number of misses, then we can use the resultant shared cache states to obtain a safe CHMC.

We propose to use Integer Linear Programming to solve this optimization problem. ILP is already an integral part of the WCET estimation process, as most WCET analyzers use the IPET formulation of ILP [2] to determine the worst case path in the program. ILP has also been used for cache hit-miss analysis, by constructing the Cache Conflict Graph (CCG) of programs ([15], [16]). However, because of scalability issues and the success of the Abstract Interpretation based approaches, most WCET analyzers employ AI-based techniques for cache analysis.

The IPET formulation requires the WCET of each instruction in the program as a constant. In our formulation, we relate the WCET of each instruction to the number of interfering accesses before the instruction.

As an example, for the program fragment of Figure 1a, we associate binary variables x_a and x_c with the second accesses to a and c respectively, to store their hit/miss status. Integer variables z_1, z_2, z_3, z_4, z_5 are associated with the program points P,Q,R,S,T respectively, and they store the number of interfering accesses occurring at those program points. Now, for the second access to a to become a miss, there needs to be at least one interfering access at P or Q, and for the second access to c to become a miss, there needs to be at least one interfering access at S or T. Consider the following ILP:

$$\text{Maximize } x_a + x_c,$$
$$subject\ to$$
$$x_a \leq z_1 + z_2 \quad (1)$$
$$x_c \leq z_4 + z_5 \quad (2)$$
$$z_1 + z_2 + z_3 + z_4 + z_5 \leq 1 \quad (3)$$

The objective function maximizes the number of misses caused due to interferences. Equations 1 and 2 depict the access constraints, which state the minimum number of interferences to cause the access to be a miss. Equation 3 is the

TABLE I. NOTATION

Symbol	Explanation
y_i	Integer variable storing the execution count of basic block b_i
x_{ij}^h	Integer variable storing the number of shared cache hits of instruction a_{ij}
x_{ij}^m	Integer variable storing the number of shared cache misses of instruction a_{ij}
x_{ij}^π	Integer variable storing the number of shared cache misses of instruction a_{ij} along path π
z_{ij}	Integer variable storing the total number of interfering accesses occurring just before instruction a_{ij}
w_{ij}	Integer variable storing the execution count of edge between basic blocks b_i and b_j
p_{ij}^π	Eviction distance of instruction a_{ij} along path π
c_i	Execution time of basic block b_i not affected by interferences
B_s	Number of interfering accesses mapped to cache set s
B_s^{cb}	Number of interfering cache blocks mapped to set s

interference budget constraint, which encodes the maximum number of interferences available. Solving the above ILP will give the maximum value of the objective function to be 1, with either one of z_1, z_2, z_4, z_5 assigned as 1.

V. THE ILP FORMULATION

A. Notation

We now give a general description of our ILP for an arbitrary program. We analyze programs running on each core separately. Let $b_1, \ldots, b_n$ be the basic blocks of the program running on core c. Let $a_{i1}, \ldots, a_{il_i}$ be the instructions in basic block b_i for all $i = 1, \ldots, n$, whose CAC is A or U at the shared cache level. Basic block i contains l_i such instructions. We ignore accesses which are satisfied by the private caches, since they will not be affected by the interferences from other cores. We associate a constant c_i with basic block b_i, which is the total execution time of those instructions in b_i which are not affected by interferences. This includes accesses which are satisfied by the private caches, non-memory-accessing instructions, etc. Let cb_{ij} be the cache block in the shared cache accessed by a_{ij}.

Let A be the associativity of the shared cache. Let $\mathcal{M}$ be the set of all cache blocks that can be stored in the shared cache. Let $cachestate_{ij}^s : \{1, \ldots, A\} \rightarrow 2^{\mathcal{M}}$ be the abstract shared cache state of set s as determined by the must analysis (ignoring the interfering accesses), just before instruction j of basic block i. If $m \in cachestate_{ij}^s(h)$, then h is the *age* of m just before a_{ij}. If $cb_{ij} \in cachestate_{ij}^s(h), 1 \leq h \leq A$, then the instruction a_{ij} will be a guaranteed shared cache hit (without interferences). Table I lists the variables and constants that will be used in our ILP formulation.

B. Objective Function

Before specifying the objective function of our ILP formulation, we briefly explain the IPET formulation, which is used to find the worst case path in a program. Let $e_1, \ldots, e_n$ be the WCET of basic blocks $b_1, \ldots, b_n$. $y_1, \ldots, y_n$ are integer variables storing the execution count of basic blocks, and w_{ij} stores the execution count of the edge between b_i and b_j. The IPET formulation is:

$$\text{Maximize } \sum_{i=1}^{n} e_i y_i, \text{ subject to}$$
$$\forall i = 1, \ldots, n, \; y_i = \sum_{j \in pred(b_i)} w_{ji} = \sum_{k \in succ(b_i)} w_{ik}$$
$$\text{Loop Constraints ...}$$

$pred(b)$ and $succ(b)$ give the predecessors and successors of basic block b respectively. The objective is to find the execution counts of basic blocks which maximizes the execution time of the program. The execution counts are constrained by the program structure, which basically places the restriction that the number of times execution enters a basic block must be the same as the number of times execution leaves the basic block, and this will also be the execution count of the basic block. The variable w_{ij} stores the number of times execution left basic block i and entered basic block j.

