Scalable and Precise Refinement of Cache Timing Analysis via Model Checking

Sudipta Chattopadhyay Abhik Roychoudhury
National University of Singapore
{sudiptac,abhik}@comp.nus.edu.sg

***Abstract*—Hard real time systems require absolute guarantees in their execution times. Worst case execution time (WCET) of a program has therefore become an important problem to address. However, performance enhancing features of a processor (*e.g.* cache) make WCET analysis a difficult problem. In this paper, we propose a novel approach of combining abstract interpretation and model checking for different varieties of cache analysis ranging from single to multi-core platforms. Our modeling is used to develop a precise yet scalable timing analysis method on top of the Chronos WCET analysis tool. Experimental results demonstrate that we can obtain significant improvement in precision with reasonable analysis time overhead.**

I. Introduction

Worst-case execution time (WCET) analysis of real-time embedded software is an important problem. WCET estimates of tasks are used for system level schedulability analysis. WCET estimation usually involves a program level path analysis (to determine the infeasible paths in the program's control flow graph), micro-architectural modeling (to accurately determine the maximum execution time of the basic blocks), and a calculation phase (which combines the results of path analysis and micro-architectural modeling).

Micro-architectural modeling usually involves systematically considering the timing effects of performance enhancing processor features such as pipeline and caches. Cache analysis for real-time systems is usually accomplished by abstract interpretation. This involves estimating the cache behavior of a basic block B by considering the incoming flows to B in the control flow graph. The memory accesses of the incoming flows are analyzed to determine the cache hits/misses for the memory accesses in B. Since programs contain loops, such an analysis of memory accesses involves an iterative fixed point computation via a method known as abstract interpretation. Abstract interpretation is usually efficient, but the results are often not precise. This is because the estimation of memory access behavior are "joined" at the control flow merge points - resulting in an over-estimation of potential cache misses returned by the method.

In this paper, we develop a cache analysis framework which improves the precision of abstract interpretation, without appreciable loss of efficiency. We augment abstract interpretation with a gradual and controlled use of model checking, a path sensitive search based formal verification method. Because of path sensitivity in its search - model checking is known to be of high complexity. Hence abstract interpretation based analysis cannot be naively replaced with model checking for analysis of cache behavior. Recent works [1] which have advocated combination of abstract interpretation and model checking for multicore software analysis - restrict the use of model checking to program path level; cache analysis is still accomplished only by abstract interpretation. Indeed almost all current state-of-the-art WCET analyzers (such as Chronos [2], aiT [3]) perform cache analysis via some variant of abstract interpretation. Model checking is usually found to be not scalable for micro-architectural analysis because of the huge search space that needs to be traversed. The main novelty of our work lies in integrating model checking with abstract interpretation for timing analysis of cache behavior.

Our baseline analysis is abstract interpretation. Potential cache conflicts identified by abstract interpretation are then subjected to model checking. Our goal is to rule out "false" cache conflicts which can occur only on infeasible program paths. Such false conflicts are considered by abstract interpretation since its join operator (which merges the estimates from paths at control flow join points) conservatively considers all possible cache conflicts on any path in the control flow graph. The path sensitive search in model checking naturally rules out the infeasible program paths and the cache conflicts incurred therein.

One appealing nature of our analysis method is that the results are always safe. We start with the results from abstract interpretation and gradually refine the results with repeated runs of model checking. Model checking is a property verification method which takes in a system/program P and a temporal logic property φ, where φ is interpreted over the execution traces[1] of P. It checks whether all execution traces of P satisfy φ. Given a potentially conflicting pair of memory blocks, we can model check a property that the pair never conflicts in any execution trace of the program. If indeed the conflict pair is introduced due to the over-approximation in abstract interpretation - model checking verifies that the conflict pair can never be realized. We can then rule out the cache misses estimated due to the conflict pair and tighten the estimated time bounds.

The property checked in a single run of model checking involves certain cache conflicts identified by abstract

[1]We consider only Linear Time Temporal Logic properties here.

interpretation - model checking then verifies whether these conflicts are indeed realizable. Thus, the scalability of our framework is never in question. Given a time budget T, we can first employ abstract interpretation and then employ as many runs of model checking as we can within time T. Of course, given more time, the results are more precise.

Contributions: In summary, this paper presents a generic cache analysis framework based on abstract interpretation and model checking. Depending on the time budget for analysis and the analysis precision required - the framework can be tuned to analyze cache hit/miss classifications for timing analysis. We further show that the framework can be instantiated with a wide variety of cache analyses - (i) analysis of cache behavior in a single program, (ii) analysis of cache related preemption delay for a multi-tasking system where the tasks are running on a single core, and (iii) analysis of shared caches in multi-cores. Our experimental results on the moderate to large scale WCET benchmarks [4] show substantial improvement in the precision of timing analysis results with limited time overheads. This yields a parameterizable cache analysis framework for real-time systems which is generic, precise and scalable.

II. Related work

Since the initiation of WCET analysis research, cache modeling has been an active topic in this area. Initial works used Integer Linear Programming (ILP) [5] for modeling intra-task cache conflicts. However, ILP based approach for cache modeling faces scalability concerns in terms of analysis time. Subsequently, a novel WCET analysis approach has been proposed in [6], which efficiently composes abstract interpretation based micro-architectural modeling and ILP based path analysis. The solution proposed in [6] has been proved scalable and it has also been adopted in commercial tool chain ([3]).

In multi-tasking system, additional difficulties arise in modeling inter-task cache conflicts. Inter-task cache conflicts are generated by a high priority task when it preempts a low priority task. The bound on additional cache misses due to preemption is called *cache related preemption delay* (CRPD). In last decade, there has been an extensive amount of research to estimate CRPD [7], [8], [9] using abstract interpretation. Recently, two advancements in CRPD estimation ([10] and [11]) have improved and generalized the previously proposed approaches for set-associative caches.

With the extensive deployment of multi-core architectures, it has also become important to adopt the existing cache analysis techniques for multi-cores. Multi core architectures employ shared resources (*e.g.* shared cache). Therefore, a few research groups have already proposed the modeling of shared cache ([12], [13] and [14]) based on abstract interpretation.

In [15], it is argued that model checking alone is not suitable for WCET analysis due to the state space explosion problem. On the other hand, [16] uses model checking alone for cache and path analysis. However, [16] does not employ the modeling of other important micro-architectural features (*e.g.* pipeline) and it is unclear whether the employed technique would remain scalable in presence of pipeline or other micro-architectural features. In contrast, our technique can easily be integrated with the modeling of different micro-architectural features (*e.g.* pipeline). Nevertheless, some recent advances [17] have employed full model checking based approach for software timing analysis in pipelined processor. However, [17] faces some common scalability issues (*e.g.* state space explosion) in presence of caches.

