TIMING ANALYSIS OF CONCURRENT PROGRAMS RUNNING ON SHARED CACHE MULTI-CORES

LI YAN

M.Sc., NUS

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2010

Acknowledgements

I would like to thank my supervisor Professor Tulika Mitra for her professional guidance and her invaluable advice and comments for the thesis during my study.

Especially thanks go to Professor Abhik Roychoudhury for his guidance as well as helpful suggestions.

I would like to thank Vivy Suhendra and Liang Yun who have collaborated with me and have given me continue guidance through the last year.

My acknowledgements go out to all my friends Shi Chenwei, Zhen Hanxiong for their warm-hearted help and beneficial discussions.

Finally, heartful thanks go for my family for their support with heart and soul.

All errors are my own.

Abstract

Memory accesses form an important source of timing unpredictability. Timing analysis of real-time embedded software thus requires bounding the time for memory accesses. Multiprocessing, a popular approach for performance enhancement, opens up the opportunity for concurrent execution. However due to contention for any shared memory by different processing cores, memory access behavior becomes more unpredictable, and hence harder to analyze. In this thesis, we develop a timing analysis method for concurrent software running on multi-cores with a shared instruction cache. We do not handle data cache, shared memory synchronization and code sharing across tasks. The method progressively refines the lifetime estimates of tasks that execute concurrently on multiple cores, in order to estimate potential conflicts in the shared cache. Possible conflicts arising from overlapping task lifetimes are accounted for in the hit-miss classification of accesses to the shared cache, to provide safe execution time bounds. We show that our method produces tighter worst-case response time (WCRT) estimates than existing shared-cache analysis on a real-world embedded application.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

Caches are commonly utilized to enhance performance in embedded computing systems. Cache management is handled by hardware, lending transparency that, while desirable to ease programming effort, leads to unpredictable timing behavior for real-time software. Worst-case execution time (WCET) analysis for real-time applications requires that the access time for each memory access is safely bounded, in order to guarantee that timing constraints are met. With the presence of performance-enhancing features in today's systems, this can be a challenging feat. One such feature is multiprocessing, which opens the opportunity for concurrent execution and memory sharing, and at the same time introduces the problem of estimating the impact of resource contention.

A lot of research efforts have been invested in modeling dynamic cache behavior in single-processing systems. In the context of instruction caches, a particularly popular technique is *abstract interpretation* [2, 24] which introduces the concept of *abstract cache states* to represent complete possible cache contents at a given program point, enabling subsequent *Cache Hit-Miss Classification* of memory accesses into 'Always Hit', 'Always Miss', 'Persistent/First Miss', and 'Not Classified'. The latency corresponding to each of these situations can then be incorporated in the WCET calculation.

Hardy and Puaut [8] further extend the abstract interpretation method to safely produce worst-case hit/miss access classification in multi-level set-associative caches. They address a main weakness in the previous cache hierarchy analysis [14], where unclassified L1 hit/miss results have been conservatively interpreted as Always Miss in the WCET estimation. However, in the subsequent L2 analysis, this interpretation will lead to the assumption that L2 is always accessed for that reference. On set-associative caches with a Least Recently Used replacement policy, the abstract cache state update may then arrive at an over-optimistic estimation of the age of the reference in L2, leading to unsafe classification of certain actual L2 misses as L2 hits. Hardy and Puaut's approach rectifies this problem by introducing the concept of *Cache Access Classification* to model the propagation of access from a cache level to the level above it: Always, Never, or Uncertain. When a reference cannot be classified as Always Miss nor Always Hit at L1, the access to L2 is Uncertain for that reference. For such accesses, the L2 analysis joins the abstract cache state resulting from an actual access and the abstract cache state corresponding to no access. Considering both these cases avoids overlooking the situation that may give rise to an execution time higher than the estimated WCET.

As multi-cores are increasingly adopted in high-performance embedded systems, the design choices for cache hierarcy also expand. While each L1 cache is typically required to remain closely and privately adjoined to each processing core in order to provide single-cycle latency, letting the multiple cores share a common L2 cache is seen as beneficial in situations where memory usage is not always balanced across cores. When L2 cache is shared, a core will be able to occupy a larger share during its busy period, and relinquish the space to be used by other cores when it is idle. This architecture is implemented for example in Power5 dual-core chip [20], XBox360's Xenon processor [5], and Sun UltraSPARC T1 [22]. Certainly, the analysis effort required for this configuration is also more complex, as memory contention across the multiple cores significantly affects the shared cache behaviour. In particular, accesses to the L2 cache originating from different cores may conflict in the shared cache. Thus, isolated cache analysis of each task that does not account for this effect will not safely bound the execution time of the task.

The only technique in literature that has addressed shared-cache analysis so far is one by Yan and Zhang [26]. Their approach first applies abstract interpretation to tasks independently and produce the hit-miss classification at both L1 and L2. In the next step, conflicting cache lines across the multiple processing cores are identified. If these lines were previously categorized as hits, they will be converted to misses. In this approach, all tasks executing in a different core than the one under consideration are treated as potential conflicts regardless of their actual execution time frame, thus the resulting estimate is not tight. We also note that their work has not addressed the problem with conservative multi-level cache analysis observed by [8] as elaborated above, thus it will be prone to unsafe estimation when applied to set-associative caches. This concern, however, is orthogonal to the issues arising from cache sharing.

Motivated by this situation, this thesis proposes a tight and safe multi-level cache analysis for multi-cores that include a shared L2 cache. Our method includes progressively tightening lifetime analysis of tasks that execute concurrently across the multiple cores, in order to identify potential contention in the shared cache. Possible conflicts arising from overlapping task lifetimes are then accounted for in the hit-miss classification of accesses to the shared cache.

1.2 Organization of the Thesis

We introduce some related fundamental concepts related to timing analysis of multi-cores with a shared instruction cache in Section 2 and literature review in Section 3. From section 4, we list our primary contributions devoted to timing analysis for concurrent software running on multi-cores with a shared instruction cache. Following that, our analysis framework is illustrated in Section 5. Estimation results are shown to validate our approach later in Section 6. Finally, the thesis proposes the future work in Section 7 and concludes in Section 8.

2 Background

Static analysis of programs to give guarantees about execution time is a difficult problem. For sequential programs, it involves finding the longest feasible path in the program's control flow graph while considering the timing effects of the underlying processing element. For concurrent programs, we also need to consider the time spent due to interaction and resource contention among the program threads.

What makes static timing analysis difficult? Clearly it is the variation in the execution time of a program due to different inputs, different interaction patterns (for concurrent programs) and different micro-architectural states. These variations manifest in different ways, one of the major variations being the time for memory accesses. Due to the presence of caches in processing elements, a certain memory access may be cache hit or miss in different instances of its execution. Moreover, if caches are shared across processing elements as in shared cache multi-cores, one program thread may have constructive or destructive effect on another in terms of cache hits/misses. This makes the timing analysis of concurrent programs running on shared-cache multi-cores a challenging problem. We address this problem in our work. Before that, we will give some background on Abstract Interpretation, Message Sequence Charts (MSCs) and Message Sequence Graphs (MSGs) — our system model for describing concurrent programs. In doing so, we also introduce our case study with which we have validated our approach. We conclude this section by detailing our system architecture — the platform on which the concurrent application is executed.

2.1 Abstract Interpretation

In the context of instruction caches, a particularly popular technique is *abstract interpretation* [2, 24] which introduces the concept of *abstract cache states* to represent complete possible cache contents at a given program point, enabling subsequent *Cache Hit-Miss Classification* of memory accesses into 'Always Hit', 'Always Miss', 'Persistent/First Miss', and 'Not Classified'. The latency corresponding to each of these situations can then be incorporated in the WCET calculation.

This approach works as follows [14, 21]:

Assume a two-way set-associative cache with four cache lines and Least Recently Used (LRU) replacement policy.

Firstly, the concrete cache state (CCS) given a program point is defined. The concrete cache state is the exact result cache state for a given program point. In this way, each concrete cache state represents a real cache state.

Next, the abstract cache state (ACS) given a program point is defined. Obviously, if we use CCS to do cache analysis, the possible cache states probably will grow exponentially due to conditional executions or loops and thus renders the problem to be unsolvable within finite time. To avoid this, an abstract cache state is defined so that just one state can gather all possible occurring concrete states for each program point.

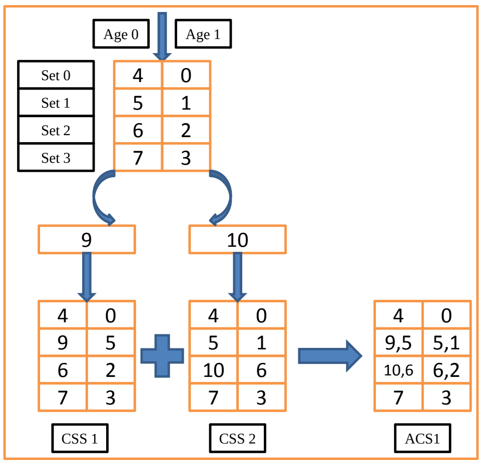


Figure 1: An example of CCS and ACS.