The loop constraints give an upper bound on the execution count of the header basic block of each loop in the program, and are typically supplied by the programmer. The WCET of basic blocks in the above formulation are assumed to be constants and are obtained using micro-architecture analysis. After solving the ILP, the basic blocks whose execution counts are non-zero are considered to be on the worst-case path of the program, while the maximum value of the objective function will be the WCET of the program.

In our formulation, the WCET of basic blocks are not constants. The execution time of an instruction which accesses the shared cache depends on the interferences arriving from other cores. Our objective is to distribute the interferences across the program, such that they cause the maximum increase in the execution time of shared cache-accessing instructions, and all interferences occur on the worst-case path. Consider the following ILP:

$$\text{Maximize } \sum_{i=1}^{n}(c_i y_i + \sum_{j=1}^{l_i}(e^h x_{ij}^h + e^m x_{ij}^m))$$
$$\text{subject to}$$
$$\forall i = 1, \ldots, n, \quad y_i = \sum_{j \in pred(b_i)} w_{ji} = \sum_{k \in succ(b_i)} w_{ik}$$
$$\forall i = 1, \ldots, n, \; \forall j = 1, \ldots, l_i, \quad x_{ij}^h + x_{ij}^m = y_i$$
$$\text{Loop Constraints ...}$$
$$\text{Access Constraints ...}$$
$$\text{Interference Budget Constraints ...}$$

e^h and e^m are the execution time of an instruction, in the event of a shared cache hit and shared cache miss respectively. In the objective function, we have separated out the constant portion of the execution time of a basic block (i.e $c_i y_i$), that is not affected by interferences. For each instruction accessing the shared cache, we have associated two variables (x_{ij}^h and x_{ij}^m), which will store the hit and miss counts of the instruction in the shared cache. These variables will depend on the number of interferences affecting the instruction.

Then, we have the constraints on the execution count of basic blocks and the loop constraints, which are the same as

those in the IPET formulation. Generally, $e^m > e^h$, hence, in the absence of any other constraints, to maximize the objective function, every instruction accessing the shared cache will be assumed to incur a miss. The access constraints will place an upper bound on the number of shared cache misses (i.e. x_{ij}^m) experienced by an instruction. The interference budget constraints will place an upper bound on the number of interferences for each cache set. The objective function will ensure that the interferences are distributed in such a manner that they cause the maximum number of shared cache misses. In the next few subsections, we give a detailed explanation of the access and interference budget constraints. First, we will define the concept of a hitting path of an instruction, which will allow us to link the number of interferences with its hit/miss counts.

C. Hitting Paths

We define a program path to be a sequence of instructions of the program, which follow the program order. Given an instruction a which accesses the cache block m mapped to cache set s, a path π in the program is called a **hitting path** of a if

1) π begins with instruction a' which also accesses m,
2) π ends with instruction a and has no other accesses to m other than a and a', and
3) the number of distinct cache blocks (other than m) mapped to s and accessed by instructions in π is less than the cache associativity (A).

Note that a and a' could be the same instruction. Intuitively, if the actual execution of the program reaches instruction a by flowing along a hitting path of a, then the cache block m (accessed by a) is guaranteed to be in the cache, and hence the access by a will be a cache hit. This is because after the instruction a', the cache block m will be present in the cache and will be the most recently accessed block. Now, at least A distinct cache blocks (mapped to set s) need to be brought into the cache to evict m. But since the number of distinct cache blocks accessed after a' is strictly less than A, m is guaranteed to escape eviction, and hence instruction a will be a cache hit.

Conversely, if the instruction a experiences a cache hit, then the execution must have passed through a hitting path of a. Again, since a is a cache hit, m must be in the cache, just before the execution of a. Consider the last instruction in the execution flow, that brought m to the cache. The path starting from this instruction and ending at a would be a hitting path.

If the number of distinct cache blocks accessed on path π (excluding m) is h, then the eviction distance of instruction a along π is defined to be $A - h$. The eviction distance gives minimum number of extra cache blocks required to be accessed on π to cause instruction a to be a cache miss. Thus, at least $A - h$ interfering accesses, which are mapped to set s and coming from other cores, must arrive during the execution of the hitting path π to cause instruction a to be a cache miss.