In summary, abstract interpretation based approach is scalable for cache analysis and it is easy to integrate with other micro-architectural features (*e.g.* pipeline). On the other hand, model checking can give the most accurate result, but it is difficult to scale in terms of analysis time. A recent approach [1] has therefore looked at the combination of abstract interpretation and model checking. However, [1] uses model checking for path analysis only; cache analysis is performed by conventional abstract interpretation methods. In this paper, we study the combination of model checking and abstract interpretation for different cache analysis to design a scalable and precise WCET analysis framework. Our analysis can be stopped after any number of model checking steps and the results are always safe. Thus our framework gives the designer a precision-scalability tradeoff which (s)he can choose to use.

III. Background

WCET analysis of a single task: WCET analysis of a single task is broadly composed of two different phases: i) micro-architectural modeling and ii) path analysis. Micro-architectural modeling analyzes the timing characteristics of different hardware components (*e.g.* cache, pipeline, branch predictor) and works at the granularity of basic blocks. As an outcome of micro-architectural modeling, we obtain the WCET of each basic block in the examined program. On the other hand, path analysis uses the WCET of each basic block as input and searches for the *longest feasible program path*. Our baseline implementation employs the separated cache and path analysis as proposed in [6]. [6] uses abstract interpretation (AI) for cache analysis and integer linear programming (ILP) for path analysis. We assume *least recently used* (LRU) cache replacement policy. We implement *must* and *may* cache analysis to classify memory blocks as *all-hit* (AH) and *all-miss* (AM) respectively. *Must* analysis is used along with virtual inline and virtual unrolling (VIVU) as discussed in [6]. In VIVU approach, each loop is unrolled once to distinguish the *cold cache misses* at first iteration of the loop. AH categorized memory blocks are always in cache when accessed. On the other hand, AM categorized memory blocks are never in cache when accessed. If a memory block cannot be classified as either of two (AH

or AM), it is considered *unclassified* (NC). Cache analysis outcome is used for computing the WCET of each basic block. Finally, longest path search in a program is formulated as an integer linear program. The formulated ILP uses the basic block WCETs and structural constraints imposed by program control flow graph (CFG). Infeasible program path informations are also encoded as separate ILP constraints using the technique explored in [18]. The solution of the formulated ILP returns the whole program WCET.

Inter-task cache conflict analysis: Inter-task cache conflict analysis is required to find an upper bound on cache misses due to preemption. The bound on cache misses (or additional clock cycles) due to preemption is called cache related preemption delay (CRPD). CRPD analysis revolves around the notion of two basic concepts: *useful cache blocks* (UCB) and *evicting cache blocks* (ECB). UCBs are computed by analyzing the preempted task and ECBs are computed by analyzing the preempting task. A UCB is a block that may be *cached* before preemption and may be *used* later, resulting in a cache hit in the absence of preemption. The number of UCBs imposes a bound on CRPD. On the other hand, the preempting task can cause additional cache misses in a cache set only if it uses the same cache set during its execution. For a particular cache set, the set of cache blocks used by the preempting task during its execution is known as ECB for the corresponding cache set. ECBs are used to check whether a particular UCB could be *evicted* by the preempting task. We can disregard all UCBs from CRPD computation if they could not be evicted by the set of computed ECBs. Therefore, a combined analysis with UCB and ECB may tighten the CRPD estimates obtained with UCB alone [8]. Recently, two approaches ([10] and [11]) have improved and generalized the state-of-the-art CRPD estimation framework ([7], [8]) for set associative caches. We implement both the techniques proposed in [10] and [11] in our baseline CRPD estimation framework. Therefore, our baseline implementation captures the current state-of-the-art AI-based CRPD analysis.

Inter-core cache conflict analysis: Inter-core cache conflict analysis computes the conflicts generated in shared cache. Conflicts in shared cache, on the other hand, are generated by the tasks running on different cores. Till now, only a few solutions have been proposed for analyzing timing behaviors of shared cache [12], [14], [13]. However, all of them suffer from over-estimating the inter-core cache conflicts. We use our former work on shared cache analysis [12], which employs a separate shared cache conflict analysis phase. Shared cache conflict analysis may change the categorization of a memory block m from all-hit (AH) to unclassified (NC). This analysis phase first computes the number of unique conflicting shared cache accesses from different cores. Then it is checked whether the number of conflicts from different cores can potentially replace m from shared cache. More precisely, cache hit/miss categorization

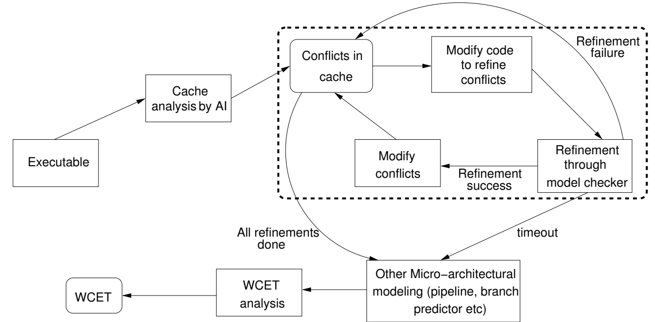

Figure 1. General framework of our WCET analysis which combines abstract interpretation and model checking

(CHMC) of m is changed from all-hit (AH) to unclassified (NC) if and only if the following condition holds:

$$N - age(m) < |\mathcal{M}_c(m)| \tag{1}$$

where $|\mathcal{M}_c(m)|$ represents the number of conflicting memory blocks from different cores which may potentially access the same L2 cache set as m. N represents the associativity of shared L2 cache and $age(m)$ represents the *age* of memory block m in shared L2 cache set in the absence of inter-core conflicts. Therefore, $N - age(m)$ specifically represents the amount of shift that memory block m can tolerate before being replaced from the cache. We call the term $N - age(m)$ as *residual age* of m.

IV. Analysis framework

A. General framework

Figure 1 demonstrates the general analysis framework. Our goal is to refine different types of abstract interpretation (AI) based cache analysis through model checking (MC). *Cold cache misses* are unavoidable and AI based cache analysis can accurately predict the set of cold cache misses. However, AI based cache analysis suffers from overestimating the *conflict misses* in a cache. On the other hand, conflicts in a particular cache set may come from different sources. We focus on all three types of conflicts which may arise in a cache: first, intra-task cache conflicts which is created by different memory blocks accessed by a particular task and mapping into the same cache set. Secondly, inter-task cache conflicts which is created when a high priority task preempts a low priority task. Finally, inter-core cache conflicts which is generated in the shared cache by a task running on a different core. Figure 2 pictorially represents all forms of above mentioned cache conflicts.