Figure 1 is an example of CCS and ACS. It shows a conditional execution. Program line 9 is then-part while program line 10 is else-part. After the control flow joins again, both CCS' (that is CSS1 and CSS2 in the figure) represent possible cache states and have to be considered for the remainder of program execution. It also depicts the corresponding ACS (that is ACS1). There is only one output ACS containing sets of program lines that may be cached at this point of execution. In effect, the output CCS' are merged into this output ACS. Merging conserves space but reduces the amount of information. For example, the output ACS does not show that either program lines 9 or 10 can be cached.

To catch as more information as possible, abstract semantics should consist of an abstract domain and a set of proper abstract semantic functions, so called transfer functions, for the program statements computing over the abstract domain. They describe how the statements transform abstract data. They must be monotonic to guarantee termination. An element of the abstract domain represents sets of elements of the concrete domain. The subset relation on the sets of concrete states determines the complete partial order of the abstract domain. The partial order on the abstract domain corresponds to precision, i. e., quality of information. To combine abstract values, a join operation is needed. In our case this is the least upper bound operation, t, on the abstract domain, which also defines the partial order on the abstract domain. This operation is used to combine information stemming from different sources, e. g. from several possible control flows into one program point.

We have three types of operations on ACS defined as following. To make it clearly interpreted, we just assume LRU as the cache replacement strategy. However, it can be extended to other cache replacement policies such as FIFO, pseudo-LRU and so on which are explained specifically in [9]. Since each set is independently updated when LRU cache replacement policy is adopted, we illustrate operations of cache state using only one set of cache for simplicity. Further, we assume a 4-way cache.

- **Must Analysis:** Must analysis determines the set of all memory blocks that are guaranteed to be present in the cache at a given program point. This analysis is similarly to do set intersection of multiple abstract cache states where the position of a memory block is an upper bound of its age among all the abstract cache states.

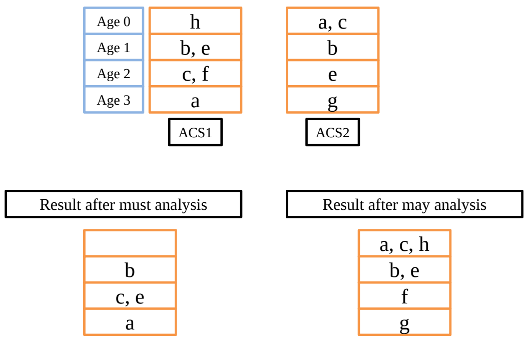


Figure 2: An example of must and may analysis.

- **May Analysis:** The may analysis determines all memory blocks that may be in the cache at a given program point. It is used to guarantee the absence of a memory block in the cache. This analysis is similarly to do set unions of abstract cache state where the position of a memory block is a lower bound of its age among all the abstract cache states. Figure 2 is an example of must and may analysis.

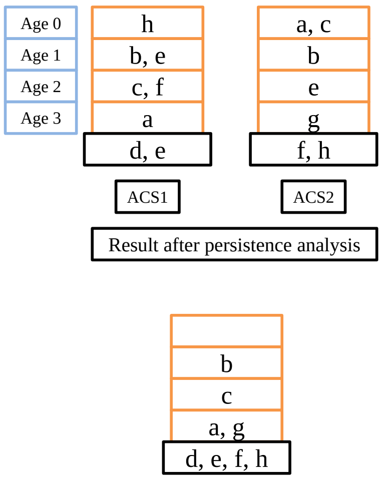


Figure 3: An example of persistence analysis.

- **Persistence Analysis:** This analysis is used to improve the classification of memory references. It collects the set of all memory blocks that are never evicted from the cache after the first reference, which means that a first execution of a memory reference may result in either a hit or a miss, but all non-first executions will result in hits. This analysis is similarly to do unions of abstract cache states where the position of a memory block is a upper bound of its age among all the abstract cache states. Additionally, we assume a virtual cache line with the maximal age in a set of cache which holds those cache lines that could once have been removed from the cache. Figure 3 is an example of persistence analysis.

The cache analysis results can be used to classify the memory blocks in the following manner. Each instruction can be classified into AH, AM, PS or NC.

- **Always Hit (AH)** If a memory block is present in the ACS corresponding to must analysis, its references will always result in cache hits.
- **Always Miss (AM)** If a memory block is not present in the ACS corresponding to may analysis, its references are guaranteed to be cache misses.
- **Persistence (PS)** If a memory block is guaranteed to be present not in the virtual line after persistence analysis, it will never to be evicted from the cache. Therefore, it can be classified as persistent where the second and all further executions of the memory reference will always be cache hits.
- **Not Classified (NC)** The memory reference cannot be classified as either AH, AM, or PS.

2.2 Message Sequence Charts

Our system model consists of a concurrent program visualized as a graph, each node of which is a Message Sequence Chart or MSC [1] . A MSC is a variant of an UML sequence diagram with a formal semantics and is a modeling notation that emphasizes the inter-process interaction, allowing us to exploit its structure in our timing analysis. The individual processes in the MSC appear as vertical lines. Interactions between the processes are shown as horizontal arrows across vertical lines. The computation blocks within a process are shown as "tasks" on the vertical lines.

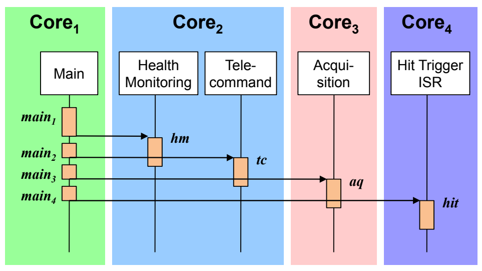


Figure 4: A simple MSC and a mapping of its processes to cores.

Figure 4 shows a simple MSC with five processes (vertical lines). It is in fact drawn from our DEBIE case study, which models the controller for a space debris management system. The five processes are mapped on to four cores. Each process is mapped to a unique core, but several processes may be mapped to the same core (e.g., *Health-monitoring* and *Telecommand* processes are mapped to core 2 in Figure 4). Each process executes a sequence of "tasks" shown via shaded rectangles (e.g., $main_1$, hm, tc are tasks in Figure 4). Each task is an arbitrary (but terminating) sequential program in our setting and we assume there is no code sharing across the tasks.

Semantically, an MSC denotes a set of tasks and prescribes a partial order over these tasks. This partial order is the transitive closure of (a) the total order of the tasks in each process (time flows from top to bottom in each process), and (b) the ordering imposed by the send-receive of each message (the send of a message must happen before its receive). Thus in Figure 4, the tasks in the *Main* process execute in the sequence $main_1, main_2, main_3, main_4$. Also, due to message send-receive ordering, the task $main_1$ happens before the task hm. However, the partial ordering of the MSC allows tasks hm and tc to execute concurrently.

We assume that our concurrent program is executed in a static priority-driven non-preemptive fashion. Thus, each process in an MSC is assigned a unique static priority. The priority of a task is the priority of the process it belongs to. If more than one processes are mapped to a processor core, and there are several tasks contending for execution on the core (such as the tasks hm and tc on core

2 in Figure 4), we choose the higher priority task for execution. However, once a task starts execution, it is allowed to complete without preemption from higher priority tasks.

2.3 Message Sequence Graph

A Message Sequence Graph (MSG) is a finite graph where each node is described by an MSC. Multiple outgoing edges from a node in the MSG represent a choice, so that exactly one of the destination charts will be executed in succession. While an MSC describes a single scenario in the system execution, an MSG describes the control flow between these scenarios, allowing us to form a complete specification of the application.

To complete the description of MSG, we need to give a meaning to MSC concatenation. That is, if M_1, M_2 are nodes (denoting MSCs) in an MSG, what is the meaning of the execution sequence $M_1, M_2, M_1, M_2, \ldots$? We stipulate that for a concatenation of two MSCs say $M_1 \circ M_2$, all tasks in M_1 must happen before any task in M_2. In other words, it is as if the participating processes synchronize or hand-shake at the end of an MSC. In MSC literature, it is popularly known as synchronous concatenation [3].

2.4 DEBIE Case Study

Our case study consists of DEBIE-I DPU Software [7], an in-situ space debris monitoring instrument developed by Space Systems Finland Ltd. The DEBIE instrument utilizes up to four sensor units to detect particle impacts on the spacecraft. As the system starts up, it performs resets based on the condition that precedes the boot. After initializations, the system enters the Standby state, where health monitoring functions and housekeeping checks are performed. It may then go into the Acquisition mode, where each particle impact will trigger a series of measurements, and the data are classified and logged for further transmission to the ground station. In this mode too, the Health Monitoring

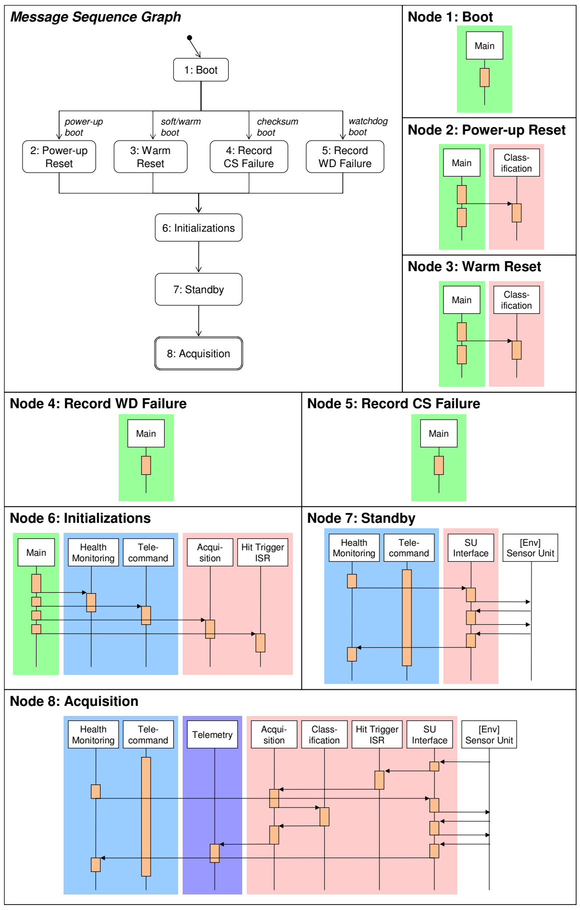


Figure 5: A multi-core architecture with shared cache.

process continues to periodically monitor the health of the instrument and to run housekeeping checks.