In the example in Figure 1, the hitting path of the second access to a is $a-b-a$, and its eviction distance is 1. Similarly, the hitting path of the second access to c is $c-d-c$, and its eviction distance is also 1. In general, consider an instruction a and its hitting path $\pi = a' - a_1 - \ldots - a_k - a$, where a and a' access cache block m (which is mapped to cache set s). $a_1, \ldots, a_k$ are the intervening instructions on the path, which access cache blocks mapped to set s. Since π is a hitting path, the number of distinct cache blocks accessed by the intervening instructions will be less than the cache associativity. Let p_a^π be the eviction distance. Let x_a be a binary variable, and let $z_{a_1}, \ldots, z_{a_k}, z_a$ be integer variables storing the number of interfering accesses (mapped to s) occurring just before instructions $a_1, \ldots, a_k, a$ respectively. Consider the following inequality :

$$\mathsf{p}_\mathsf{a}^\pi \mathsf{x}_\mathsf{a} \leq \mathsf{z}_{\mathsf{a}_1} + \mathsf{z}_{\mathsf{a}_2} + \ldots + \mathsf{z}_{\mathsf{a}_k} + \mathsf{z}_\mathsf{a}$$

First, note that $z_{a_1}, \ldots, z_{a_k}, z_a$ capture all the interferences which may occur on the hitting path π. If an interfering access occurs on π, it will happen after a' and before a, and thus before any of the intervening accesses. Thus, the r.h.s. of the above equation captures the total number of interfering accesses that may occur on π. If x_a is 1 in the above equation, then the total number of interfering accesses will be greater than or equal to the eviction distance. This will result in the access a becoming a cache miss. Thus we have encoded an **access constraint** which relates the integer variables storing the number of interferences with the binary variable which indicates a cache hit/miss due to interferences.

Interferences occurring before instruction a' will not have any effect on the cache hit-miss status of instruction a, and hence they can be ignored in the above equation. Note that interferences mapped to set s will not affect the contents of other cache sets. In the above formulation, we have assumed that interfering accesses arrive just before instructions which access the set s. This is safe because even if an interfering access arrives earlier, its effect will only be seen on the instructions accessing the set s.

Also, for shared caches, the hitting path of any instruction should begin with an instruction which is guaranteed to access the shared cache. In other words, the CAC of the start instruction of a hitting path must be A. The intervening instructions on the hitting path (including the instruction a) can have a CAC of A or U. If instruction a is inside a loop and has a CAC of U, we allow its hitting path to start with a.

D. Access Constraints

If an instruction is classified as AH in the shared cache (assuming no interferences), then for all paths from the start of the program to the instruction, it will experience a cache hit. In other words, every path from the start of the program to the instruction should contain a hitting path of the instruction. For the instruction to incur a cache miss due to interferences, the number of interfering accesses should exceed the eviction distance on at least one hitting path.

Let instruction j of basic block i (denoted by a_{ij}) be a cache hit in the shared cache, and let $\pi_1, \ldots, \pi_r$ be the hitting paths of this instruction. For the moment, assume that a_{ij} is not inside any loop. Let the hitting path $\pi_l = a_{i'j'} - a_{i_1 j_1} - \ldots - a_{i_k j_k} - a_{ij}$. Let $p_{ij}^{\pi_l}$ be the eviction distance along this path. Then, the access constraints for path π_l are:

$$\mathsf{p}_{\mathsf{ij}}^{\pi_\mathsf{l}} \mathsf{x}_{\mathsf{ij}}^{\pi_\mathsf{l}} \leq \mathsf{z}_{\mathsf{i_1 j_1}} + \ldots + \mathsf{z}_{\mathsf{i_k j_k}} + \mathsf{z}_{\mathsf{ij}} \quad (4)$$

$$\mathsf{x}_{\mathsf{ij}}^{\pi_\mathsf{l}} \leq \mathsf{y}_{\mathsf{i'}} \quad (5)$$

$x_{ij}^{\pi_l}$ stores the number of shared cache misses along the path π_l. Equation 4 gives the relation between $x_{ij}^{\pi_l}$ and the total number of interfering accesses on π_l, as we saw in the last subsection. Equation 5 ensures that the instruction $a_{i'j'}$ occurs on the worst case path. If $a_{i'j'}$ is not on the worst case path, the hitting path π_l itself is not on the worst case path, and hence no shared cache misses can be caused along the path. Remember that we also want to distribute the interferences only on the worst case path, and hence, we have to ensure that interferences not on the worst case path do not contribute in converting a cache hit to a cache miss. Hence, we add the following constraint for all a_{ij}:

$$\mathsf{z_{ij}} \leq \mathsf{A y_i} \tag{6}$$

If A interfering accesses arrive at a program point, then the entire shared cache will be emptied (from the perspective of core c). Hence, a valid upper bound for the number of interfering accesses at a program point is A multiplied by the number of times the program point is reached on the worst case path. The above constraint also ensures that if b_i is not on the worst case path, then no interferences will be assigned before instructions inside b_i.

For each hitting path π_l, we thus obtain an upper bound on $x_{ij}^{\pi_l}$. The access a_{ij} would become a cache miss if $x_{ij}^{\pi_l}$ is non-zero on at least one hitting path. In fact, $x_{ij}^{\pi_l}$ will be non-zero for at most one hitting path, since at most one hitting path can be on the worst case path, and we have ensured that all interferences are to be distributed on the worst case path only. The following constraint imposes the upper bound on x_{ij}^{m}:

$$\mathsf{x_{ij}^m} \leq \sum_{\mathsf{l}=1}^{\mathsf{r}} \mathsf{x_{ij}^{\pi_l}} \tag{7}$$

The above constraints are added for instructions which were classified as AH, assuming no interferences. For instruction a_{ij} which accesses the shared cache and is classified as AM or U, we add the following constraint:

$$\mathsf{x_{ij}^m} = \mathsf{y_i} \tag{8}$$

E. Handling Loops

If an instruction inside a loop is classified as AH, then in every iteration of the loop, the instruction experiences a cache hit. Whenever it experiences a cache hit, execution must have passed through a hitting path. The hitting path for the first iteration may begin outside the loop, while the hitting path for the rest of the iterations will begin within the loop itself.