Even though the basic goal of our framework is cache conflict refinement, the notion of cache conflict may vary depending on the outcome of AI based cache analysis. For example, in inter-task cache conflict refinement, initial CRPD analysis produces a set of ECBs, which can be considered as the set of cache conflicts. On the other hand, during intra-task and inter-core cache conflict refinement, we get the cache hit miss classification (AH, AM or NC) of

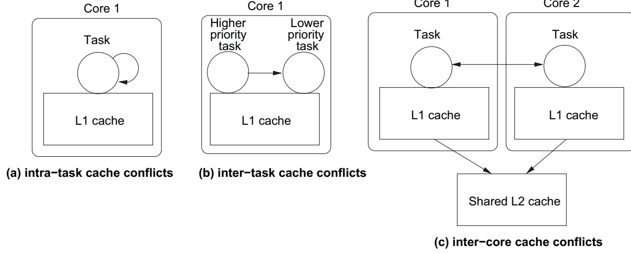

Figure 2. Variants of cache conflicts

each memory block. A memory block might be categorized as NC due to its conflicts with more than one memory block. Therefore, by refining one NC categorized memory block into AH, we may reduce more than one cache conflict pairs, which may in turn result in an improvement of WCET.

In Figure 1, the dotted boxed portion has different implementations for refining different types of cache conflicts (*i.e.* intra-task, inter-task and inter-core). The refinement of cache conflicts is iteratively performed through model checking on a modified program. We rule out the cache accesses for which AI has generated precise information. Therefore, the model checker refinement phase works on a very small subset of all cache accesses. The iterative refinement through model checking eliminates several infeasible paths from the candidate program, resulting in the removal of several unnecessary conflicts generated in a particular cache set. The iterative refinement is continued as long as the time budget permits or all possible refinements have been performed by MC. Recall that the WCET analysis process can broadly be categorized into two phases: micro-architectural modeling and path analysis. The infeasible path exploration by the model checker is only performed for refining cache conflicts (*i.e.* during the micro-architectural modeling phase). For path analysis, our framework encodes the infeasible path informations as separate ILP constraints (for details, refer to [18]). Infeasible path constraints are finally used in the global ILP formulation for computing WCET. There are two important advantages of our framework: first, the iterative MC refinement can be terminated at any point if the time budget exceeds. The resulting *cache conflicts*, after a partial refinement, can *safely* be used for estimating the WCET or CRPD. Secondly, our framework can be composed with other micro-architectural features (*e.g.* pipeline, branch prediction) and thereby, not affecting the flexibility of AI-based cache analysis.

A general code transformation framework: Any code transformation for refining various cache conflicts can be represented by a quintuple $< \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} >$ as follows:

- $\mathcal{L}$: Set of conflicting memory blocks in the cache set for which the refinement is being made.
- $\mathcal{A}$: The property which need be checked by the model checker. The property is placed in form of an "assertion" clause, which validates $\mathcal{A}$ for all possible execution traces of the modified code.
- $\mathcal{P}_l$: Set of positions in the code where the conflict count would be incremented. These are the set of positions where some memory block in $\mathcal{L}$ might be accessed.
- $\mathcal{P}_c$: Position in the code where property $\mathcal{A}$ would be placed.
- $\mathcal{I}$: Set of positions in the code to reset conflict count. Recall that we consider LRU cache replacement policy. A memory block m becomes the *most recently used* immediately after it is accessed. Therefore, if we are counting cache conflicts with m, the conflict count must be reset after m is accessed.

Any model checker refinement pass corresponds to a specific cache set and therefore, conflicts are defined for a specific cache set in each code transformation. Consequently, computation of $\mathcal{L}$ and $\mathcal{P}_l$ depends only on the cache set for which the conflicts are being refined. On the other hand, $\mathcal{A}$, $\mathcal{P}_c$ and $\mathcal{I}$ depends on the type of cache conflict (*i.e.* intra-task, inter-task or inter-core) being refined.

In subsequent sections, we shall describe the instantiation of the framework in Figure 1 for refining different versions of cache conflicts (as shown in Figure 2). We shall also show how $\mathcal{A}$, $\mathcal{P}_c$ and $\mathcal{I}$ are configured depending on the type of cache conflict being refined.

V. Refinement of intra-task cache conflicts

In this section we describe the refinement of cache conflicts shown in Figure 2(a). Recall that the memory blocks are classified as AH (*all-hit*), AM (*all-miss*) or NC (*unclassified*) by [6]. AH and AM are guaranteed categorizations by AI based cache analysis. Therefore, AH and AM categorized memory blocks do not have any scope for refinement. On the other hand, AI based cache analysis fails to give guaranteed information (in this case cache hit or cache miss) for NC categorized memory blocks. Consequently, we use the model checker to refine the set of NC categorized memory blocks.

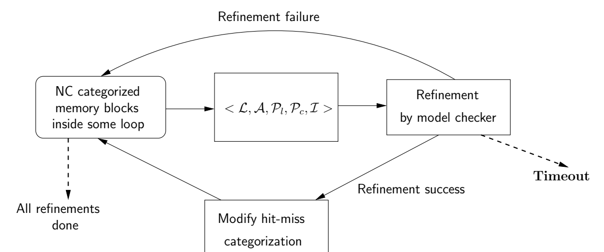

Figure 4. Refinement of intra-task conflict analysis

Figure 4 demonstrates the instantiation of our general framework for reducing the over-estimation in intra-core WCET analysis. As shown in Figure 4, we only target the NC categorized memory blocks inside some loop. Therefore, we concentrate only on a few memory blocks whose successful refinement may lead to a reasonable WCET improvement. For each of the NC categorized memory blocks