The MSG for the DEBIE case study (with different colors used to show the mapping of the processes to different processor cores) is shown in Figure 5. This MSG is acyclic. For MSGs with cycles, the number of times each cycle can be executed needs to be bounded for worst-case response time analysis.

2.5 System architecture

The generic multi-core architecture we target here is quite representative of the current generation multi-core systems as shown in Figure 6. Each core on chip has its own private L1 instruction cache and a shared L2 cache that accommodates instructions from all the cores. In this work, our focus is on instruction memory accesses and we do not model the data cache. We assume that the data memory references do not interfere in any way with the L1 and L2 instruction caches modeled by us (they could be serviced from a separate data cache that we do not model).

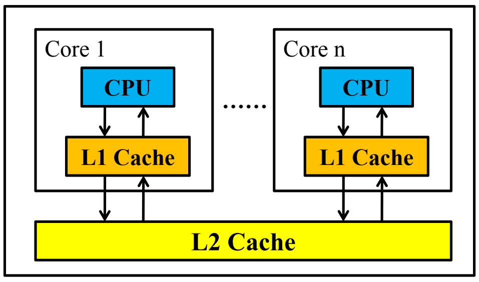


Figure 6: A multi-core architecture with shared cache.

3 Literature Review

There have been a lot of research efforts in modeling cache behavior for WCET estimation in single-core systems. A widely adopted technique is the abstract interpretation ([2, 24]) which also forms the foundation to the framework presented in this thesis.

Mueller [15] extends the technique for multi-level cache analysis; Hardy and Puaut [8] further adjust the method with a crucial observation to produce safe estimates for set-associative caches. Other proposed methods that attempt exact classification of memory accesses for private caches include data-flow analysis [15], integer linear programming [12] and symbolic execution [13].

Cache analysis for multi-tasking systems mostly revolves around a metric called *cache-related preempted delay (CRPD)*, which quantifies the impact of cache sharing on the execution time of tasks in a preemptive environment. CRPD analysis typically computes cache access footprint of both the preempted and preempting tasks ([10, 25, 16]). The intersection then determines cache misses incurred by the preempted task upon resuming execution due to conflict in the cache. Multiple process activations and preemption scenarios can be taken into account, as in [21]. A different perspective in [23] considers WCRT analysis for customized cache, specifically the prioritized cache, which reduces inter-task cache interference.

In multiprocessing systems, tasks in different cores may execute in parallel while sharing memory space in the cache hierarchy. Due to the complexity involved in static analysis of multiprocessors, time-critical systems often opt not to exploit multiprocessing, while non-critical systems generally utilize measurement-based performance analysis. Tools for estimating cache access time are presented, among others, in [19], [6] and [11]. It has also been proposed to perform static scheduling of memory accesses so that they can be factored in to achieve reliable WCET analysis on multiprocessors [18].

The only technique in literature that has addressed inter-core shared-cache analysis so far is the one proposed by Yan and Zhang [26]. Their approach accounts for inter-core cache contention by detecting accesses across cores which map to the same set in the shared cache. They treat all tasks executing in a different core than the one under consideration as potential conflicts regardless of their actual execution time frames; thus the resulting estimate is highly pessimistic. We also note that their work has not addressed the problem with multi-level cache analysis observed by [8] (a "non-classified" access in L1 cache cannot be safely assumed to always access L2 cache in the worst case) and will be prone to unsafe estimation when applied to set-associative caches. This concern, however, is orthogonal to the issues arising from cache sharing. Our proposed analysis is able to obtain improved estimates by exploiting the knowledge about interaction among tasks in the multiprocessor.

4 Contributions

Based on the literature review presented, our contributions in the thesis are as following.

- The first contribution we make in this thesis is that we take into account the execution interval of tasks to minimize the overestimation of interferences in the shared cache between pairs of tasks from different cores and we validate our estimation with experiments. We compare our method with the only approach [26] in literature. And the only approach to model the conflicts for L2 cache blocks among the cores is the following. Let T be the task running on core 1 and T' be the task running on core 2. Also let $M_1, \ldots, M_X$ $(M'_1, \ldots, M'_Y)$ be the set of memory blocks of thread T (T') mapped to a particular cache set C in the shared L2 cache. Then we simply deduce that all the accesses to memory blocks $M_1, \ldots, M_X$ and $M'_1, \ldots, M'_Y$ will be misses in L2 cache. However, we observed that if a pair of tasks from different cores cannot overlap in terms of execution interval, they are not able to affect each other in terms of conflict misses and thus we can reduce the number of estimated conflict misses in the shared cache.
- Another contribution in this thesis is that we embrace set-associative caches in our analysis as opposed to only direct mapped caches and this creates additional opportunities for improving the timing estimation. For simplicity, direct-mapped cache is often assumed to be adopted. However, this assumption is not practical since set-associative cache is prevalent.

In summary, we develop a timing analysis method for shared cache multicores that enhances the state-of-the-art approach.

5 Approach

5.1 Overview

In this section, we present an overview of our timing analysis framework for concurrent applications running on a multi-core architecture with shared caches. For ease of illustration, we will throughout use the example of a 2-core architecture. However, our method is easily scalable to any number of cores as will be shown in the experimental evaluation. As we are analyzing a concurrent application, our goal is to estimate the Worst Case Response Time (WCRT) of the application.

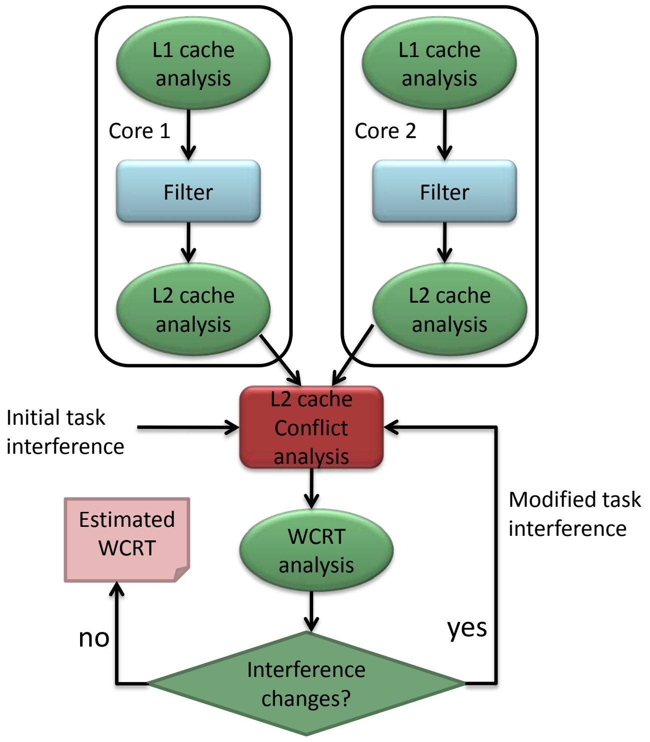


Figure 7: Our Analysis Framework

Figure 7 shows the workflow of our timing analysis framework. First, we perform the L1 cache hit/miss analysis for each task mapped to each core independently. As we assume a non-preemptive system, we can safely analyze the cache effect of each task separately even if multiple tasks are mapped to the same processor core. For preemptive systems, we need to include cache-related preemption delay analysis ([10, 25, 16, 21]) in our framework.

The filter at each core ensures that only the memory accesses that miss in the L1 cache are analyzed at the L2 cache level. Again, we first analyze the L2 cache behavior for each task in each core independently assuming that there is no conflict from the tasks in the other cores. Clearly, this part of the analysis does not model any multi-core aspects and we do not propose any new innovations here. Indeed, we employ the multi-level non-inclusive instruction cache modeling proposed recently [8] for intra-core analysis.

