T H E N A T I O N A L U N I V E R S I T Y
o f S I N G A P O R E

S c h o o l o f C o m p u t i n g
Computing 1, 13 Computing Drive, Singapore 117417

TRA5/11

Scalable and Precise Refinement of Cache Timing Analysis via Model Checking

Sudipta Chattopadhyay and Abhik Roychoudhury

May 2011

T e c h n i c a l   R e p o r t

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. Subsequently, worst case execution time (WCET) has increasingly become an important problem. However, performance enhancing features of a processor (*e.g.* cache) makes 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 of tasks is 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 commonly 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-approximation 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 potential 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 thus 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. Thus, 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 parameterizable cache analysis framework for real-time systems which is generic, precise and scalable.

II. Related work

Research on WCET analysis has been initiated decades ago. Cache modeling has been an active topic of research in this area. Initial works used Integer Linear Programming (ILP) [5] for cache modeling. However, [5] faces scalability problems in terms of analysis time. Subsequently, abstract interpretation based cache analysis [6] has been proposed. [6] has efficiently composed the cache modeling with ILP based path analysis and the solution has also been adopted in commercial tool chain [3]. [6] has also been extended with level 2 cache analysis in single core by [7].

In last decade, there has also been an extensive amount of research to bound cache related preemption delay (CRPD) [8], [9], [10]. Recently, [11] has improved the CRPD computed by previous approaches and [12] has eliminated a potential *unsafe* CRPD computation for set-associative caches.

With the advent of multi-core architectures, research on multi-core specific hardware resources has also become very popular. Analysis of shared cache [13], [14], [15] is one such example. All the above mentioned cache analysis use abstract interpretation for modeling purposes.

In [16], it is argued that model checking alone is not suitable for WCET analysis due to the state space explosion problem. On the other hand, [17] uses model checking alone for cache and path analysis. However, [17] does not employ the modeling of other micro-architectural features (*e.g.* pipeline, branch predictor etc) and it is unclear whether the employed technique would remain scalable in presence of pipeline or other micro-architectural features.

In summary, abstract interpretation based approach is scalable and easy to integrate with other micro-architectural features. 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] keeps the model checking on path analysis level and uses abstract interpretation for cache analysis. In this paper, we study the composition of model checking and abstract interpretation for different cache analysis to design a scalable and precise WCET analysis framework. Compared to previous approaches, our framework has a major advantage of easily being composed with the analysis of other micro-architectural components (*e.g.* pipeline, branch prediction).

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 behaviour of different micro-architectural components (*e.g.* cache, pipeline, branch predictor). Typically, micro-architectural modeling works on the granularity of basic blocks. As an outcome of micro-architectural modeling, we get the WCET of each basic block in the examined program. On the other hand, path analysis computes the infeasible program paths. Our baseline implementation for micro-architectural modeling and path analysis is a separated one, as proposed in [6]. As a baseline, we also use the widely adopted abstract interpretation based cache analysis proposed in [6]. 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 *cold cache misses* in first iteration of the loop. If a memory block is categorized as AH, it means that the memory block is always in cache whenever it is accessed. On the other hand, if a memory block is categorized as AM, it means that the memory block is never in the cache whenever it is 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 (ILP). The formulated ILP uses the basic block WCETs and structural constraints imposed by program control flow graph (CFG). 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 *evicted 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 and may be used later, resulting in a cache hit. The number of UCBs pose 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. Recently, [11] has improved the previous approaches [8], [9] by reducing the number of UCBs to consider for CRPD computation. Another recent work ([12]) has improved and corrected the CRPD analysis for set-associative caches. Our implementation of CRPD analysis includes the correction for set-associative caches as proposed in [12] and also includes the improvement suggested by [11]. Therefore, our basic implementation is the best known and correct implementation so far. For a detailed description of CRPD analysis, readers are referred to [8].