The hitting path inside a loop may be traversed multiple times (e.g. in multiple iterations), but each time, the number of interferences should exceed the eviction distance of the hitting path to cause a shared cache miss. This is because in each iteration, the start instruction of the hitting path will bring the cache block to the shared cache and make it the most recently accessed block. Hence, in every iteration, the age of the accessed cache block will be reset to 1 as the execution enters the hitting path, and so every iteration requires interferences to cause a cache miss. In other words, if p_a^π is the eviction distance of instruction a along path π, and if π is inside a loop with T iterations, at least Tp_a^π interferences are required to arrive on the path π to cause a to be a cache miss on all iterations.

Equivalently, if a total of Z interferences are supposed to arrive on the path π, which is inside a loop, then these Z interferences should be distributed across iterations such that exactly p_a^π interferences will arrive each iteration, so that the number of cache misses incurred by a due to interferences will be $\lfloor Z/p_a^\pi \rfloor$. This is the maximal number of misses that Z interferences on π can cause for instruction a.

Let us now look at the access constraints defined in the previous subsection from the perspective of instructions inside loops. If a_{ij} is not inside a loop, then x_{ij}^m will be a binary variable, which will be 1 if the instruction experiences a cache miss due to interferences. This is ensured by the access constraints added for each hitting path. On the other hand, if the instruction is inside a loop, then $x_{ij}^{\pi_l}$ will give the maximal number of misses that can be caused by the assigned interferences along the hitting path π_l. This is because, according to Equation 4, $x_{ij}^{\pi_l} = \lfloor (z_{i_1 j_1} + \ldots + z_{i_k j_k} + z_{ij})/p_{ij}^{\pi_l} \rfloor = \lfloor Z/p_{ij}^{\pi_l} \rfloor$, where Z would be the total number of interferences on the hitting path. We just argued earlier that this is the maximum number of misses that Z interferences along a hitting path can cause for an instruction inside a loop.

If an instruction is not inside any loop, then at most one hitting path of the instruction will be on the worst case path. However, consider an instruction inside any arbitrarily nested loop. For each parent loop of the instruction, a hitting path originating from the parent loop could be on the worst case path. Hence, $x_{ij}^{\pi_l}$ may be non-zero for multiple hitting paths π_l. Equation 7 ensures that the contribution of each hitting path will be considered while finding the total number of misses caused due to interferences. Also, the maximum number of misses caused by a hitting path would be upper bounded by the number of times the first instruction of the hitting path is executed (i.e. the number of times the hitting path itself is executed). This is ensured by Equation 5.

We note that this method of counting interferences occurring inside loops introduces slight imprecision in the analysis. For program points inside a loop, we only keep count of the total number of interferences arriving at the program point, but the distribution of interferences across iterations is not part of the ILP. We assume that if the interferences cause misses to some instruction, then they will arrive optimally w.r.t. that instruction, so that the maximum number of misses are caused. However, the optimal distribution may be different for different instructions.

For example, suppose a total of 6 interferences occur at some program point, which is on the hitting paths of two instructions a_1 and a_2. Suppose the eviction distance of a_1 is 2, while the eviction distance of a_2 is 3. Both a_1 and a_2 are inside the same loop. Now, the optimal distribution for a_1 would be 2 interference per iteration, which will result in 3 misses for a_1. The optimal distribution for a_2 would be 3 interference per iteration, which will result in 2 misses for a_2. Hence, our ILP will count a total of 5 misses for a_1 and a_2, but this will never occur during actual execution. The feasible optimal distribution is 3 interferences for 2 iterations, resulting in a total for 4 misses (2 for a_1 and a_2). Experimentally, we found substantial precision improvement with our approach, which indicates that the above problem may not have a significant impact on precision.

F. Interference Budget Constraints

Since we know the tasks running on all the cores, we can determine the number of interferences that are generated by each core. Each instruction in the program whose CAC is A or U at the shared cache level is considered to make an interfering access to the shared cache. If such an instruction is inside a loop, then the number of interfering accesses will be the iteration count of the loop. For core c, let B_s be the total number of interferences mapped to cache set s. B_s would be the sum of interferences from every other core apart from c. Let B_s^{cb} be the number of distinct interfering cache blocks mapped to set s. Let CCB_s be the set of interfering cache blocks mapped to set s.

Let $INST_s$ be the set of instructions in the program (running on core c) whose CAC is A or U, and which access cache blocks mapped to set s. For every cache set s, the interference budget constraint is:

$$\sum_{\mathsf{a_{ij}} \in \mathsf{INST_s}} \mathsf{z_{ij}} \leq \mathsf{B_s} \tag{9}$$

Since z_{ij} will be non-zero only for instructions which are on the worst case path (ensured by Equation 6), the above constraint ensures that all B_s interferences will be distributed across instructions in $INST_s$ which are on the worst case path.