The main challenge in safe and accurate execution time analysis of a concurrent application is the detection of conflicts for shared resources. In our target platform, we are modeling one such shared resource: the L2 cache. A first approach to model the conflicts for L2 cache blocks among the cores is the following. Let T be the task running on core 1 and T' be the task running on core 2. Also let $M_1, \ldots, M_X$ ($M'_1, \ldots, M'_Y$) be the set of memory blocks of thread T (T') mapped to a particular cache set C in the shared L2 cache. Then we simply deduce that all the accesses to memory blocks $M_1, \ldots, M_X$ and $M'_1, \ldots, M'_Y$ will be misses in L2 cache. Indeed, this is the approach followed by the only shared L2 cache analysis proposed in the literature [26].

A closer look reveals that there are multiple opportunities to improve the conflict analysis. The first and foremost is to estimate and exploit the lifetime information for each task in the system, which will be discussed in detail in the following. If the lifetimes of the tasks T and T' (mapped to core 1 and core 2, respectively) are completely disjoint, then they cannot replace each other's memory blocks in the shared cache. In other words, we can completely bypass shared cache conflict analysis among such tasks.

The difficulty lies in identifying the tasks with disjoint lifetimes. It is easy to recognize that the partial order prescribed by our MSC model of the concurrent application automatically implies disjoint lifetimes for some tasks. However, accurate timing analysis demands us to look beyond this partial order and identify additional pairs of tasks that can potentially execute concurrently according to the partial order, but whose lifetimes do not overlap (see Section 5.2 for an example). Towards this end, we estimate a conservative lifetime for each task by exploiting the Best Case Execution Time (BCET) and Worst Case Execution Time (WCET) of each task along with the structure of the MSC model. Still the problem is not solved as the task lifetime (i.e., BCET and WCET estimation) depends on the L2 cache access times of the memory references. To overcome this cyclic dependency between the task lifetime analysis and the conflict analysis for shared L2 cache, we propose an iterative solution.

The first step of this iterative process is the conflict analysis. This step estimates the additional cache misses incurred in the L2 cache due to inter-core conflicts. In the first iteration, conflict analysis assumes very preliminary task interference information — all the tasks (except those excluded by MSC partial order) that can potentially execute concurrently will indeed execute concurrently. However, from the second iteration onwards, it refines the conflicts based on task lifetime estimation obtained as a by-product of WCRT analysis component. Given the memory access times from both L1 and L2 caches, WCRT analysis first computes the execution time bounds of every task, represented as a range. These values are used to compute the total response time of all the tasks considering dependencies. The WCRT analysis also infers the interference relations among tasks: tasks with disjoint execution intervals are known to be non-interfering, and it can be guaranteed that their memory references will not conflict in the shared cache. If the task interference has changed from the previous iteration, the modified task interference information is presented to the conflict analysis component for another round of analysis. Otherwise, the iterative analysis terminates and returns the WCRT estimate. Note the feedback loop in Figure 7 that allows us to improve the lifetime bounds with each iteration of the analysis.

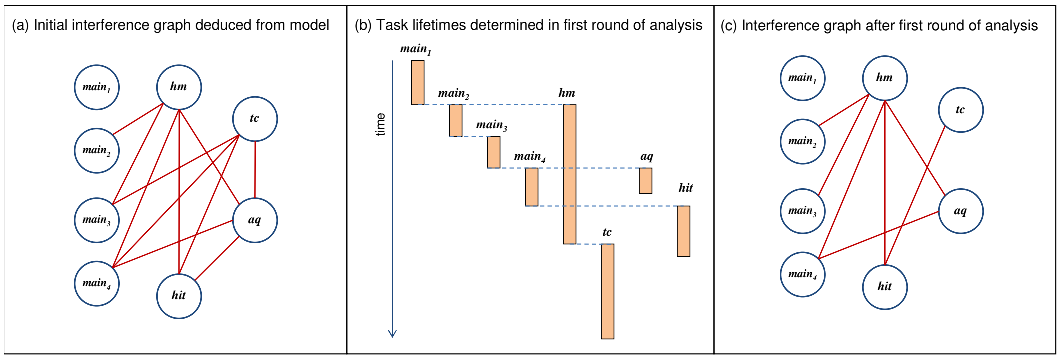

(a) Initial interference graph deduced from model
(b) Task lifetimes determined in first round of analysis
(c) Interference graph after first round of analysis
main1
main2
main3
main4
hm
tc
aq
hit
time

Figure 8: The working of our shared-cache analysis technique on the example given in Figure 4

5.2 Illustration

We illustrate our iterative analysis framework on the MSC depicted in Figure 4. Initially, the only information available are (1) the dependency specified in the model, and (2) the mapping of tasks to cores. Two tasks t, t' are known *not* to interfere if either (1) t' depends on t as per the MSC partial order, or (2) t and t' are mapped to the same core (by virtue of the non-preemptive execution).

We can thus sketch the initial interference relation among tasks in an *interference graph* as shown in Figure 8(a). Each node of the graph represents a task, and an edge between two nodes signifies potential conflict between the tasks represented by the nodes. This is the input to the cache conflict analysis component (Figure 7), which then accounts for the perceived inter-task conflicts and accordingly adjusts L2 cache access time of conflicting memory blocks.

In the next step, we compute BCET and WCET values for each task. These values are used in the WCRT analysis to determine task lifetimes. Figure 8(b) visualizes the task lifetimes after the analysis for this particular example. Here, time is depicted as progressing from top to bottom, and the duration of task execution is shown as vertical bar stretching from the time it starts to the time it completes.

The overlap between the lifetimes of two tasks signifies the potential that they may execute concurrently and may conflict in the shared cache. Conversely, the absence of overlap in these inferred lifetimes tell us that some tasks are well separated (e.g., *aq* and *tc*) so that it is impossible for them to conflict in the shared cache. For instance, here *tc* starts later than *hm* on the same core, and thus has to wait until *hm* finishes execution. By that time, most of the other tasks have finished their execution and will not conflict with *tc*. Based on this information, our knowledge of task interaction can be refined into the interference graph shown in Figure 8(c). This information is fed back as input to the cache conflict analysis, where some of the previously assumed evictions in the shared cache can now be safely ruled out.

Our analysis proceeds in this manner iteratively. The initial conservative assumption of task interferences is refined over the iterations. In the next section, we provide detailed description of the analysis components and show that our iterative analysis is guaranteed to terminate.

5.3 Analysis Components

The first step of our analysis framework is the independent cache analysis for each core (see Figure 7). As mentioned before, we use the multi-level non-inclusive cache analysis proposed by Hardy and Puaut [8] for this step. However, some background on this intra-core analysis is required to appreciate our shared cache conflict analysis technique. Hence, in the next subsection, we provide a quick overview of the intra-core cache analysis.

5.3.1 Intra-Core Cache Analysis

The intra-core cache analysis step employs abstract interpretation method [24] at both L1 and L2 cache levels. The additional step for multi-level caches is the filter function (see Figure 7) that eliminates the L1 cache hits from accessing the L2 cache. The L1 cache analysis computes the three different abstract cache states (ACS) at every program point within a task [24]. In this thesis, we consider LRU replacement policy, but the cache analysis can be extended for other replacement

Table 1: Filter function

L1 Classification	**L2 Access**
Always Hit (AH)	Never (N)
Always Miss (AM)	Always (A)
Not Classified (NC)	Uncertain (U)

polices as shown in [9].

As described in Section 2.1, we classify each instruction into AH, AM, PS and NC.

For a Persistent (PS) memory block, we further classify it as Always Miss (AM) for its first reference and Always Hit (AH) for the rest of the references. Once the memory blocks have been classified at L1 cache level, we proceed to analyze them at L2 cache level. But before that, we need to apply the filter function that eliminates L1 cache hits from further consideration [8]. The filter function is shown in Table 1.

A reference classified as always hit will never access L2 cache ("Never") whereas a reference classified as always miss will always access L2 cache ("Always"). The more complicated scenario is with the non-classified references. [8] has shown that it is unsafe to assume that a non-classified reference will always access L2 cache. Instead, its status is set to "Uncertain" and we consider both the scenarios (L2 access and no L2 access) in our analysis for such references.

The intra-core L2 cache analysis is identical to L1 cache analysis except that (a) a reference with "Never" tag is ignored, i.e., it does not update abstract cache states, and (b) a reference r with "Uncertain" tag creates two abstract cache states (one updated with r and the other one not updated with r) that are "joined" together.

The pseudo-code of the intra-core cache analysis is shown in Figure 9 (for L1) and Figure 10 (for L2).