Inter-core cache conflict analysis: Inter-core cache conflict analysis computes the conflicts generated in shared cache by a task running on a different core. Till now, only a few solutions have been proposed for shared cache analysis [13], [15], [14]. However, all of them suffer from over-estimating the cache conflicts generated by the task running on a different core. We use our former work on shared cache analysis [13] which employs a separate shared cache conflict analysis phase. Shared cache conflict analysis changes the categorization of a memory block m from all-hit (AH) to unclassified (NC). The categorization is changed by computing the number of unique conflicting cache accesses from other core and checking whether the number of conflicts can potentially replace m. More specifically, categorization of m is changed from AH to NC if the following condition holds:

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

where $|\mathcal{M}_c(m)|$ represents the number conflicting memory blocks from other core 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 disregarding the conflicts from other core. 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

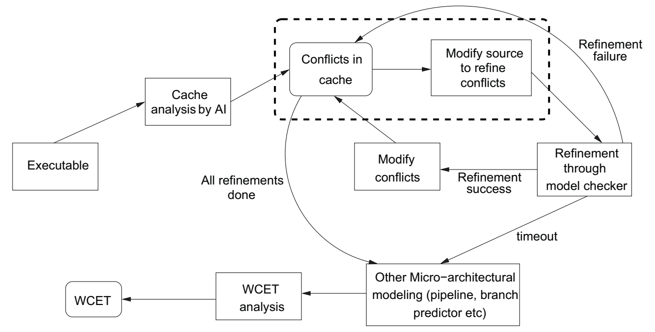
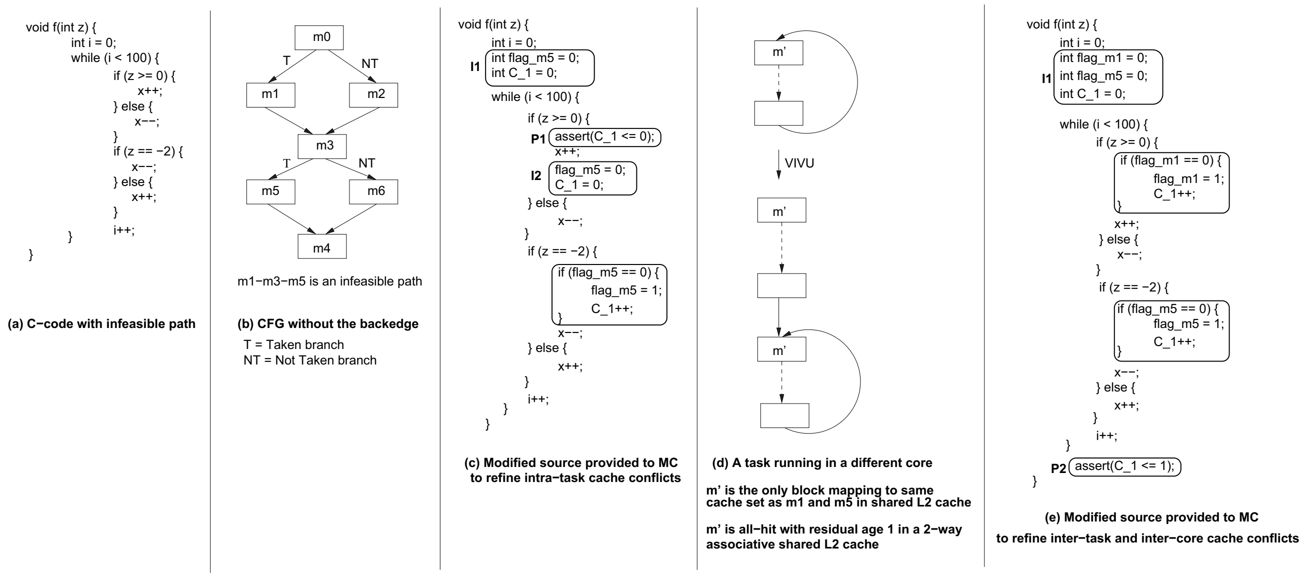