In general, the number of interfering accesses, B_s could be much larger than the number of interfering cache blocks, B_s^{cb}. While distributing interferences across the entire program, we have to use B_s, but while distributing accesses on a single hitting path, we can use B_s^{cb}. An instruction will suffer a cache miss due to interferences on a hitting path only if both the number of interfering accesses and the number of interfering cache blocks exceeds the eviction distance of the hitting path. Hence, we only add the access constraint for the hitting path π of an instruction a_{ij}, if B_s^{cb} is greater than or equal to the eviction distance p_{ij}^{π}. If $B_s^{cb} < p_{ij}^{\pi}$ for all hitting paths π of the instruction, no access constraint will be added for any hitting path, and instead we will add the constraint $x_{ij}^h = y_i$.

G. Handling Code/Data sharing

So far, we have assumed that interfering cache blocks will be different from the cache blocks accessed by core c, so that every interfering access will increase the age of all cache blocks in the shared cache. However, if programs running on different cores use shared libraries, it is possible that same cache blocks may be accessed by multiple cores. A similar scenario would occur for data caches, if programs on different cores share data variables.

We can simply ignore the sharing of code/data during our analysis, since this only affects the precision of the analysis. In the presence of sharing, interfering cache blocks may already be present in the cache, in which case they will not increase the ages of older cache blocks. Consider instruction a_{ij}, which accesses cache block m mapped to cache set s, and let π be a hitting path of the instruction. Let M^{π} be the set of cache blocks mapped to set s, and accessed by the instructions in path π. Interfering accesses which access cache blocks in $CCB_s \cap M^{\pi}$ will not contribute in increasing the age of m, since these cache blocks will also be accessed by instructions in the hitting path. Hence, we calculate the set $CCB_s \setminus M^{\pi}$, and if $|CCB_s \setminus M^{\pi}| \geq p_{ij}^{\pi}$, only then we add the access constraint for π. Again, if $|CCB_s \setminus M^{\pi}| < p_{ij}^{\pi}$ for all hitting paths π of instruction a_{ij}, then we conclude that this access can never cause a miss due to interferences, and hence add the constraint $x_{ij}^h = y_i$.

H. Proof of Safety

Given an interference Budget $\{B_s\}_{s \in \mathcal{S}}$ for each cache set and program P running on core c, we would like to show that the WCET obtained using our ILP would be greater than the execution time of any actual execution instance of P in the presence of interferences from the budget. The objective function of our ILP finds a program path such that the execution time of the program along the selected path would be maximum after assigning interferences on the path which generate the most number of misses. The ILP tries to maximize the execution time subject to the access constraints, and the access constraints give an upper bound on the number of shared cache misses due to interferences. If every feasible shared cache miss due to interferences is also allowed by the access constraints, this would mean that the ILP will maximize execution time taking into account every feasible shared cache miss. Hence, we will show that if an arrival of interferences can cause a shared cache miss during actual execution, then the same assignment of interferences in our ILP will also result in a shared cache miss.

An instruction can suffer a feasible shared cache miss only if it experiences a shared cache hit without interferences. Hence, the CHMC of the instruction without interferences should be U or AH. First, consider an instruction whose CHMC is U and which experiences a shared cache miss due to interferences during actual execution. Such instructions are already considered as shared cache misses in our ILP (Equation 8).

Now, consider an instruction a whose CHMC is AH. Suppose a experiences a shared cache miss due to interferences along the hitting path π during actual execution. Assume that π begins at instruction a'. Suppose the interferences arrive before instructions $a_1, \ldots, a_k$ on the path. If the CAC of instruction a' is A, then there will be an access constraint for π in our ILP. The CAC of instructions $a_1, \ldots, a_k$ must be either A or U, so the interference variables $z_{a_1}, \ldots, z_{a_k}$ will be part of the access constraint. If the sum of interferences exceeds the eviction distance during actual execution, then the same assignment of interferences to $z_{a_1}, \ldots, z_{a_k}$ will also exceed the eviction distance in the access constraint. Thus, the shared cache miss will be allowed and will be taken into consideration by the ILP.

If the CAC of a' is U, then there will be hitting path π' such that π is a sub-path of π'. This is because a is classified as AH, and hence there must be some instruction a'' before a which accesses the cache block accessed by a and has a CAC of A. The hitting path starting from a'' (call it π') will contain the accesses $a_1, \ldots, a_k$. Hence, $z_{a_1}, \ldots, z_{a_k}$ will be part of the access constraint for π', while $p^{\pi'} \leq p^{\pi}$, which will allow the shared cache miss for the same assignment of interferences.