Algorithm 1: L1 cache analysis for a task t

```
1  AnalyseScopeL1 ($main\_procedure$, $empty\_ACS$, $empty\_ACS$) ;

   Function AnalyseScopeL1 ($Sc$, $in\_must$, $in\_may$)
2  $ACS\_in\_must(Sc.entry) := in\_must$;
3  $ACS\_in\_may(Sc.entry) := in\_may$;
4  foreach basic block $b$ in the topological order of $Sc$'s CFG do
5      if $b$ has more than one incoming edges then
6          $ACS\_in\_must(b)$ := IntersectMaxAge ($\{ACS\_out\_must(b') \mid b'$ is a predecessor of $b\}$) ;
7          $ACS\_in\_may(b)$ := UnionMinAge ($\{ACS\_out\_may(b') \mid b'$ is a predecessor of $b\}$) ;
8      if $b$ abstracts a loop $L$ then
9          $(ACS\_tr\_must, ACS\_tr\_may)$ := AnalyseScopeL1 ($L$, $ACS\_in\_must(b)$, $ACS\_in\_may(b)$) ; // first iteration
10         $(ACS\_out\_must(b), ACS\_out\_may(b))$ := AnalyseScopeL1 ($L$, $ACS\_tr\_must(b)$, $ACS\_tr\_may(b)$) ; // subsequent
11     else
12         $ACS\_curr\_must := ACS\_in\_must(b)$;
13         $ACS\_curr\_may := ACS\_in\_may(b)$;
14         foreach reference $r$ in $b$ in execution order do
15             if $r \in ACS\_curr\_must$ then $CHMC_{r,1} := AH$;  $CAC_{r,2} := N$;
16             else if $r \notin ACS\_curr\_may$ then $CHMC_{r,1} := AM$;  $CAC_{r,2} := A$;
17             else $CHMC_{r,1} := NC$;  $CAC_{r,2} := U$;
18             $ACS\_curr\_must$ := Update ($ACS\_curr\_must$, $r$) ;
19             $ACS\_curr\_may$ := Update ($ACS\_curr\_may$, $r$) ;
20         $ACS\_out\_must(b) := ACS\_curr\_must$;
21         $ACS\_out\_may(b) := ACS\_curr\_may$;
22     if $b$ contains a function call to procedure $P$ then
23         $(ACS\_out\_must(b), ACS\_out\_may(b))$ := AnalyseScopeL1 ($P$, $ACS\_out\_must(b)$, $ACS\_out\_may(b)$) ;
24 return $(ACS\_out\_must(Sc.exit), ACS\_out\_may(Sc.exit))$;
```

Figure 9: Intra-core cache analysis for L1

Algorithm 2: L2 cache analysis for a task t

```
1  AnalyseScopeL2 ($main\_procedure$, $empty\_ACS$, $empty\_ACS$) ;

   Function AnalyseScopeL2 ($Sc$, $in\_must$, $in\_may$)
2  $ACS\_in\_must(Sc.entry) := in\_must$;
3  $ACS\_in\_may(Sc.entry) := in\_may$;
4  foreach basic block $b$ in the topological order of $Sc$'s CFG do
5      if $b$ has more than one incoming edges then
6          $ACS\_in\_must(b)$ := IntersectMaxAge ($\{ACS\_out\_must(b') \mid b'$ is a predecessor of $b\}$) ;
7          $ACS\_in\_may(b)$ := UnionMinAge ($\{ACS\_out\_may(b') \mid b'$ is a predecessor of $b\}$) ;
8      if $b$ abstracts a loop $L$ then
9          $(ACS\_tr\_must, ACS\_tr\_may)$ := AnalyseScopeL2 ($L$, $ACS\_in\_must(b)$, $ACS\_in\_may(b)$) ; // first iteration
10         $(ACS\_out\_must(b), ACS\_out\_may(b))$ := AnalyseScopeL2 ($L$, $ACS\_tr\_must(b)$, $ACS\_tr\_may(b)$) ; // subsequent
11     else
12         $ACS\_curr\_must := ACS\_in\_must(b)$;
13         $ACS\_curr\_may := ACS\_in\_may(b)$;
14         foreach reference $r$ in $b$ where $CAC_{r,2} \neq N$ do
15             if $r \in ACS\_curr\_must$ then $CHMC_{r,2} := AH$;
16             else if $r \notin ACS\_curr\_may$ then $CHMC_{r,2} := AM$;
17             else $CHMC_{r,2} := NC$;
18             if $CAC_{r,2} = U$ then
19                 $ACS\_curr\_must$ := IntersectMaxAge ($\{ACS\_curr\_must$, Update ($ACS\_curr\_must$, $r$) $\}$) ;
20                 $ACS\_curr\_may$ := UnionMinAge ($\{ACS\_curr\_may$, Update ($ACS\_curr\_may$, $r$) $\}$) ;
21             else
22                 $ACS\_curr\_must$ := Update ($ACS\_curr\_must$, $r$) ;
23                 $ACS\_curr\_may$ := Update ($ACS\_curr\_may$, $r$) ;
24         $ACS\_out\_must(b) := ACS\_curr\_must$;
25         $ACS\_out\_may(b) := ACS\_curr\_may$;
26     if $b$ contains a function call to procedure $P$ then
27         $(ACS\_out\_must(b), ACS\_out\_may(b))$ := AnalyseScopeL2 ($P$, $ACS\_out\_must(b)$, $ACS\_out\_may(b)$) ;
28 return $(ACS\_out\_must(Sc.exit), ACS\_out\_may(Sc.exit))$;
```

Figure 10: Intra-core cache analysis for L2

```
1 foreach task $t$ do
2     foreach reference $r$ in task $t$ where $CAC_{r,2} \neq N$ AND $CHMC_{r,2} = AH$ do
3         foreach task $u$ potentially interfering with $t$'s execution do
4             $CfSet := \{r' \mid r' \in u$ AND $CAC_{r',2} \neq N$ AND $r'$ maps to the same L2 cache set as $r\}$;
5             if $CfSet \neq \emptyset$ then $CHMC_{r,2} := NC$;
```

Algorithm 3: L2 conflict miss analysis

Figure 11: L2 cache conflict analysis

5.3.2 Cache Conflict Analysis

Cache block containing single instruction Shared L2 cache conflict analysis is the central component of our framework. It takes in two inputs, namely the task interference graph (see Figure 8) generated by the WCRT analysis step and the abstract cache states plus the classification corresponding to L2 cache analysis for each core. If accurate task interference information is not available (that is, in the first iteration of our method), all tasks executing on a different core than the task under consideration (and are not dependent according to the partial order of MSC) are assumed to be potentially conflicting. The goal of this step is to identify all potential conflicts among the memory blocks from the different cores due to sharing of the L2 cache.

Let T be a task executing on core 1 and can potentially conflict with the set of tasks $\mathcal{T}'$ executing on core 2 according to the task interference graph. Now let us investigate the impact of the L2 memory accesses of $\mathcal{T}'$ on the L2 cache hit/miss status of the memory blocks of T. First, we notice that if a memory reference of $\mathcal{T}'$ is always hit in the L1 cache, it does not touch the L2 cache. Such memory reference will not have any impact on task T. So we are only concerned with the memory references of $\mathcal{T}'$ that are guaranteed to access the L2 cache ("Always") or may access the L2 cache ("Uncertain"). For each cache set C in the L2 cache, we collect the set of unique memory blocks $\mathcal{M}(C)$ of $\mathcal{T}'$ that map to cache set C and can potentially access the L2 cache (i.e., tagged with "Always" or "Uncertain").

If a memory block m of task T has been classified as "Always Miss" or "Non-Classified" for L2 cache, the impact of interfering task set $\mathcal{T}'$ cannot downgrade this classification. Hence, we only need to consider the memory blocks of task T that have been classified as "Always Hit" for L2 cache. Let m be one such memory reference of T that has been classified as "Always Hit" in the L2 cache and it maps to cache set C. If $\mathcal{M}(C) \neq \emptyset$, then the memory accesses from interfering tasks can potentially evict m from the L2 cache. So we change the classification of m from "Always Hit" to "Non-Classified". Note that actual task interaction at runtime will determine whether the eviction indeed occurs, thus the access is regarded as "Non-Classified" rather than "Always Miss". The pseudo-code of the cache conflict analysis is shown in Figure 11.

Handling large cache blocks We have so far implicitly assumed that a memory block contains only one instruction. In reality, a memory block contains multiple instructions (specially for L2 caches) so as to exploit spatial locality. These multi-instruction cache blocks introduce additional complications into our timing analysis. Let m be a 16-byte memory block of task T containing four 32-bit instructions $I1, I2, I3, I4$. Further, m is completely contained within a basic block in the program corresponding to task T. In a sequential execution where there is no conflict from the other tasks, we are only concerned about categorizing the cache hit/miss status of instruction $I1$ in memory block m. This is because, execution of $I1$ will bring in the entire memory block m to the cache and hence $I2, I3, I4$ are guaranteed to be cache hits. However, in a concurrent execution, the situation is very different. A memory access from an interfering task can evict the memory block m from the cache between the execution of $I1$ and $I2$. In this case, when $I2$ is fetched, it can result in a cache miss. In other words, in a concurrent execution, we can no longer work at the granularity of memory blocks while computing cache hit/miss classification.

We handle large cache blocks (i.e., blocks with more than one instructions) in the following manner. First, we notice that if a memory block has been classified as "Always Hit" even after conflict analysis, it is guaranteed not to be evicted from the cache. However, a memory block with a classification of "Always Miss" or "Non-Classified" can potentially incur additional cache misses at instruction level due to conflicting memory accesses from the other core. For each such memory block m mapped to cache set C, we check if $\mathcal{M}(C) = \emptyset$. If not, then we modify the classification of all but the first instruction in m to "Non-Classified". The first instruction retains the original classification of "Always Miss" or "Non-Classified".