Figure 1. General framework of our WCET analysis which combines abstract interpretation and model checking

analysis can accurately predict the set of cold cache misses. However, AI based cache analysis suffers from accurately predicting 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 3 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 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 each memory block. Since our goal is to improve timing precision, we concentrate on *unclassified* (NC) memory blocks. 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 source program. The model checker is invoked only for improving the timing analysis result (WCET or CRPD). Therefore, the model checker is invoked only for a subset of cache accesses which AI has failed to analyze accurately. Consequently, the amount of time required for refinement could be tolerated. 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


T
NT
m0
m1
m2
m3
m5
m6
m4
m1−m3−m5 is an infeasible path
VIVU
m'
(a) C−code with infeasible path
(b) CFG without the backedge
T = Taken branch
NT = Not Taken branch
(c) Modified source provided to MC to refine intra−task cache conflicts
(d) A task running in a different core
m' is the only block mapping to same cache set as m1 and m5 in shared L2 cache
m' is all−hit with residual age 1 in a 2−way associative shared L2 cache
(e) Modified source provided to MC to refine inter−task and inter−core cache conflicts

```
void f(int z) {
        int i = 0;
        while (i < 100) {
                if (z >= 0) {
                        x++;
                } else {
                        x−−;
                }
                if (z == −2) {
                        x−−;
                } else {
                        x++;
                }
                i++;
        }
}
```

```
void f(int z) {
    int i = 0;
I1  int flag_m5 = 0;
    int C_1 = 0;
    while (i < 100) {
        if (z >= 0) {
P1          assert(C_1 <= 0);
            x++;
I2          flag_m5 = 0;
            C_1 = 0;
        } else {
            x−−;
        }
        if (z == −2) {
            if (flag_m5 == 0) {
                flag_m5 = 1;
                C_1++;
            }
            x−−;
        } else {
            x++;
        }
        i++;
    }
}
```

```
void f(int z) {
    int i = 0;
    int flag_m1 = 0;
I1  int flag_m5 = 0;
    int C_1 = 0;

    while (i < 100) {
        if (z >= 0) {
            if (flag_m1 == 0) {
                flag_m1 = 1;
                C_1++;
            }
            x++;
        } else {
            x−−;
        }
        if (z == −2) {
            if (flag_m5 == 0) {
                flag_m5 = 1;
                C_1++;
            }
            x−−;
        } else {
            x++;
        }
        i++;
    }
P2  assert(C_1 <= 1);
}
```

Figure 2. Refinement of various cache conflicts

performed by MC. 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 due to various cache conflict refinement can be generally represented by a quintuple $<\mathcal{L}, \mathcal{C}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I}>$ as follows:

- $\mathcal{L}$: Set of conflicting memory blocks in the cache set for which the refinement is being made.
- $\mathcal{C}$: The property to be checked by the model checker. The property is placed in form of an "assertion" clause, which validates $\mathcal{C}$ for all possible execution traces.
- $\mathcal{P}_l$: Set of positions in source 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 source code where the property $\mathcal{C}$ would be placed.
- $\mathcal{I}$: Set of positions in the source code to reset conflict count.

Any refinement corresponds to a specific cache set and therefore, conflicts are defined for a specific cache set in each transformation. Therefore, 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, $\mathcal{C}$, $\mathcal{P}_c$ 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 3). We shall also show how $\mathcal{C}$, $\mathcal{P}_c$ and $\mathcal{I}$ are configured depending on the type of cache conflict being refined.

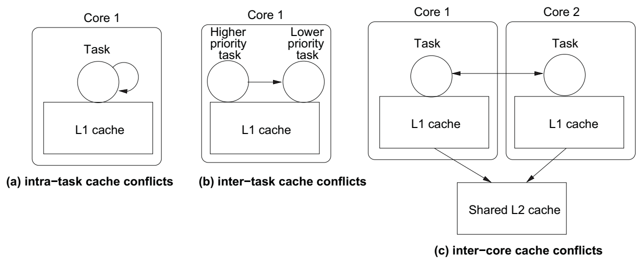


Figure 3. Variants of cache conflicts

V. Refinement of intra-task cache conflicts

In this section we describe the refinement of cache conflicts shown in Figure 3(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 demonstrates the instantiation of our general framework for reducing the over-estimation in WCET analysis. As shown in Figure 4, we only target the NC categorized memory blocks inside some loop (eliminating all NC categorized memory blocks that are accessed only once). Therefore, we concentrate only on a few memory blocks whose