VI. Experimental Results

We have implemented our technique of optimal interference placement on top of the Chronos [3] WCET analyzer. Chronos is an open-source WCET analyzer which supports instruction and data cache analysis for a 2-level cache hierarchy using the standard AI-based approach. We extended Chronos by adding support for multi-core shared instruction cache analysis. We implemented two techniques for performing shared cache analysis : (1) Hardy et al.'s [1] approach, which assumes that all interfering accesses to the shared cache occur at all program points and (2) Our approach, which solves an ILP to obtain the optimal placement of interferences, causing maximum increase in execution time. As stated earlier, most of the current approaches for shared cache analysis assume all interferences at all program points. Hence, we compare the precision of WCET obtained using our approach and Hardy et al.'s approach. We use the open source program lp_solve to solve the generated ILPs. Our experiments were conducted on a 4-core Intel i5 CPU with 4 GB memory.

To generate the ILP, we have to determine the access constraints for each instruction which is classified as AH. This requires finding all the hitting paths of such an instruction. To determine the hitting paths, we use a simple breadth first search (BFS) of the program CFG in the reverse direction starting from the instruction. We continue the search until we find another instruction which accesses the same cache block and has a CAC of A. We keep track of the instructions which access the same cache set on each such path returned by the BFS. If an instruction is inside a loop, then during the BFS, we only allow one traversal of the back edge of any loop. Hence, each hitting path can contain at most one instance of a program point. This is safe because we only keep track of the total number of interferences at each program point in our ILP, and not its distribution across iterations. Hence, even though an actual hitting path may pass through a program point multiple times (e.g. in case of a nested inner loop), its effect will be captured in our ILP.

For the experiment, we assume a 2-core architecture with a shared L2 instruction cache. Our cache architecture is: 1-KB 4-way L1 cache with block size 32 bytes, and a 4-KB 8-way L2 cache with block size 32 bytes. We assume that a L1 hit takes 1 cycle, a L2 Hit takes 6 cycles, and memory access takes 30 cycles. We use 27 benchmarks from the Mälardalen WCET benchmark suite [17]. The L2 cache performance of the benchmarks is crucial in determining the impact of shared cache analysis on WCET. Clearly, more L2 accesses and L2 hits would mean more reliance on the precision of shared cache classification. Table II gives the percentage of L2 accesses (L2 accesses/Total accesses) and percentage of L2 Hits (L2 hits/L2 accesses) for all benchmarks for the above cache architecture, assuming no interferences to the shared cache. These results are obtained using the single-core cache analysis of the benchmarks. Hence, L2 hit percentage considers only guaranteed L2 hits, while L2 access percentage considers all possible L2 accesses.

To compute WCET of a benchmark on a 2-core architecture, we assume that the benchmark runs on one core and the worst-case adversary (i.e. the benchmark which causes the maximum number of shared cache misses due to interferences) runs on the other core. Table III shows the WCET assuming no interference (Column 2), WCET assuming all interferences at all program points (Column 3), WCET assuming optimal interference placement (Column 4). Column 5 shows the absolute precision improvement of OPT-INTERF over ALL-INTERF (computed as $\frac{\mathsf{WCET}_{\mathsf{ALL}}-\mathsf{WCET}_{\mathsf{OPT}}}{\mathsf{WCET}_{\mathsf{ALL}}}$). Column 6 shows the relative precision improvement, which is defined as $\frac{\mathsf{WCET}_{\mathsf{ALL}}-\mathsf{WCET}_{\mathsf{OPT}}}{\mathsf{WCET}_{\mathsf{ALL}}-\mathsf{WCET}_{\mathsf{NO}}}$. It represents the precision improvement of OPT-INTERF over ALL-INTERF as a percentage of the maximum possible improvement. The relative precision improvement is useful for benchmarks whose absolute precision improvement is low because of smaller number of L2 accesses.

TABLE II. L2 performance of Benchmarks with no interference

Benchmark	% of L2 accesses	% of L2 hits	Benchmark	% of L2 accesses	% of L2 hits
adpcm	10.02	99.78	insertsort	0.0002	8.33
bs	8.91	0.00	jfdctint	14.74	99.94
bsort100	0.01	98.94	lms	1.47	74.78
cnt	0.01	7.14	matmult	0.01	0.00
cover	5.72	87.77	minver	28.25	0.13
crc	0.34	2.96	ndes	8.89	18.36
duff	2.41	4.55	ns	0.25	6.67
edn	0.00	0.00	nsichneu	26.13	0.04
expint	16.10	97.89	prime	6.37	98.17
fdct	2.65	1.30	qsort-exam	40.20	0.00
fft	2.87	2.12	qurt	29.75	14.99
fir	0.07	6.67	sqrt	6.57	58.33
st	0.06	21.92	statemate	34.05	0.66
ud	6.31	1.29			

TABLE III. Comparison of WCET obtained using our approach and Hardy et al.'s approach

Benchmark	WCET (No-Interf)	WCET (All-Interf)	WCET (Opt-Interf)	% Abs. improv.	% Rel. improv.
adpcm	255153	599097	257082	57.08	99.44
expint	6543	20847	10093	52.49	75.18
lms	647803	795115	650659	18.16	98.06
cover	3172	6100	3724	38.95	81.15
prime	31160	65912	31904	51.06	97.81
bsortl00	7360848	7385448	7361136	0.38	98.82
fft	62827	63331	63235	0.15	19.04
ndes	138677	159965	147197	8	60
qurt	13677	15501	14445	6.8	57.9
sqrt	1037	1541	1229	20.24	61.9
jfdctint (43%)	1721591	5221751	3734351	28.61	42.5

The results in Table III are shown for the 11 benchmarks for which we found precision improvement because of optimal interference placement. For the rest of the benchmarks, the WCET obtained using OPT-INTERF and ALL-INTERF were almost the same. We note that in those cases, WCET obtained assuming no interference was also the same as WCET obtained using ALL-INTERF. Hence, the reason for no precision improvement was because of the L2 performance of those benchmarks.