Optimization for Set-Associativity In the discussion so far, we blindly converted each "Always Hit" reference to "Non-Classified" if there are potential memory accesses to the same cache set from the other interfering tasks. However, for set-associative caches, we can perform more accurate conflict analysis. Again, let m be a memory reference of task T at program point p that has been classified as "Always Hit" in the L2 cache and it maps to cache set C. Clearly, m is present in the abstract cache state (ACS) at program point p corresponding to *must analysis*. Let $age(m)$ be the age of reference m in the ACS of must analysis. The definition of ACS implies that m should stay in the cache for at least $N - age(m)$ unique memory block references where N is the associativity of the cache [24]. Thus, if $|\mathcal{M}(C)| \leq N - age(m)$, memory block m cannot be evicted from the L2 cache by interfering tasks. In this case, we should keep the classification of m as "Always Hit".

5.3.3 WCRT Analysis

In this step, we take the results of the cache analysis at all levels to determine the BCET and WCET of all tasks. Table 2 presents how we deduce the latency of a reference r in the best and worst case given its classification at L1 and L2. Here, hit_L denotes the latency of a hit at cache level L, which consists of (1) the total delay for cache tag comparison at all levels $l : 1 \ldots L$, and (2) the latency to bring the content from level L cache to the processing core. $miss_{L2}$, the L2 miss latency, consists of (1) the total delay for cache tag comparison at L1 and L2 caches, and (2) the latency to access the reference from the main memory and bring it to the processing core.

Table 2: Access latency of a reference in best case and worst case given its classifications

L1 cache	**L2 cache**	**Access latency**	
		Best-case	**Worst-case**
AH	–	hit_{L1}	hit_{L1}
AM	AH	hit_{L2}	hit_{L2}
AM	AM	$miss_{L2}$	$miss_{L2}$
AM	NC	hit_{L2}	$miss_{L2}$
NC	AH	hit_{L1}	hit_{L2}
NC	AM	hit_{L1}	$miss_{L2}$
NC	NC	hit_{L1}	$miss_{L2}$

As a general rule, an AH reference at level L incurs hit_L latency for all cases, and an AM reference at level L incurs $miss_L$ latency for all cases. An NC reference is interpreted as hits in the best case, and as misses in the worst case. We assume an architecture free from timing anomaly so that we can assign miss latency to an NC reference in the worst case. Having determined the latency of each reference, we can compute the best-case and worst-case latency of each basic block by summing up all incurred latencies. A shortest (longest) path search is then applied to obtain the BCET (WCET) of the whole task.

In order to compute the WCRT of MSG, we need to know the time interval of each task. The task ordering within a node of the MSG model (denoting an MSC) is given by the partial order of the corresponding MSC. The task ordering across nodes of the MSG model are captured by the directed edges in the MSG. Given a task t, we use four variables $EarliestReady[t]$, $LatestReady[t]$, $EarliestFinish[t]$, and $LatestFinish[t]$ to represent its execution time information. Given a task t, its execution interval is from $EarliestReady[t]$ to $LatestFinish[t]$. These notations are explained below:

- $EarliestReady[t]/LatestReady[t]$: earliest/latest time when all of t's predecessors have completed execution.
- $EarliestFinish[t]/LatestFinish[t]$: earliest/latest time when task t finishes its execution.
- $separated(t, u)$: If tasks t and u do not have any dependencies and their

Algorithm 4: EarlistTime and LatestTime Computation of MSG, G

```
1  $step = 0$ ;
2  Initliaze $separated[.,.]$ to 0;
3  foreach node $i \in G$ do
4      $EarliestReady[i] = 0; LatestReady[i] = 0;$;
5  $EarliestTimes(G)$;
6  repeat
7      $LatestTimes(G)$;
8      $Separated\_Computation()$ ;
9      $step = step + 1$;
10 until $separated[.,.]$ is unchanged or step > MAX_STEP
   ;
11 function (EarliestTimes(MSG G))
12 foreach node $i \in G$ in topologically sorted order do
13     $EarliestFinish[i] = EarlistReady[i] + BCET[i]$;
14     foreach immediate successor $k$ of $i$ do
15         $EarlistReady[k] = max(EarliestReady[k], EarlistFinish[i])$;
16 function (LatestTimes(MSG G))
17 foreach node $i \in G$ in topologically sorted order do
18     $LatestStart[i] = LatestReady[i]$;
19     $S_{peer} = \{j | \neg separated[i,j] \wedge\ i,j$ are on the same core$\}$;
20     foreach $j \in S_{peer}$ do
21         $LatestStart[i] = LatestStart[i] + WCET[j]$; ;
22     $LatestFinish[i] = LatestStart[i] + WCET[i]$;
23     foreach immediate successor $k$ of $i$ do
24         $latestReady[k] = max(latestReady[k], latestFinish[i])$;
```

Figure 12: EarlistTime and LatestTime Computation

execution interval do not overlap or if asks t and u have dependencies , then $separated(t,u)$ is assigned true; otherwise it is assigned false.

In a non-preemptive system, $EarliestFinish[t] = EarliestReady[t] + BCET[t]$. Also, task t is ready only after all its predecessors have completed execution, that is, $EarliestReady[t] = max_{u \in P}(EarliestFinish[u])$, where P is the set of predecessors of task t. For a task t without any predecessor $EarliestReady[t] = 0$.

However, latest finish time of a task is not only affected by its predecessors but also its peers (non-separated tasks on the same core). For task t, we define

$$S^t_{peers} = \{t' | \neg separated[t', t] \wedge\ t', t \text{ are on the same core}\}$$

In other words, S^t_{peers} is the set of tasks whose execution interfere with task t

on the same core. Let P be the set of predecessors of task t. Then we have

$$\begin{aligned} LatestReady[t] &= max_{u \in P}(LatestFinish[u]) \\ LatestFinish[t] &= LatestReady[t] + WCET[t] + \sum_{t' \in S^t_{peers}} WCET[t'] \end{aligned}$$

However, the change of latest times of tasks may lead to different interference scenario (i.e., $separated[.,.]$ may change), which might change the latest finish times. Thus, latest finish times are estimated iteratively until the $separated[.,.]$ do not change. $separated[t,u]$ is initialized to 0 if tasks t and u do not have any dependency and 1 otherwise. When iterative process terminates, we are able to derive the final application WCRT as

$$WCRT = max_t\ LatestFinish(t) - min_{t'}\ EarliestReady(t')$$

that is, the duration from the earliest start time of any task until the latest completion time of any task. The pseudo-code of the $EarlistTime$ and $LatestTime$ computation is shown in 12. Note that this iterative process within WCRT analysis is different from the iterative process shown in Figure 7.

A by-product of WCRT analysis is the set of tasks that can potentially conflict in L2 cache, that is, tasks whose execution intervals (from $EarliestReady$ to $LatestFinish$) overlap. This information, if different from the previous iteration, will be fed back to the cache conflict analysis to refine the classification for L2 accesses.

5.4 Termination Guarantee

Now we proceed to prove that the iterative L2 cache conflict analysis framework shown in Figure 7 terminates.

Theorem 5.1. *For any task t, the level 2 cache conflict analysis does not change its BCET.*

Proof. Our level 2 cache conflict analysis only considers the memory blocks classified as “Always Hit” for L2 cache as shown in figure 12. Some of these memory blocks might be changed to “Non-Classified” due to interference from conflicting tasks while others remain as “Always Hit”. There are two possibilities according to Table 2. One possibility is memory blocks are classified as L1 “Always Miss”, but for both two cases (L2 AH and NC), the memory blocks are considered as L2 cache hit for the best case. The other possibility is memory blocks are classified as L1 “Non-Classified”, but for both two cases (L2 AH and NC), the memory blocks are considered as L1 cache hit for the best case. Hence, our L2 cache analysis does not change the task’s BCET. □

Theorem 5.2. *For a task t, its EarliestReady[t] does not change across iterative L2 cache and WCRT analysis.*

Proof. We prove Theorem 5.2 by contradiction. Assume for a task t, its $earlistReady[t]$ changes. Thus, this must be due to the change of its predecessors’s $earliestReady[t]$, because a task’s BCET remains unchanged according to Theorem 5.1. In the end, $earliestReady[src]$ must change (src are the tasks without any predecessors), contradicting with fact that $earliestReady[src] = 0$ always. □

Now we can infer tasks that can potentially conflict in L2 cache, that is, tasks whose execution intervals (from $EarliestReady$ to $LatestFinish$) overlap. This information, if different from the previous iteration, will be fed back to the cache conflict analysis to refine the classification for L2 accesses to compute the updated WCET and time interval for each task. Our iterative analysis is guaranteed to terminate, because the task interferences are shown to monotonically decrease.

Theorem 5.3. *Task interferences monotonically decrease (strictly decrease or remain the same) across different iterations of our analysis framework.*

Proof. We prove by induction on number of iterations.
Base Case: In the first iteration, tasks are assumed to conflict with all the tasks on other cores (except those excluded by partial order). This is the worst case task interference scenario. Thus, the task interferences of the second iteration

definitely monotonically decrease compared to the first iteration.

Induction Step: We need to show that the task interferences monotonically decrease from iteration n to iteration $n+1$ assuming that the task interferences monotonically decrease from iteration $n-1$ to n. We prove by contradiction. Assume two tasks i and j do not interfere at iteration n, but interfere at iteration $n+1$. There are two cases.