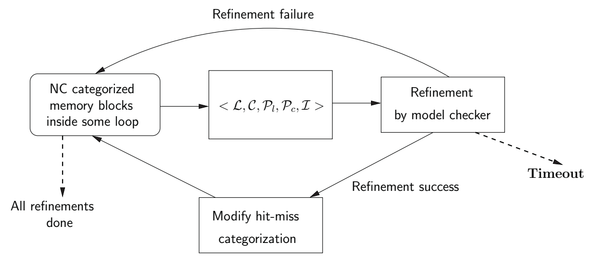

Refinement failure
NC categorized memory blocks inside some loop
$< \mathcal{L}, \mathcal{C}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} >$
Refinement by model checker
Timeout
Refinement success
All refinements done
Modify hit-miss categorization

Figure 4. Refinement of intra-task conflict analysis

successful refinement may lead to a reasonable WCET improvement. For each of the NC categorized memory blocks under consideration, we call our general code transformation framework (as shown in Figure 4). Let us assume we want to refine the categorization of a particular memory reference m mapping to a cache set i. m is the *most recently used* cache block immediately after it is accessed. Therefore, 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 represents the number of unique conflicts in cache set i and N is the associativity of the cache. For a successful refinement (m being a cache hit), C_i must be less than or equal to $N-1$ each time m is accessed. Therefore, in our general transformation framework, $\mathcal{C}$ is of form $C_i \leq N-1$. More over, $\mathcal{P}_c$ is the program point immediately before the reference for which the refinement is being made. As said before, m is the most recently used cache block immediately after it is accessed. Therefore, the conflicts are reset immediately after m is accessed. Consequently, $\mathcal{I}$ is the program point immediately after the reference for which the refinement is being made.

We demonstrate our technique through an example in Figure 2(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 2(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 2(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 is 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 with condition ($flag\_m5$ serves the purpose of guard), so that we count only unique memory block accesses. 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 2(c). Consequently, the categorization of $m1$ can be upgraded to *all-hit* except one (one miss accounts for cold cache miss).

VI. Refinement of inter-task cache conflicts

Here we show the refinement of inter-task cache conflicts (as shown in Figure 2(b)) and thereby reduce the over-estimation introduced by CRPD analysis. A major source of over-estimation in CRPD analysis comes from the computation of evicted cache blocks (ECB). ECB denotes the set of cache blocks possibly touched by the preempting task. ECB computation is path insensitive and therefore, it does not account the infeasible paths in the preempting task. As before, we use a model checker for refining the number of ECBs by eliminating infeasible paths found in the preempting task.

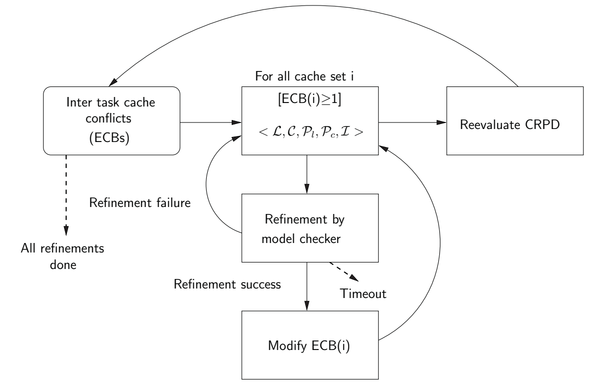

For all cache set i
[ECB(i)≥1]
Inter task cache conflicts (ECBs)
$< \mathcal{L}, \mathcal{C}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} >$
Reevaluate CRPD
Refinement failure
Refinement by model checker
All refinements done
Refinement success
Timeout
Modify ECB(i)