The benchmarks in Table III are exactly those benchmarks which exhibit substantial number of L2 accesses and L2

TABLE IV. RESULTS OBTAINED BY HALVING THE L2 CACHE

Benchmark	% of L2 hits	% Abs. improv.	% Rel. improv.
adpcm	49.89	24.34	98.92
expint	13.92	10.01	87.88
prime	48.75	25.74	98.33
ndes	15.32	1.45	13.10
jfdctint (43%)	85.96	27.95	49.67

TABLE V. RESULTS OBTAINED BY HALVING THE L1 CACHE

Benchmark	% of L2 accesses	% of L2 hits	% Abs. improv.	% Rel. improv.
fdct (50%)	22.27	88.34	17.39	27.43
fir	2.27	97.33	30.25	93.53
ud	10.55	17.88	1.44	12.90
insertsort	0.06	99.64	1.37	99.28

hits (as shown in Table II). The massive absolute precision improvement in benchmarks such as $adpcm, expint, prime$ corresponds to the almost perfect L2 behavior of these benchmarks. Similarly, $bsort100$ and fft show very low absolute improvement, but these benchmarks hardly access the shared L2 cache. Moreover, all benchmarks (except fft) show very high relative precision improvement, which means OPT-INTERF is able to achieve a high percentage of the total improvement possible. For almost all the benchmarks, the worst-case adversary was either $nsichneu$ or $statemate$, as these benchmarks perform a large number of L2 accesses, and also access a large number of distinct cache blocks in L2 (because of which their own L2 hit percentage is very low).

The average absolute precision improvement over the 11 benchmarks is 25.63 %, while the average relative precision improvement is 72%. Except $jfdctint$, the analysis time (including time for solving the ILP) for all the benchmarks was reasonably small (~few seconds). However, the ILP solver was not able to solve the ILP for $jfdctint$ in any reasonable duration of time. $jfdctint$ has approximately 150K L2 hits, which are distributed across 4 loops. To reduce the analysis time, we picked 1 loop containing approximately 43% of the L2 hits, and performed optimal interference placement for the L2 hits inside the loop. For the rest of the program, we assumed all interferences at all program points. With such a selective strategy, the ILP solver was able to solve the ILP in few seconds and the absolute precision improvement of the WCET so obtained was 28.61 %. The average time required to compute the WCET using our approach is 4.79s, while the average computation time of Hardy et al.'s approach is 0.2s.

Changing L1, L2 cache sizes : Since some of the benchmarks showed almost perfect L2 behavior, we decided to decrease the L2 cache size and then measure the precision improvement. Keeping the same L1 cache size, we changed L2 to be a 2-KB 8-way cache with block size 32 bytes (half of the original L2 size). As a result, the L2 hit percentage of 6 of the 11 benchmarks in Table III became 0. Hence, these benchmarks did not show any precision improvement using our approach. The L2 hit percentage of the rest of the benchmarks, along with the precision improvement in their WCET, are shown in Table IV.

The results indicate that as the L2 hit percentage decreases, the absolute improvement in precision of WCET also decreases, which is as expected. The relative improvement still remains quite high, which indicates that our approach is still able to achieve a high percentage of the total precision improvement possible. The average absolute precision improvement over the 5 benchmarks is 15.02 %. Again, for the $jfdctint$ benchmark, we had to limit optimal interference placement to 1 loop.

We also changed L1 to be a 0.5-KB 4-way cache with block size 32 bytes (half of the original L1 size), and restored L2 to its original size. As a result, we found precision improvement in the WCET of 4 more benchmarks (apart from those in Table III). The precision improvement along with the L2 performance of these benchmarks is listed in Table V.

Again, the precision improvement is correlated with the increased L2 access (and hit) percentage of the benchmarks, while the benchmarks which do not show any precision improvement (not shown in the table) continued to have almost zero L2 access percentage for the new cache configuration as well. Note that the benchmarks which showed precision improvement with the original cache configuration continued to do so with new cache configuration. For the benchmark $fdct$, the original ILP was not solved in a reasonable amount of time, so we selected 8 out of the 16 cache sets and limited optimal interference placement to L2 hits mapping to those 8 cache sets.

Scalability : The scalability of our approach depends on the hardness of the generated ILP, which in turn depends on the number of L2 hits to be analyzed, the number of hitting paths, the number of interferences to be distributed, etc. In particular, we found during our experiments that the number of distinct L2 hits in the program to be analyzed significantly effected the complexity of the ILP (both $jfdctint$ and $fdct$ had large number of distinct L2 cache hits). There is a trade off between scalability and precision of the WCET, and to achieve scalability we propose the following strategies to simplify the ILP problem.