- $EarliestReady[j] \geq LatestFinish[i]$ at iteration n, but $EarliestReady[j] < LatestFinish[i]$ at iteration $n+1$. This implies that $LatestFinish[i]$ at iteration $n+1$ increases because $EarliestReady[j]$ remains unchanged across iterations according to Theorem 5.1. $LatesteFinish[i]$ at iteration $n+1$ can increase due to three reasons: (a) at iteration $n+1$, the WCET of task i itself increases; (b) the WCET of some tasks which task i depends on directly or indirectly increases; and (c) the WCET of some tasks increases as a result of which either the number of peers of task i ($|S^i_{peers}|$) increases or the WCET of a peer of task i increases. In summary, at least one task's WCET is increased. The WCET increase at iteration $n+1$ of some task implies that more memory blocks are changed from "Always Hit" to "Non-Classified" due to the task interference increase at iteration n. However, this contradicts with the assumption that task interference monotonically decrease at iteration n.

- $EarliestReady[i] \geq LatestFinish[j]$ at iteration n, but $EarliestReady[i] < LatestFinish[j]$ at iteration $n+1$. The proof is symmetric to the first case.

$\square$

As task interferences decrease monotonically across iterations, the analysis must terminate.

6 Experiments

6.1 Setup

We evaluate our analysis technique on a real-world application adapted from DEBIE-I DPU Software [7], shown in Figure 5 and Papabench [17], which is a Unmanned Aerial Vehicle (UAV) control application. For the DEBIE benchmark, there are total 35 tasks. The code size and the mapping of tasks to the cores in a 4-core system are shown in Table 3. As shown, the code size of tasks vary from 320 bytes to 23,288 bytes. For the 2-core setting, the tasks assigned to Core 3 and Core 4 are merged into Core 1. For Papabench, there are 28 tasks. The code size of tasks vary from 96 bytes to 6,496 bytes. The detailed task size and mapping of papabench are shown in Table 4. In Figure 13, we show the average number of tasks mapped to a set for both DEBIE and Papabench. As shown, there are quite number of conflicts at task level. With our accurate task lifetime analysis, many tasks mapped to the set do not conflict due to disjoint lifetime.

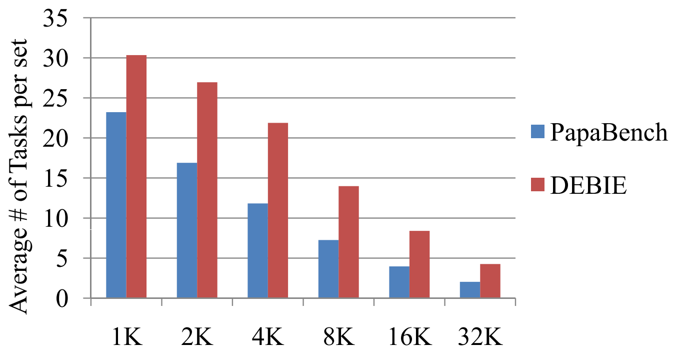


Figure 13: Average number of task per set for different size of cache.

Our analysis is based on SimpleScalar [4]. As we are modeling the cache, we assume a simple in-order processor with unit-latency for all data memory references. We perform all experiments on a 3GHz Pentium 4 CPU with 2GB memory. The individual tasks are compiled into SimpleScalar-compliant binaries, and their control flow graphs (CFGs) are extracted as input to the cache analysis framework. Our analysis produces the WCRT result when the iterative work flow as shown in Figure 7 terminate. The estimate produced after the first iteration assumes that any pair of tasks assigned to different cores may execute concurrently and evict each other's content from the shared cache. This value is essentially the estimation result following Yan-Zhang's technique [26] — the only available shared-cache analysis method in the literature (see Section 3). The improvement in WCRT estimation accuracy due to our proposed analysis is demonstrated by comparing this value to the final estimation result of our technique after iterative tightening.

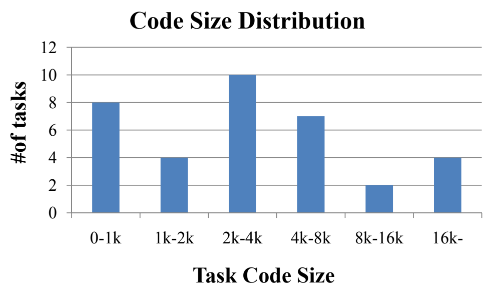


Figure 14: Code size distribution of DEBIE benchmark.

As we are modeling the cache, we assume a simple in-order processor with unit-latency for all data memory references. We perform all experiments on a 3GHz Pentium 4 CPU with 2GB memory.

Our analysis produces the WCRT result when the iterative work flow as shown in Figure 7 terminates. The estimate produced after the first iteration assumes that any pair of tasks assigned to different cores may execute concurrently and evict each other's content from the shared cache. This value is essentially the estimation result following Yan-Zhang's technique [26] — the only available shared-cache analysis method in the literature (see Section 3). The improvement in WCRT estimation accuracy due to our proposed analysis is demonstrated by comparing this value to the final estimation result of our technique. In the following, we evaluate DEBIE benchmark first.

6.2 Comparison with Yan-Zhang's method

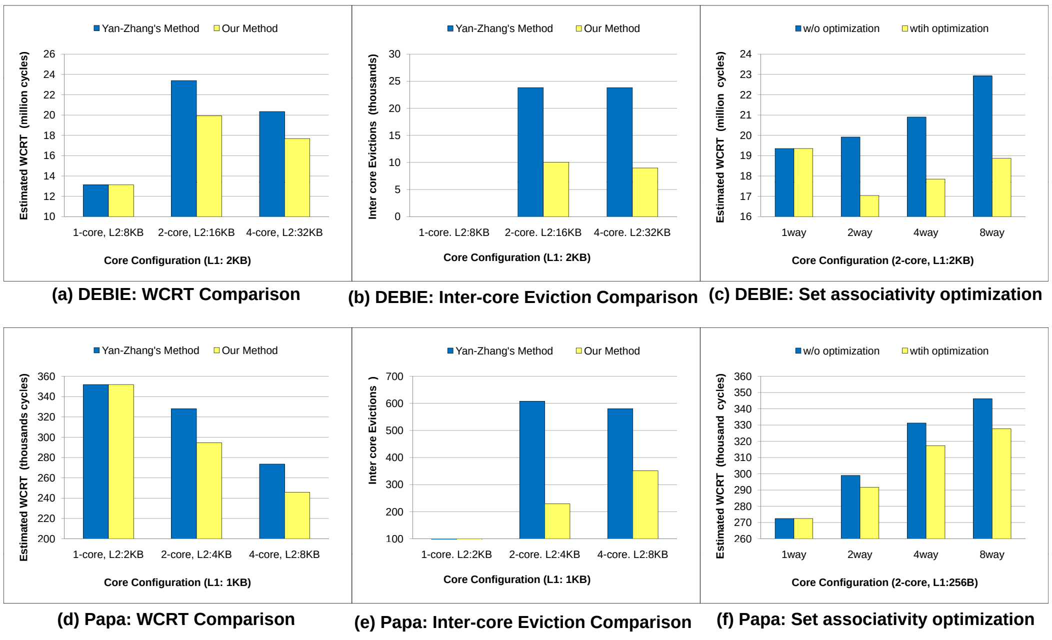
Yan-Zhang's analysis [26] is restricted to direct mapped cache. Thus, to make a fair comparison, we first configure both L1 and L2 as direct mapped caches.

Table 3: Characteristics and settings of the DEBIE benchmark

MSC	**Task**	**Codesize (bytes)**	**Core**
1	*boot_main*	3,200	1
2	pwr_main_1	9,456	1
	pwr_main_2	3,472	1
	pwr_class	1,648	4
3	wr_main_1	3,408	1
	wr_main_2	5,952	1
	wr_class	1,648	4
4	*rcs_main*	3,400	1
5	*rwd_main*	3,400	1
6	$init\_main_1$	320	1
	$init\_main_2$	376	1
	$init\_main_3$	376	1
	$init\_main_4$	376	1
	init_health	5,224	2
	init_telecm	4,408	2
	init_acqui	200	4
	init_hit	616	4
7	sby_health_1	16,992	2
	sby_health_2	448	2
	sby_telecm	23,288	2
	sby_su_1	6,512	4
	sby_su_2	4,392	4
	sby_su_3	1,320	4
8	acq_health_1	16,992	2
	acq_health_2	448	2
	acq_telecm	23,288	2
	acq_acqui_1	3,136	4
	acq_acqui_2	3,024	4
	acq_telemt	3,768	3
	acq_class	3,064	4
	acq_hit	8,016	4
	acq_su_0	2,536	4
	acq_su_1	6,512	4
	acq_su_2	4,392	4
	acq_su_3	1,320	4

Table 4: Characteristics and settings of the Papa benchmark

Core	Task	Codesize (bytes)
1	fm_0	808
	fm_1	96
	fm_2	96
	fm_3	1,696
	fm_4	136
	fm_5	248
1	fv_0	520
	fv_0	656
2	fr_0	384
	fr_1	4,552
2	fs_0	272
	fs_1	992
	fs_2	1,840
3	am_0	768
	am_1	96
	am_2	96
	am_3	1,240
	am_4	1,536
3	ad_0	352
	ad_1	2,296
	ad_2	6,496
4	as_0	560
	as_1	2,744
	as_2	1,720
	as_3	168
	as_4	656
4	ag_0	400
4	ar_0	5,520



(a) DEBIE: WCRT Comparison **(b) DEBIE: Inter-core Eviction Comparison** **(c) DEBIE: Set associativity optimization**

(d) Papa: WCRT Comparison **(e) Papa: Inter-core Eviction Comparison** **(f) Papa: Set associativity optimization**

Figure 15: Comparison between Yan-Zhang's method and our method and the improvement of set associativity optimization.