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 at cache set i and C_i represents the number of cache blocks accessed by the preempting task at cache set i. The refinement of ECBs are performed in an iterative manner. In each iteration, we refine $ECB(i)$ with an immediate better value. More precisely, if $ECB(i) = N$, we use the model checker to verify whether $C_i \leq N-1$. The property also explains the entry $\mathcal{C}$ in our code transformation framework. For a successful refinement, $ECB(i)$ is updated with new value $N-1$. After all cache sets are checked in one pass, we reevaluate the CRPD. If the CRPD value is desirable, the refinement can *safely* be terminated. Finally, we check the conflicts generated by the entire preempting task. Therefore, the conflicts are initialized only once, at the beginning of the preempting task (explains $\mathcal{I}$) and the property $\mathcal{C}$ is placed at the end of preempting task (explains $\mathcal{P}_c$).

We again demonstrate the idea using the example in Figure 2(a). Suppose, the task in Figure 2(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. Therefore, the ECB computation in literatures assumes that both $m1$ and $m5$ conflict in cache with the low priority task. The transformation of code is shown in Figure 2(e). C_1 denotes the number of unique memory block accesses in the corresponding cache set (in which $m1$ and $m5$ map) by the preempting task. $flag\_m1$ and $flag\_m5$ are used as guards, so that C_1 counts only unique memory blocks. Due to presence of $m1$ and $m5$, number of ECBs at the corresponding cache set is two. To refine the number of ECBs, we check whether the value of C_1 is less than or equal to 1. When the modified source is passed to a software model checker, it can find out the infeasible path $m1$-$m3$-$m5$ and satisfy the verification criteria (*i.e.* $C\_1 \leq 1$).

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 3(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.

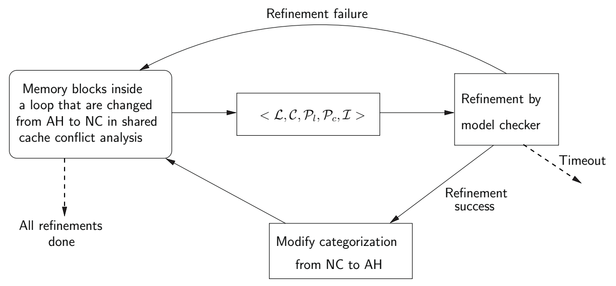

Refinement failure
Memory blocks inside a loop that are changed from AH to NC in shared cache conflict analysis
$< \mathcal{L}, \mathcal{C}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} >$
Refinement by model checker
Timeout
Refinement success
All refinements done
Modify categorization from NC to AH

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. Assume m is such a memory block mapped to shared L2 cache set i. Further assume a conflicting task running on a different core uses C_i number of cache blocks in shared L2 cache set i. Therefore, if we can prove that C_i is less than or equal to the residual age of m, we can be sure that m cannot be evicted from the shared L2 cache due to inter-core conflicts. Consequently, categorization of m can be reverted back to AH. In our transformation framework, we formulate $\mathcal{C}$ as a property of form $C_i \leq residual\_age(m)$. On the other hand, we check the conflicts generated by an entire task running on a different core. Therefore, the conflicts are initialized only once, at the beginning of the conflicting task (explains $\mathcal{I}$) and the property $\mathcal{C}$ is placed at the end of conflicting task (explains $\mathcal{P}_c$).

Coming back to the example in Figure 2(a), assume that $m1$ and $m5$ maps 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 2(d), when it is run parallely on a different core with the task in Figure 2(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, leading to unnecessary conflict misses. However, as before, if we modify the source as in Figure 2(e) and pass it to the model checker, the model checker will satisfy the assertion. Consequently, we shall be able to derive that number of conflicting blocks from other core 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.

Although we show the transformation for a two core system, our framework does not have the strict limitation of working only for two cores. The model checker can handle only one task at one invocation. 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 $\{t_1, t_2, \ldots, t_X\}$ into a single task T. Our code transformation technique can be applied to T in exactly same manner as described above to refine conflicts from $\{t_1, t_2, \ldots, t_X\}$.