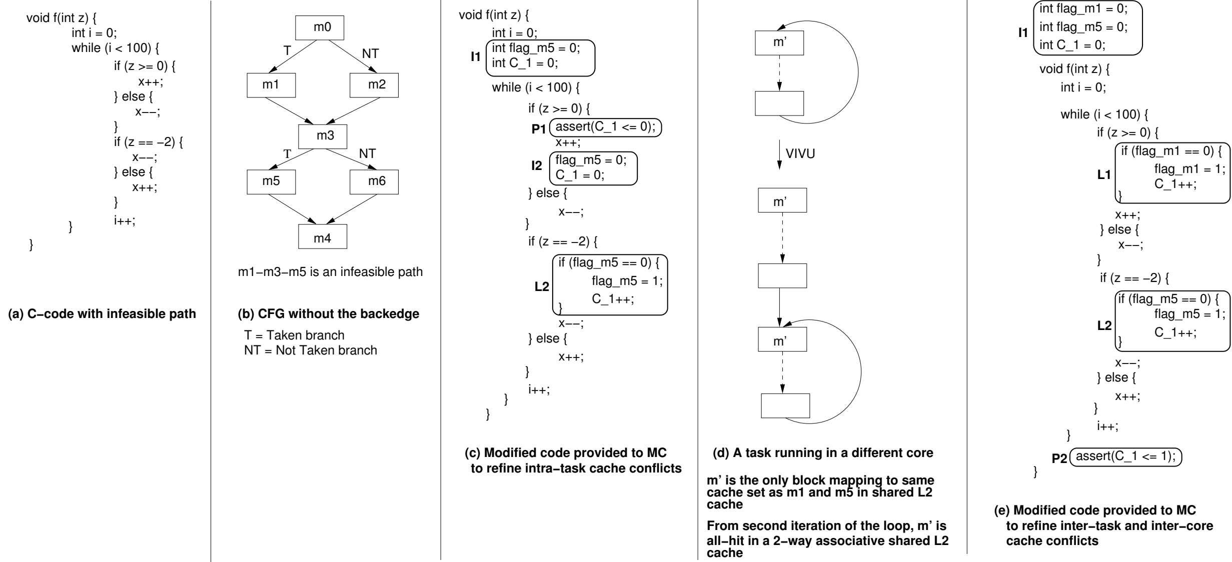

Figure 3. Refinement of various cache conflicts

under consideration, we call our general code transformation framework (as shown in Figure 4). Let us assume that we want to refine the categorization of a particular memory reference m mapping to a cache set i. m becomes the *most recently used* cache block immediately after it is accessed. We would like to check whether m could be evicted from the cache between any two of its consecutive references. Let us assume C_i counts the number of unique conflicts in cache set i and N is the associativity of the cache. Since we use LRU cache replacement policy, it would require at least N unique conflicts to replace m (from cache set i) after m is referenced. Therefore, if C_i is less than or equal to $N-1$, we can guarantee that m cannot be *evicted* from the cache. The model checker is used to check an "assertion" property $C_i \leq N-1$ just before m is referenced. More over, as m is the most recently used cache block immediately after it is accessed, we need to reset the conflict count C_i after m is referenced.

We demonstrate our technique through an example in Figure 3(a). Parameter z can be considered as a user input. Corresponding control flow graph (CFG) of the loop body and the accessed memory blocks are shown in Figure 3(b). For illustration purposes, assume a direct-mapped L1 cache where $m1$ and $m5$ are mapped to the same cache set and rest of the memory blocks do not conflict in L1 cache with $m1$ or $m5$. A correct AI-based cache analysis will classify both $m1$ and $m5$ accesses as NC. Figure 3(c) shows the transformation to refine the NC categorization of $m1$. Since the cache is direct mapped, the refinement of $m1$ is possible only if there are no other conflicting cache accesses between any two consecutive accesses of $m1$. Variable C_1 serves the purpose of counting the number of conflicts. Since $m5$ is the only conflicting memory block, C_1 is incremented before the access of $m5$. Increment of C_1 is guarded by condition ($flag_m5$ serves the purpose of guard), so that we count only unique cache conflicts. The above transformation of code is *fully automated* and we pass the transformed code to a software model checker. As $m1$-$m3$-$m5$ is an infeasible path (due to the conflicting conditions $z \geq 0$ and $z = -2$), a software model checker satisfies the assertion clause "P1" in Figure 3(c). Therefore, we conclude that $m1$ cannot be evicted from cache. Similarly, using a separate model checker refinement pass, we can also conclude that $m5$ cannot be evicted from cache. As marked in Figure 3(c), our code transformation framework $< \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} >$ is configured as follows: $\mathcal{L} = \{m5\}$, $\mathcal{A}$ is the "assertion" clause checking the property $C_1 \leq 0$, $\mathcal{P}_l = \{L2\}$, $\mathcal{P}_c = \{P1\}$ and $\mathcal{I} = \{I1, I2\}$.

VI. Refinement of inter-task cache conflicts

We now show the refinement of inter-task cache conflicts (as shown in Figure 3(b)) and thereby reduce the over-estimation introduced by CRPD analysis. A major source of over-estimation in CRPD analysis may come from the computation of evicting cache blocks (ECB). ECB denotes the set of cache blocks possibly touched by the preempting task. Recall that CRPD depends on the set of useful cache blocks (UCBs) replaced from cache due to preemption. Whether a UCB could be replaced due to preemption, on the other hand, depends on the set of ECBs conflicting with it. Therefore, more precise the set of ECBs, more precise the CRPD we get. If ECB computation does not take into account the infeasible paths in preempting task, set of ECBs might be over-approximated. Therefore, over-estimation in CRPD analysis will also increase. Consequently, we use a model checker for refining the number of ECBs by eliminating infeasible paths found in the preempting task.

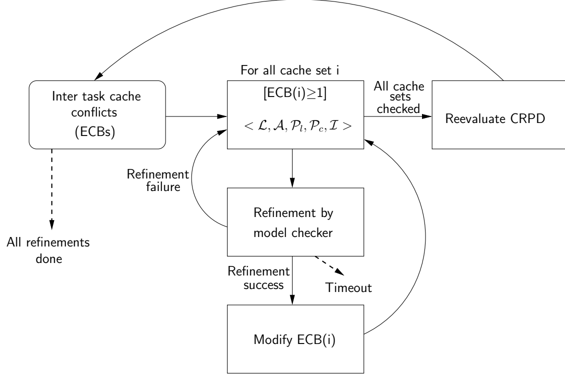

Figure 5. Refinement of inter-task conflict analysis

The refinement of ECBs can be represented in Figure 5. Let us assume $ECB(i)$ represents the number of ECBs computed for cache set i and C_i counts the unique conflicts (in our transformed code) in cache set i generated by the preempting task . The refinement of ECBs are performed in an iterative manner. In each iteration, for all non-zero $ECB(i)$, we try to refine $ECB(i)$ with an immediately smaller value. More precisely, if $ECB(i) = N$, we use the model checker to verify an "assertion" property $C_i \leq N-1$. Note that we need to count the cache conflicts generated by the entire preempting task. Therefore, the conflict count C_i need to be initialized only once, before any cache blocks accessed by the preempting task and the "assertion" property $C_i \leq N - 1$ is placed immediately before the exit point of the preempting task. If the model checker successfully verifies the "assertion" property $C_i \leq N - 1$, we can guarantee that the preempting task cannot generate more than $N-1$ conflicts in cache set i. Therefore, we can update $ECB(i)$ with $N-1$. After each iteration (*i.e.* after checking refinement of $ECB(i)$ with an immediately smaller value for all cache set i), we re-evaluate the CRPD.

The workflow of refinement, as shown in Figure 5, is not restrictive. More specifically, CRPD need not be evaluated after each iteration. The designer may choose to wait for all possible refinements and evaluate the CRPD once all refinements by the MC has been completed. However, evaluation of CRPD after each iteration gives the designer two advantages: first, (s)he can choose to terminate the MC refinement process when a tolerable value of CRPD has been obtained and secondly, (s)he can choose to terminate the MC refinement process if the value of CRPD has not been changed for a reasonably long number of iterations.