In *Cache Set Selection (CSS)*, we perform optimal interference placement for interferences belonging to selected cache sets only. The interferences belonging to the remaining cache sets will be assumed to occur at all program points. This strategy not only reduces the number of interferences to be distributed, but also reduces the number of L2 hits to be analyzed, as L2 hits accessing the un-optimized cache sets will already be classified as hit/miss before the ILP formulation. We have used this strategy in our experiments for $fdct$, resulting in improved precision of the WCET in a reasonable duration of time.

In *Program Area Selection (PAS)*, optimal interference placement is carried out over a selected program segment. It is assumed that all interferences occur at all program points for the rest of the program. For example, loops running for large number of iterations and containing L2 hits would be an ideal candidate for optimal interference placement, as the number of distinct L2 hits would be small, and the interference budget may limit the L2 misses to a portion of the iterations. This strategy could be useful for large programs containing many loops. We applied this strategy to the $jfdctint$ benchmark for

the original cache configuration. $jfdctint$ has 4 loops, out of which we selected 1 loop for optimal interference placement.

In *Interference Selection (IS)*, if an interfering cache block is accessed a large number of times (by the other cores), then it is assumed that the interfering access to the cache block will occur at all program points. This will reduce the number of interfering accesses to be distributed.

Different approaches may work for different types of programs, and the selection also depends of the type of the adversary programs (i.e. the programs running on other cores). For example, PAS gave better precision than CSS for $jfdctint$, while for $fdct$, CSS was more beneficial. For each approach, there are choices which may provide better precision in smaller amount of analysis time. For example, in CSS, there is a choice of which cache sets should be selected for optimal interference placement. Heuristics such as higher number of L2 hits, smaller number of distinct L2 hits, etc. can be used to select appropriate cache sets. Similarly, in PAS, heuristics can be used to identify program segments which will benefit from optimal interference placement. We note that in our experiments, while using CSS, we picked cache sets which had higher number of L2 hits mapped to them. We found that this approach provided better precision than any arbitrary selection of cache sets. Similarly for PAS, we selected loops with higher number of L2 hits.

Finally, note that even though in our experiments, we have used our ILP formulation to directly find the WCET, it can also be used to find a safe hit-miss classification of cache accesses in the presence of interferences. This hit-miss classification can later be taken as input while performing other architectural analyses.

VII. 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. Current approaches for WCET estimation are highly imprecise, and most of the imprecision stems from the shared cache analysis, which assumes that all interferences from other cores can arrive at all program points. In our work, we propose a new approach for shared cache analysis, where we assume that interferences from other cores will arrive at the worst possible program points, causing maximum number of shared cache misses. We build an ILP to locate the worst possible program points and to optimally distribute the interferences across them. To build the ILP, we find the hitting paths for each shared cache hit, which are paths in the program where the arriving interferences can cause damage. We specify the minimum number of interferences required for a shared cache miss, and then leave the optimal distribution of interferences to the ILP.

We have implemented our approach in Chronos, an open-source WCET analyzer, and compared the WCETs obtained using our approach and earlier approaches for a large number of benchmarks. For all benchmarks which make sufficient number of shared cache accesses, our approach is significantly more precise than earlier approaches. We also test our approach for different cache configurations, and show that it continues to give better precision. We discuss the scalability issues related to our approach, and give a number of techniques to decrease the analysis time at the cost of lower precision.

The nature of the problem of optimal interference placement suggests that there could be algorithmic approaches to solve the problem. Such approaches may scale better over larger programs. Also, our approach has the property that smaller number of interferences will lead to lower WCET estimates. Hence, there is scope for building new scheduling algorithms for real-time systems which could guarantee low cache interference between tasks scheduled on different cores. This could also lead to a relaxed schedulability test, allowing more task-sets to be scheduled.

VIII. Acknowledgements

This work was supported by Microsoft Corporation and Microsoft Research India under the Microsoft Research India PhD Fellowship Award. The authors would also like to acknowledge the support of the IMPECS project.

References

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

[2] Yan-Tsun Steven Lee and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, 1995.

[3] 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.

[4] Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared l2 instruction caches. In *RTAS*, 2008.

[5] Jun Yan and Wei Zhang. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *RTCSA*, 2009.

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

[7] Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudhury, Timon Kelter, Marwedel Peter, and Falk Heiko. A unified WCET analysis framework for multi-core platforms. In *RTAS*, 2012.

[8] Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *RTSS*, 2011.

[9] Vivy Suhendra and Tulika Mitra. Exploring locking and partitioning for predictable shared caches on multi-cores. In *DAC*, 2008.

[10] Marco Paolieri, Eduardo Quiñones, Franciso J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.

[11] 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 *RTSS*, 2011.

[12] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS*, 2013.

[13] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *EMSOFT*, 2012.

[14] Damien Hardy and Isabelle Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.

[15] Yan-Tsun Steven Lee, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *RTSS*, 1995.

[16] Yan-Tsun Steven Lee, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *RTSS*, 1996.

[17] WCET projects / benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.