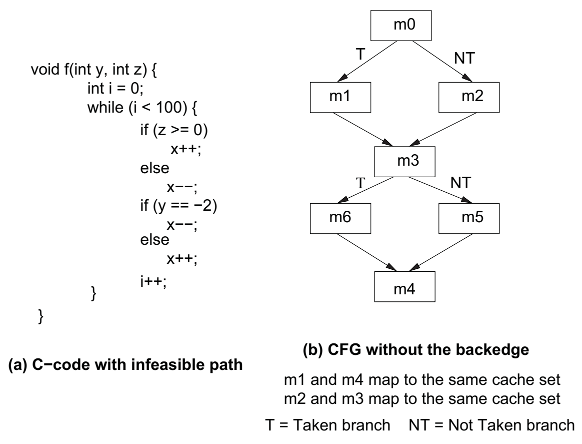
VIII. Optimizations and extensions

Reducing number of calls to model checker: To reduce the number of calls to model checker, we cache the verification results. It could be observed that during inter-task and inter-core cache conflict refinement, $\mathcal{P}_c$ does not depend on the cache set for which refinement is being made (property $\mathcal{C}$ is always put at the end of conflicting task). Therefore, we store the result returned by the model checker as a triple $(set,\ result_{mc},\ conflicts)$. The triple has the following significance: i) set : Cache set for which the refinement is being made. ii) $result_{mc}$: Returned result by the model checker. Assume $result_{mc}$ is zero for a successful verification and nonzero otherwise. iii) $conflicts$: Number of conflicts in the assertion clause ($\mathcal{C}$). In Figure 2(e), we store $(1, 0, 1)$ 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 checked, where set' is the cache set for which the conflict is being refined. We search the cached results of form $(set,\ result_{mc},\ conflicts)$ and take an action as follows:

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

Hierarchical refinement: Till now, we have only discussed checking the conflicts generated in a particular cache set. However, cache conflicts generated in different cache sets can be correlated. Consider the program and its corresponding CFG shown in Figure 7. y and z are input variables



```
void f(int y, int z) {
        int i = 0;
        while (i < 100) {
                if (z >= 0)
                     x++;
                else
                     x−−;
                if (y == −2)
                     x−−;
                else
                     x++;
                i++;
        }
}
```

(a) C−code with infeasible path
m0
T
NT
m1
m2
m3
T
NT
m6
m5
m4
(b) CFG without the backedge
m1 and m4 map to the same cache set
m2 and m3 map to the same cache set
T = Taken branch NT = Not Taken branch

Figure 7. Correlated cache conflicts

whose values are unknown. Assume a direct mapped cache and further assume that $m1$ ($m2$) and $m4$ ($m3$) are mapped to same cache set i (j). However, careful examination reveals that either $m1$ or $m2$ (but not both) can be accessed in one invocation of the task. Therefore, either $m1$ will conflict with $m4$ or $m2$ will conflict with $m3$ but both conflicts are impossible to happen in a single invocation. Any AI-based cache analysis will conclude both $m4$ and $m3$ accesses as NC. According to the previous reasoning, we know that only one can be NC but not both. Therefore, we can modify the source code to check a correlated property. Assuming C_i and C_j denote the conflicts in set i and set j respectively, the correlated property should be of the form as follows:

$$C_i \geq N \Rightarrow C_j \leq N-1 \wedge C_j \geq N \Rightarrow C_i \leq N-1 \quad (2)$$

where N is the associativity of the cache. The above property can be called a second order conflict correlation. The idea of conflict correlation can be generalized with n correlated properties. However, increasing the value of n also increases the time complexity of the analysis with less benefit. We plan to study the scalability of hierarchical refinement in future.