We again demonstrate the idea using our example in Figure 3(a). Suppose, the task in Figure 3(a) is a high priority task which may potentially preempt some low priority task. For sake of illustration, assume a 2-way set associative cache where $m1$ and $m5$ map to the same cache set i. Therefore, $ECB(i) = 2$ and the immediate refinement of $ECB(i)$ would check whether the number of unique conflicts in cache set i is less than or equal to 1. The transformed code is shown in Figure 3(e). It checks a property $C_1 \leq 1$ where C_1 counts the number of unique conflicts generated by the preempting task in cache set i. $flag_m1$ and $flag_m5$ are used as guards, so that C_1 counts only unique cache conflicts. When the modified code is passed to a software model checker, it can find out the infeasible path $m1$-$m3$-$m5$ and satisfy the property (*i.e.* $C_1 \leq 1$). Consequently, we can refine the value of $ECB(i)$ as 1. Since $ECB(i)$ is refined to a smaller value, some UCBs in the preempted task may not be evicted from cache set i after preemption, which might in turn lead to a smaller CRPD. As marked in Figure 3(e), our code transformation framework $< \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} >$ is configured as follows: $\mathcal{L} = \{m1, m5\}$, $\mathcal{P}_l = \{L1, L2\}$, $\mathcal{A}$ is the the property $C_1 \leq 1$, $\mathcal{P}_c = \{P2\}$ and $\mathcal{I} = \{I1\}$.

VII. Refinement of inter-core cache conflicts

Finally, we describe the refinement of inter-core conflicts generated in a shared cache (as shown in Figure 2(c)). Recall from Equation 1 that the precision of shared L2 cache analysis largely depends on the accuracy of estimating the term $|\mathcal{M}_c(m)|$. The model checking pass in our framework refines the set $\mathcal{M}_c(m)$ by exploiting infeasible paths in the conflicting task.

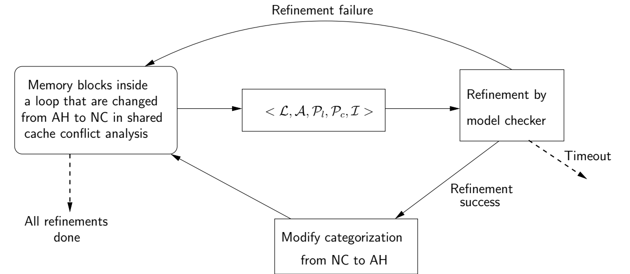

Figure 6. Refinement of shared cache conflict analysis

Figure 6 demonstrates the instantiation of our general framework for inter-core conflict refinement. We only target the memory blocks whose categorizations are changed from AH to NC in a shared cache conflict analysis phase. Consider such a memory block m mapping to an N-way associative shared L2 cache set i. Disregarding the inter-core conflicts, assume the *maximum LRU age* of m in cache set i is denoted by $age(m)$. Therefore, if the amount of inter-core conflicts (in cache set i) is bounded by $N - age(m)$, we can guarantee that m will remain a shared L2 cache hit, despite inter-core conflicts. Recall that $N - age(m)$ is called the *residual age* of m. Further assume t_c is a task which may generate inter-core cache conflicts and C_i serves the purpose of counting inter-core conflicts in shared L2 cache set i generated by t_c. Therefore, we use the model checker to verify an "assertion" property $C_i \leq N - age(m)$. Identical to inter-task cache conflict refinement, we need to check the total amount of cache conflicts generated by task t_c.

Therefore, in our transformed code, we initialize C_i only once, before any cache blocks accessed by t_c and we check the "assertion" property just before the exit point of t_c.

Coming back to the example in Figure 3(a), assume that $m1$ and $m5$ map to the same cache set of a 2-way set associative L2 cache. Further assume that we are trying to refine the shared cache conflict analysis of a task shown in Figure 3(d). The task in Figure 3(d) accesses a memory block m' mapping to the same cache set as $m1$ (or $m5$) in shared L2 cache and it is run parallely on a different core with the task in Figure 3(a). Finally assume, m' is an all-hit (AH) in L2 cache *with residual age one* but an all-miss (AM) or unclassified (NC) in L1 cache from the second iteration of the loop. Previous analysis will compute $|\mathcal{M}_c(m')|$ as 2 (due to $m1$ and $m5$ in the conflicting task). Since the residual age of m' is one, the categorization of m' will be changed to NC (Equation 1), leading to unnecessary conflict misses. We modify the code to check whether the number of unique inter-core conflicts is less than or equal to the residual age of m'. The transformation is similar to Figure 3(e) where C_1 serves the purpose of counting unique cache conflicts with m' in shared L2 cache. The model checker will satisfy the assertion P2 in Figure 3(e) due to the infeasible path $m1$-$m3$-$m5$. Consequently, we shall be able to derive that the amount of inter-core conflicts with m' never exceeds the residual age of m'. Therefore, the categorization of m' is kept all-hit (AH) from the second iteration of the loop. Configuration of our code transformation framework $< \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} >$ is identical to the inter-task cache conflict refinement as follows: $\mathcal{L} = \{m1, m5\}$, $\mathcal{P}_l = \{L1, L2\}$, $\mathcal{A}$ is the "assertion" clause checking the property $C_1 \leq 1$, $\mathcal{P}_c = \{P2\}$ and $\mathcal{I} = \{I1\}$.

Although we show the transformation for a two core system, our framework does not have the strict limitation of working only for two cores. However, one model checker invocation can verify only one task. Therefore, to refine conflicts from X different tasks $t_1, t_2, \ldots, t_X$ running on X different cores, we first employ an additional *compose* phase in transformation. The *compose* phase sequentially composes $t_1, t_2, \ldots, t_X$ (in any order) into a single task T. The infeasible paths in any task $t_1, t_2, \ldots, t_X$ are preserved in task T. Consequently, our code transformation technique can be applied to T in exactly same manner as described in the preceding to refine conflicts from $t_1, t_2, \ldots, t_X$. Since the composition is sequential, number of conflicts are accumulated from all X cores. Model checker refinement passes can then be carried out on task T.

VIII. Optimizations

To reduce the number of calls to model checker, we cache the verification results. Recall that the "assertion" property verified by the model checker was always placed at the end of conflicting task during inter-task and inter-core cache conflict refinement. However, during intra-task cache conflict refinement, the position of "assertion" property may vary and depends on the position of NC categorized memory block being refined. Therefore, the following optimization can be applied only during inter-task and inter-core conflict refinement but *not* during intra-task conflict refinement.

Model checker results are stored as a triple $(set,\ result_{mc},\ conflicts)$. The triple has the following meaning:

- set : Cache set for which the refinement is being made.
- $result_{mc}$: Returned result by the model checker. Assume $result_{mc}$ is one for a successful verification and zero otherwise.
- $conflicts$: Number of conflicts in the assertion property. If we verify an assertion property $C_i \leq N$, value of $conflicts$ is N.