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 [13] (for multiple cores). Chronos already employs detailed pipeline modeling (superscalar, out-of-order etc) and modeling of branch prediction. We have also integrated the recently

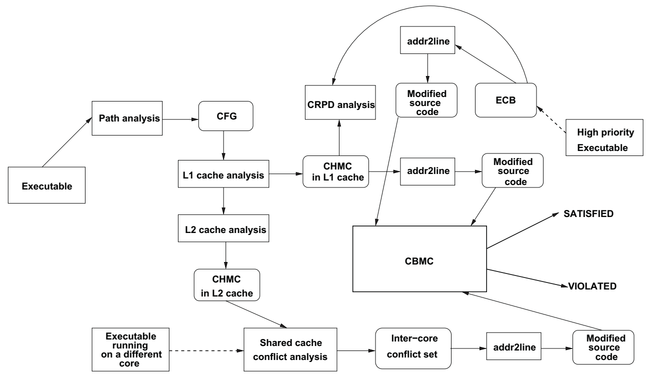


Figure 8. Implementation framework

proposed CRPD analysis ([12] and [11]) as discussed in previous section into chronos. For model checking purposes, we use C bounded model checker (CBMC) [18]. CBMC implements bounded model checking for any C program. In the verification process, CBMC unwinds loop iterations. User can specify the unwinding depth of each loop in the program. CBMC unwinds the loop to the certain depth if a user specified loop bound is detected. If the loop bound specified by the user is not sufficient, CBMC generates an unwind assertion violation. On the other hand, if user has not specified any bound for a loop, CBMC tries to determine the loop bound 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 provided sufficient loop bound as input, so that no unwinding assertion is violated. Figure 8 gives an overall picture of our implementation framework. The figure demonstrates one refinement for each type of conflicts. Chronos employes 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, which is finally passed to CBMC. CBMC either successfully verifies the property or generates a counter example. For a successful verification, we modify the conflict information.

X. Experimental evaluation

We have chosen benchmarks from [4] which are generally used for timing analysis. For evaluation of our framework, 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 2(a). Therefore, model checker refinement pass is used on the tasks from *conflicting task set*. We use a set of randomly 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,

Task	Description	code size (bytes)
qurt	Root computation of quadratic equations	4898
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

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.

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.

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 L2 cache miss latency as 6 cycles and memory 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.

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 [6] on *qurt* and *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 *qurt* and *compress*. On the other hand, *statemate* and *nsichneu* contains very large loops (in terms of code size) as well as they contain multiple paths inside a loop. 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 calls. Nevertheless, if time budget permits, the refinement process can be run longer and thereby provide more opportunities to improve the WCET. Above result demonstrates the potential of our approach even for improving the most fundamental 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. Due to a relatively small number of ECBs, CRPD computed in presence of *qurt* is negligible. Therefore, the model checker refinement pass does not give us any additional reduction in CRPD. On the other hand, 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, 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.

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).

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. For two relatively small tasks in the conflicting task set (*i.e. qurt* and *compress*), we take a 2-way set associative, 2 KB shared L2 cache and for the other two bigger tasks (*i.e. statemate* and *nsichneu*), we take a bigger 4-way associative, 8 KB shared L2 cache, both 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%). Refinement of tasks with potentially more infeasible paths (*i.e. statemate, nsichneu* and *compress*) result in more WCET improvement compared to the rest (*i.e.* qurt).

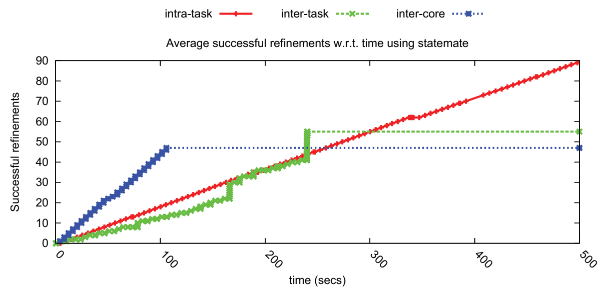
Number of CBMC calls is tolerable (maximum number of calls being only 29). Consequently, except for the two special cases using *qurt* (*i.e.* (*qurt,jfdcint*) and (*qurt,edn*)), all our experiments complete within two minutes.

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

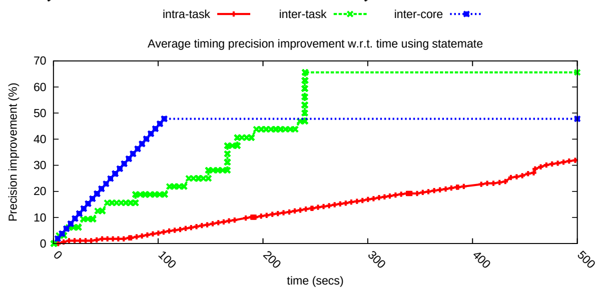
Table III
REFINEMENT OF INTRA-TASK CACHE CONFLICTS. *We use a 4-way associative, 8 KB cache with 32 bytes block size.*

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	512	474	1152	396	65.6%	390.58	52
fir + nsichneu	512	488	828	72	91.3%	304.96	34
fdct + nsichneu	512	397	612	36	94.1%	465.50	52
jfdcint + nsichneu	512	469	792	72	90.9%	554.26	46
edn + nsichneu	512	451	3492	144	95.9%	769.41	64
ndes + nsichneu	512	454	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.*



(a)



(b)

Figure 9. (a) Number of successful refinements by CBMC w.r.t time (b) Timing precision improvement w.r.t. time

Variation with time: In Figure 9(a) and Figure 9(b), we report the sensitivity of our analysis with respect to time. Figure 9(a) depicts the average number of successful refinements by CBMC when *statemate* is used as a conflicting task. On the other hand, Figure 9(b) pictures the average improvement in time precision when the same *statemate* benchmark is used. It clearly shows that given more time for refinement, our results can be improved.

Remarks: In the preceding discussion, we have shown that we can substantially improve different types of cache conflicts using refinement through a model checker. The model checker refinement depends on the type of program being checked and the size of cache for which conflicts are being refined. The refinement pass may considerably improve the WCET/CRPD in presence of infeasible paths, as shown by our experiments. If the cache size is very small compared to the size of a task, number of conflicts are indeed high. Therefore, for such small caches, even the model checker refinement may not generate any improvement. On the other hand, if the cache size is very big compared to the size of task, an AI-based analysis will generate a small number of conflicts. As a result, the model checker refinement will have a negligible improvement (since very few conflicts need to be refined).

Time taken by our framework increases with the number of model checker calls. Therefore, time taken during intra-task and inter-task conflict analysis is higher than the inter-core conflict analysis. We only check very light weight assertions by the model checker. So CBMC applies its own optimizations (*e.g.* slicing) to remove certain paths at the time of assertion checking, reducing the time for a single CBMC call. Finally, by storing the CBMC results (discussed in Section VIII), we can reduce the number of CBMC calls.

XI. CONCLUSION

In this paper, we have proposed a scalable WCET analysis framework using the combination of abstract interpretation

program set	CHMC changed	CHMC refined	$WCET_{base}$ (in cycles)	$WCET_{refined}$ (in cycles)	Improvement(%)	time (secs)	MC calls
cnt + qurt	14	7	212763	113163	46.8%	11.84	2
fir + qurt	9	3	457650	415590	9.2%	12.35	2
fdct + qurt	26	8	8994	7914	12%	40.86	5
jfdcint + qurt	40	9	144956	135866	6.3%	318.46	12
edn + qurt	69	15	175156	168376	4%	383.29	9
ndes + qurt	57	20	146142	127002	13.1%	57.68	5
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 2-way associative, 2 KB cache with 32 bytes block size for cnt and compress. A 4-way associative, 8 KB cache with 32 bytes block size is used for nsichneu and statemate for their relatively large size.*

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.

REFERENCES

[1] M. Lv et. al. Combining abstract interpretation with model checking for timing analysis of multicore software. 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] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.

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

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

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

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

[12] S. Altmeyer, C. Maiza, and J. Reineke. Resilience analysis: tightening the crpd bound for set-associative caches. 2010.

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

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

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

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

[17] Alexander Metzner. Why model checking can improve wcet analysis. In *CAV*, 2004.

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