In Figure 3(e), we store $(1, 1, 1)$ after the successful refinement (assuming $m1$ and $m5$ map to cache set 1). Assume any other assertion of form $C_{set'} \leq N'$ is needed to be verified, where set' is the cache set for which the conflicts are being refined. We search the cached results of form $(set,\ result_{mc},\ conflicts)$ and take an action as follows:

- $set = set' \wedge result_{mc} = 0 \wedge N' \geq conflicts$: Assertion failure is returned. If the refinement previously failed for a less number of conflicts, it will definitely fail for more conflicts.
- $set\ =\ set' \wedge result_{mc}\ =\ 1 \wedge N'\ \leq\ conflicts$: Assertion success is returned. If the refinement was previously satisfied for more number of conflicts, it must be satisfied for less number of conflicts.

If none of the entries satisfy the above two conditions, a new call to the model checker is made. Depending on the outcome, the new result is cached accordingly for future use.

IX. Implementation

We have used the Chronos timing analysis tool [2] in which we have already integrated the AI based cache analysis proposed in [6] (for single core) and [12] (for multiple cores). Chronos employs detailed micro-architectural modeling (superscalar, out-of-order pipeline and branch prediction). We have also integrated the recently proposed CRPD analysis ([11] and [10]) into Chronos.

For model checking purposes, we use C bounded model checker (CBMC) [19]. CBMC formally verifies ANSI-C programs through bounded model checking (BMC) [20]. For a given system/program P, BMC unwinds P to a certain depth. After unwinding, a Boolean formula is obtained that is satisfiable if and only if there exists a counter example trace. The formula is checked by a SAT procedure. If the formula is satisfiable, a counter example is produced from the output of SAT procedure. Technically, for a C program, the unwinding is achieved by unrolling the program loops to a certain depth. For a given unwinding depth n, CBMC unwinds a loop by duplicating the code of loop body n

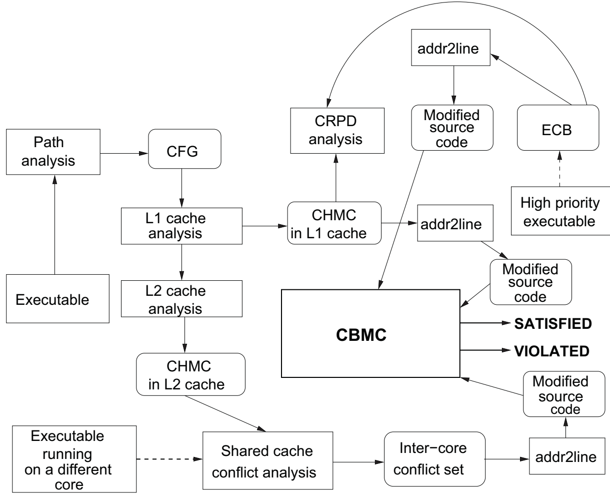

Figure 7. Implementation framework

times. Each copy is guarded by the loop entry condition and hence, covering the cases where the loop executes for less than n iterations. The main advantage of CBMC is that the tool also checks whether sufficient unwinding has been done and thereby ensures that no *longer* counterexample can exist. Technically, CBMC achieves the same by putting an "assertion" (called *unwinding assertion*) after the last copy of the unrolled loop. The assertion uses the negated loop entry condition and therefore, it ensures that the program never requires more iterations. *In summary, if no counterexample is produced by CBMC, it ensures the absence of error in the program for any execution*.

As described in the preceding, CBMC requires unwinding depth (bound) of each loop. If user does not specify any unwinding depth (loop bound), CBMC tries to determine the depth automatically. In most of our experiments, CBMC was able to determine the loop bound automatically. For the cases where CBMC failed to determine the loop bound, we passed sufficient loop bound for each loop as an input to CBMC. Recall that CBMC automatically put an "assertion" clause (called an *unwinding assertion*) after the last unwound copy of a loop. The assertion clause verifies the negated loop entry condition. Therefore, if insufficient loop bound is provided by the user, CBMC generates an unwinding assertion violation and the verification process returns a failure. Consequently, user can give a larger loop bound and rerun CBMC. However, in our experiments, we initially provided sufficient loop bounds, so that no unwinding assertion is violated. In our current implementation, CBMC is called as an external module. Therefore, for each different call of CBMC, the loop unwinding needs to be performed. Running time of our analysis can certainly improve if we can restrict the number of loop unwindings. This will require us to make use of CBMC and Chronos in a single binary executable, which could be explored in future.

Figure 7 gives an overall picture of our implementation framework. The figure demonstrates one refinement for each type of conflicts. Chronos employs AI based cache analysis directly on the executable. We use a utility `addr2line` which converts an instruction address to corresponding source code line number. The information generated by `addr2line` is used to generate the transformed code. The transformation of code is *entirely automatic*. Note that the sole purpose of the transformed code is to prove that certain cache conflicts in the original code are *infeasible*. Therefore, the timing effects generated by the original code is entirely independent of the additional code introduced in transformation. The transformed code contains an "assertion" property to be verified by CBMC. CBMC either successfully verifies the assertion property or generates a counter example. We would finally like to point out that the central contribution of this paper is an efficient composition of abstract interpretation and model checking. Therefore, even though we have used CBMC for model checking, our proposed framework (Figure 1) remains unchanged if we use a model checker that directly works on the executable (*e.g.* [21]). Nevertheless, there are certain advantages of using a model checker like [21]. Since [21] directly works on the executables, it can capture the effect of all compiler optimizations. Our technique can be integrated with [21] to make a more *robust* WCET analysis framework.

X. Experimental evaluation

We have chosen benchmarks from [4] which are generally used for timing analysis. Note that the main motivation of our work is to remove *false cache conflicts*, which were introduced due to the infeasible paths. Infeasible paths are often introduced when auto generating code from a high level modeling language (*e.g.* `esterel` as shown in [18]). For evaluation of our framework, therefore, we need a set of tasks which potentially exhibit many paths. Table I demonstrates a set of benchmarks having multiple paths. Let us call the set of tasks in Table I as *conflicting task set*. Each task in the *conflicting task set* serves the purpose of the task used in Figure 3(a). Therefore, model checker refinement pass is used on the tasks from *conflicting task set*. We use another set of selected benchmarks from [4] as shown in Table II during inter-task and inter-core conflict refinement. We call the tasks in Table II as *standard task set*. During inter-task and inter-core conflict refinement, we refine the conflicts generated by conflicting task set on the standard task set. We report our experiences for each possible combinations of standard and conflicting task set.

Task	Description	code size (bytes)
statemate	Automatically generated code from Real-time-Code generator STARC	52618
compress	Data compression program	13411
nsichneu	Simulate an extended petri-net	118351

Table I
Conflicting task set

Task	Description	code size (bytes)
cnt	Counts non-negative numbers in a matrix	2880
fir	Finite impulse response filter	11965
fdct	Fast discrete cosign transform	8863
jfdcint	discrete cosign transform on 8×8 block	16028
edn	signal processing application	10563
ndes	complex embedded code	7345

Table II
STANDARD TASK SET

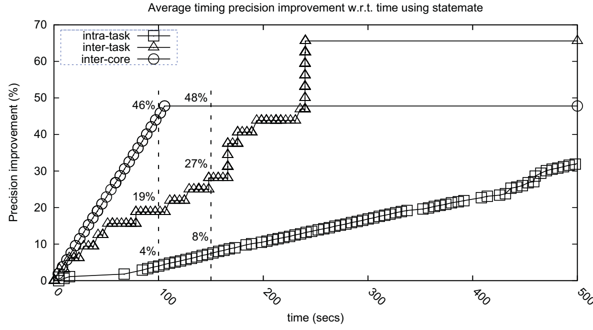

Figure 8. Timing precision improvement w.r.t. time using *statemate*

We use the following terminology in presenting the experimental data: i) $WCET_{base}$: WCET before any refinement by model checker. ii) $WCET_{refined}$: WCET after refinement by model checker. iii) $CRPD_{base}$: CRPD before any refinement by model checker. iv) $CRPD_{refined}$: CRPD after refinement by model checker. WCET improvement is computed as $\frac{WCET_{base}-WCET_{refined}}{WCET_{base}} \times 100\%$. CRPD improvement is computed similarly as $\frac{CRPD_{base}-CRPD_{refined}}{CRPD_{base}} \times 100\%$.

Our framework uses the usual 5-stage pipeline (IF-ID-EX-MEM-WB) implemented by Chronos when predicting the WCET value. The experimental data are taken for an inorder pipeline. However, the data can also be obtained for an out-of-order pipeline in exactly same manner. We fix the L1 cache miss latency as 6 cycles and L2 cache miss latency as 30 cycles for all the experiments. For the experiments which do not have an L2 cache (*e.g.* inter-task and intra-task conflict refinement), we simply take the L1 cache miss penalty as 36 cycles. All reported experiments have been performed in an Intel core-2 duo machine having 2 GB of RAM and running ubuntu 10.10 operating systems. The reported total time captures the entire time taken during analysis — including the base analysis through abstract interpretation and repeated model checking steps.

Key result: Before going to the details of each experiment, let us first demonstrate the key result of this paper via Figure 8. Figure 8 shows the improvement of WCET and CRPD with respect to wall clock time using *statemate*. The improvement of WCET is demonstrated in case of intra-task and inter-core cache conflict refinement. On the other hand, CRPD improvement is shown for inter-task cache conflict refinement. We observe that intra-task and inter-core cache conflict refinement demonstrates an almost linear improvement in timing precision (*i.e.* improvement in WCET) with respect to time. On the other hand, CRPD improvement shows a step-wise behaviour. Recall from Section VI that we refine the set of evicting cache blocks in an iterative manner. In our experiments, we observe that most of the time CRPD is improved only after one complete iteration of our inter-task cache conflict refinement. Therefore, the steady portion of the graph (where CRPD is not improved) demonstrates the time of one iteration in our inter-task cache conflict refinement.

As our result is always *safe*, a provably correct WCET/CRPD value can be obtained from any vertical cut along the time axis of Figure 8. As illustrated in Figure 8, consider the cut at 100th second. It clearly shows that if we end the model checker refinement process after 100 seconds, we can obtain 4%, 19% and 46% improvement during intra-task, inter-task and inter-core cache conflict refinement, respectively. Nevertheless, if the model checker refinement process is allowed more time to run, we can obtain better precision in our obtained result (8%, 27% and 48% respectively for intra-task, inter-task and inter-core conflict refinement after 150 seconds, as shown in Figure 8).

Reducing intra-task cache conflicts: Clearly, our refinement depends both on the choice of conflicting task set and cache size. We choose a 4-way associative, 8 KB L1 cache with 32 bytes of block size. Applying intra-task cache analysis on *compress* does not leave any NC categorized memory blocks inside loop. Therefore, our refinement pass using CBMC did not have any additional effect in improving the WCET for *compress*. On the other hand, *statemate* and *nsichneu* contain very large loops (in terms of code size) with multiple paths inside a loop body. Consequently, AI based cache analysis generates a large number of NC categorized memory blocks. The result obtained for *statemate* and *nsichneu* is presented in Table III. As shown in Table III, for both *statemate* and *nsichneu*, we are able to refine many of the NC categorized memory blocks (*e.g.* 68 out of 100 calls return success when experimenting with *statemate*). We show the refinement process for a maximum of 100 model checker steps (as shown by the "MC steps" column in Table III). Nevertheless, if time budget permits, the refinement process can be run longer and thereby provides more opportunities to improve the WCET. This result demonstrates the potential of our approach even for improving the most basic cache conflict analysis through AI.

Reducing inter-task cache conflicts: We present the result of inter-task conflict refinement in Table IV. CRPD reported in Table IV ($CRPD_{base}$ and $CRPD_{refined}$) denotes the cache related preemption delay when a low priority task from standard task set is preempted by a high priority task from conflicting task set. As before, we choose a 4-way associative, 8 KB L1 cache with 32 bytes block size. We are able to reduce the number of ECBs as well as the CRPD when *compress*, *statemate* and *nsichneu* are used as high priority tasks. CRPD improvement is significant,

program	NC inside loop	NC refined	$WCET_{base}$ (in cycles)	$WCET_{refined}$ (in cycles)	Improvement(%)	MC steps	time(secs)
statemate	350	68	19188	14834	22.7%	100	395
nsichneu	697	98	91000	84174	7.5%	100	558

Table III
REFINEMENT OF INTRA-TASK CACHE CONFLICTS. *We use a 4-way associative, 8 KB cache with 32 bytes block size.*

program (low+high)	ECB before	ECB after refinement	$CRPD_{base}$ (in cycles)	$CRPD_{refined}$ (in cycles)	Improvement(%)	time (secs)	MC calls
cnt + statemate	128	105	1260	684	45.7%	145.69	27
fir + statemate	128	112	972	72	92.6%	123.73	19
fdct + statemate	128	103	1152	396	65.6%	181.6	32
jfdcint + statemate	128	103	1224	648	47.1%	182.8	32
edn + statemate	128	103	4464	2664	40.3%	187.34	32
ndes + statemate	128	103	2844	720	74.7%	193.5	32
cnt + nsichneu	256	217	1152	396	65.6%	390.58	52
fir + nsichneu	256	218	828	72	91.3%	304.96	34
fdct + nsichneu	256	147	612	36	94.1%	465.50	52
jfdcint + nsichneu	256	198	792	72	90.9%	554.26	46
edn + nsichneu	256	176	3492	144	95.9%	769.41	64
ndes + nsichneu	256	178	5328	108	98%	743.84	61
cnt + compress	107	82	432	0	100%	139.84	27
fir + compress	107	58	324	0	100%	137.51	19
fdct + compress	107	97	396	72	81.8%	144.35	32
jfdcint + compress	107	97	648	72	88.9%	144.58	32
edn + compress	107	97	2448	108	95.6%	148.29	32
ndes + compress	107	97	900	36	96%	152.25	32

Table IV
REFINEMENT OF INTER-TASK CACHE CONFLICTS. *We use a 4-way associative, 8 KB cache with 32 bytes block size.*

program set	CHMC changed	CHMC refined	$WCET_{base}$ (in cycles)	$WCET_{refined}$ (in cycles)	Improvement(%)	time (secs)	MC calls
cnt + statemate	29	7	212913	113313	46.8%	23.54	3
fir + statemate	27	10	478860	415800	13.2%	38.41	5
fdct + statemate	83	47	14124	7374	47.8%	90.21	27
jfdcint + statemate	114	69	193436	87386	54.8%	91.53	29
edn + statemate	204	127	204136	148276	27.4%	115.65	23
ndes + statemate	225	163	208392	97932	53%	93.44	29
cnt + nsichneu	16	7	211533	113133	46.5%	23.67	2
fir + nsichneu	15	7	457680	415650	9.2%	26.99	3
fdct + nsichneu	38	23	10344	7104	31.3%	99.17	13
jfdcint + nsichneu	45	26	123746	84356	31.8%	111.25	14
edn + nsichneu	73	46	116742	80592	31%	84.09	10
ndes + nsichneu	95	68	209442	112512	46.3%	99.81	15
cnt + compress	32	14	262083	113283	56.8%	15.61	4
fir + compress	19	6	467570	415710	11.1%	22.95	4
fdct + compress	86	47	14394	7644	46.9%	69.67	15
jfdcint + compress	108	58	196466	117686	40.1%	107.41	18
edn + compress	185	101	222166	166336	25.1%	126.55	18
ndes + compress	174	122	179832	103242	42.6%	114.81	15

Table V
REFINEMENT OF INTER-CORE CACHE CONFLICTS. *We use a 4-way associative, 8 KB cache with 32 bytes block size.*

with average improvement being more than 80%. Note that we use a set associative cache. Therefore, conflicts generated from high priority tasks may just age the used cache blocks in the low priority tasks (instead of completely evicting the used cache blocks by the low priority task). Consequently, for *(cnt,compress)* and *(fir,compress)* pair, we are able to completely eliminate the CRPD. More over, we refine evicting cache blocks in cache set i only if the low priority task has used the cache set i. Therefore, even for the same preempting task, Table IV might have different number of evicting cache blocks refined (as shown in the column "ECB after refinement").

Unlike the intra-task conflict refinement, results reported in Table IV run the refinement process till end (*i.e.* unless all possible refinements have been checked). Corresponding number of CBMC calls is shown in the column "MC calls".

Reducing inter-core cache conflicts: Finally, we present the result of inter-core cache conflict refinement in Table V. In one core, we run a task from the standard task set (in Table II) and in another core, we run a task from the conflicting task set (in Table I). Reported WCETs represent the WCETs of tasks from the standard task set. For experiments reported in Table V, we need the analysis of both L1 and L2 cache. We fixed the L1 cache as a direct-mapped, 256

bytes with a block size of 32 bytes. L1 cache is taken relatively small so that we are able to generate reasonable number of conflicts in the shared L2 cache. We take a 4-way associative, 8 KB shared L2 cache having a cache block size of 32 bytes. As expected, we are able to significantly reduce the standard task WCET by refining the inter-core cache conflicts (maximum improvement upto 57%). All our experiments complete within two minutes with maximum number of model checker calls being only 29 (as shown by the column "MC calls").

XI. Conclusion and future work

In this paper, we have proposed a scalable WCET analysis framework using the combination of abstract interpretation and model checking for cache analysis. Our framework does not affect the flexibility of abstract interpretation based cache analysis and it can be composed with the analysis of different other micro-architectural features (*e.g.* pipeline). More over, our model checker refinement process is always *safe*. Therefore, the model checker refinement can be terminated at any point if the time budget is violated. Experimental results show that we can obtain significant improvement for various types of cache analysis in single and multi-cores.

Our current work can be extended in several directions: first, our framework currently handles LRU cache replacement policy. It could be extended to deal with other cache replacement policies (*e.g.* FIFO, pseudo-LRU etc). Nevertheless, this would require to change the base analysis (using abstract interpretation) and the code transformation technique to target the corresponding cache replacement policy. Secondly, we capture the *global persistence* of a memory block during intra-core cache conflict refinement. Therefore, a successful model checker refinement ensures that the corresponding memory block can *never* be evicted from the cache during program execution. Our code transformation technique can be extended to capture *scope based persistence* information. In scope based analysis, a memory block may exhibit different persistence behaviours in different loop nests (refer to [22]). Finally, using the scope based refinement technique, our framework can be extended for refining data cache analysis result. In future, we plan to investigate the refinement of data cache analysis outcome through model checking.

XII. Acknowledgement

This work was partially supported by NUS grants R-252-000-416-112 and R-252-000-385-112.

References

[1] M. Lv et. al. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS*, 2010.

[2] X. Li et. al. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 2007. http://www.comp.nus.edu.sg/~rpembed/chronos.

[3] aiT AbsInt. http://www.absint.com/ait.

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

[5] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Trans. Des. Autom. Electron. Syst.*, 4(3), 1999.

[6] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3), 2000.

[7] C.G. Lee et. al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6), 1998.

[8] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS*, 2003.

[9] Y. Tan and V.J. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *SCOPES*, 2004.

[10] S. Altmeyer and C. Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *ECRTS*, 2009.

[11] S. Altmeyer, C. Maiza, and J. Reineke. Resilience analysis: tightening the CRPD bound for set-associative caches. In *LCTES*, 2010.

[12] Y. Li et. al. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.

[13] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared l2 instruction caches. In *RTAS*, 2008.

[14] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *RTSS*, 2009.

[15] R. Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *VMCAI*, 2004.

[16] A. Metzner. Why model checking can improve WCET analysis. In *CAV*, 2004.

[17] A. E. Dalsgaard et. al., 2011. Metamoc: Modular execution time analysis using model checking.

[18] L. Ju et. al. Performance debugging of Esterel specification. In *CODES+ISSS*, 2008.

[19] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.

[20] E. Clarke et. al. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19, 2001.

[21] G. Balakrishnan et. al. Model checking x86 executables with codesurfer/x86 and wpds++. In *CAV*, 2005.

[22] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *RTAS*, 2011.