Figure 15(a) shows the comparison of the estimated WCRT between Yan-Zhang's analysis and ours on varying number of cores. The size of L1 cache is 2KB bytes with 16-byte block size. The L2 cache has 32-byte block size. The L2 cache size is doubled with the doubling of the number of cores. We assume 1 cycle latency for L1 hit, 10 cycle latency for L1 cache misses and 100 cycle latency for L2 cache misses. When only one core is employed, the tasks execute non-preemptively without any interference. Thus the two methods produce the exact same estimated WCRT. In the 2-core and 4-core settings where task interferences become significant to the analysis, our method achieves up to 15% more accuracy over Yan-Zhang's method.

As tasks are distributed on more cores, the parallelization of task execution may reduce overall runtime. But at the same time, the concurrency gives rise to inter-core L2 cache content evictions that contribute to an increase in task runtime. In this particular experiments, we observe that the WCRT value can increase (1-core to 2-core) as well as decrease (2-core to 4-core) with increasing number of cores.

In Figure 15(b), we compare the number of inter-core evictions estimated by both methods for the same configurations as in Figure 15(a). When only one core is employed, there is no inter-core evictions for both methods. For multi-core systems, due to the accurate task interference, the number of inter-core evictions of our method are much smaller than Yan-Zhang's method as shown in Figure 15(b). This explains the WCRT improvement in Figure 15(a).

6.3 Set associative caches

Our method is able to handle set-associative caches accurately by taking into account the age of the memory blocks. Figure 15(c) compares the estimated WCRT with and without the optimization for set-associativity (see Section 5.3.2) in a 2-core system. Without the optimization, all the "Always Hit" accesses are turned into "Non-Classified" accesses as long as there are conflicts from other cores, regardless of the memory blocks' age. Here, L1 cache is configured as 2KB direct mapped cache with 16-byte block size and L2 cache is configured as a 32KB set-associative cache with 32-byte block size, but varied associativity (1, 2, 4, 8). As shown in Figure 15(c), when associativity is set to 1 (direct mapped cache), there is no gain from the optimization. However, for associativity ≥ 2, the estimated WCRT is improved significantly with the optimization.

6.4 Sensitivity to L1 cache size

Figure 16(a) shows the comparison of the estimated WCRT on a 2-core system where L1 cache size is varied but L2 cache size is kept as constant. Again both L1 and L2 caches are configured as direct mapped caches due to the limitation of Yan-Zhang's analysis. Our method is able to filter out evictions among tasks with separated lifetimes and achieves up to 20% more accuracy over Yan-Zhang's method.

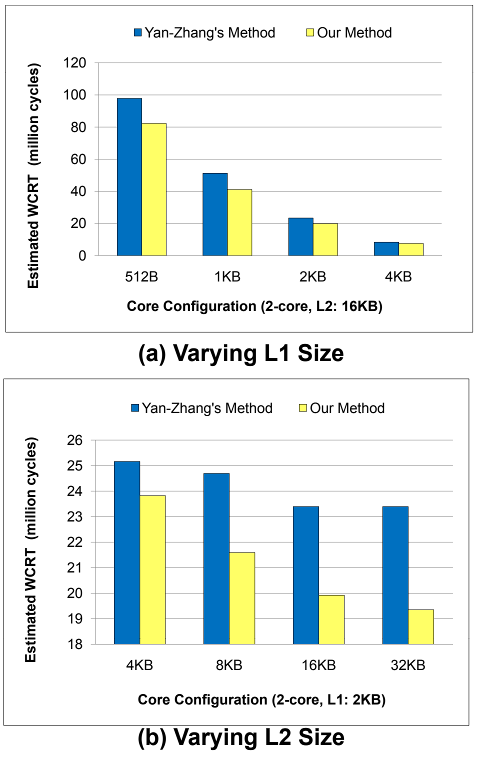


Figure 16: Comparison of estimated WCRT between Yan-Zhang's method and our method for varying L1 and L2 cache sizes.

6.5 Sensitivity to L2 cache size

Figure 16(b) shows the comparison of the estimated WCRT on a 2-core system where L2 cache size is varied but L1 cache size is kept as constant. Here too, both L1 and L2 cache are configured as direct mapped caches. We observe slightly larger improvement as we increase the L2 cache size. In general, more space in L2 cache reduces inter-task conflicts. Without refined task interference information, however, there can be significant pessimism in estimating inter-core evictions, which limits the benefit of the larger space in the perspective of Yan-Zhang's analysis. As a result our analysis is able to achieve lower WCRT estimates as compared to Yan-Zhang's method.

6.6 PapaBench

For Papabench, we evaluate our analysis in terms of the aforementioned three perspectives. In Figure 15(d) and (e), we compare the WCRT estimation and

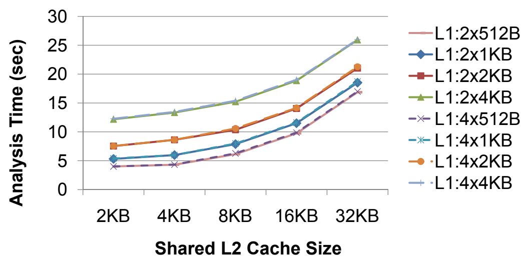


Figure 17: Runtime of our iterative analysis .

inter-core eviction. The L1 cache is 1KB bytes with 16-byte block size. L2 cache double its size with increase number of cores, starting with 2KB for 1-core system. For papabench, we achieves about 10% more accuracy over Yan-Zhang's method in terms of WCRT estimation. For set associativity optimization, L1 cache is 256B and L2 cache is configured as 8KB set-associative cache with 32-byte block size, but varied associativity (1, 2, 4, 8). The optimization gain is shown in Figure 15(f).

6.7 Scalability

Finally, Figure 17 sketches the runtime of our complete iterative analysis (L2 cache and WCRT analysis) for various configurations. It takes less than 30 seconds to complete our analysis for any considered settings.

7 Future Work

In future, we are planning to extend the work in several directions. This will also amount to relaxing or removing the restrictions in our current analysis framework.

Currently, we only handle the instruction memory hierarchy in this work. We assume that the data memory references do not interfere in any way with the L1 and L2 instruction caches modeled by us or data cache are separated from instruction cache and we do not model that.

Instead of LRU cache replacement policy, we can extend our work to handle other practical cache replacement policies such as pseudo-LRU, FIFO and so on.

We also assume that there is no code sharing between tasks. However, that is not the actual case since library calls are common in programs. We can model code sharing directly to capture the constructive effect of shared code across tasks.

8 Conclusion

In this thesis, we develop a worst-case response time (WCRT) analysis of concurrent programs, where the concurrent execution of the tasks is analyzed to bound the shared cache interferences.

We have presented a worst-case response time (WCRT) analysis of concurrent programs running on shared cache multi-cores. Our concurrent programs are captured as graphs of Message Sequence Charts (MSCs) where the MSCs capture ordering of computation tasks across processes. Our timing analysis iteratively identifies tasks whose lifetimes are disjoint and uses this information to rule out cache conflicts between certain task pairs in the shared cache. Our analysis obtains lower WCRT estimates than existing shared cache analysis methods on a real-world application.

References

[1] Message Sequence Charts. ITU-TS Recommendation Z.120, 1996.

[2] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. *Lecture Notes in Computer Science*, 1145:52–66, 1996.

[3] R. Alur and M. Yannakakis. Model checking message sequence charts. In *CONCUR*, 1999.

[4] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[5] J. Brown. Application-customized CPU design: The Microsoft Xbox 360 CPU story. Available on: `http://www-128.ibm.com/developerworks/power/library/pa-fpfxbox/?ca=dgr-lnxw07XBoxDesign`, 2005.

[6] L.M.N. Coutinho, J.L.D. Mendes, and C.A.P.S. Martins. MSCSim – Multilevel and Split Cache Simulator. In *36th Annual Frontiers in Education Conference*, 2006.

[7] European Space Agency. DEBIE – First standard space debris monitoring instrument, 2008. Available at: http://gate.etamax.de/edid/publicaccess/debie1.php.

[8] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.

[9] R. Heckmann et al. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 9(7), 2003.

[10] C.-G. Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[11] J. W. Lee and K. Asanovic. METERG: Measurement-based end-to-end performance estimation technique in QoS-capable multiprocessors. In *RTAS*, 2006.

[12] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *RTSS*, 1996.

[13] T. Lundqvist and P. Stenstrom. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3), 1999.

[14] F. Mueller. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1997.

[15] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2-3), 2000.

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

[17] F. Nemer and et al. Papabench: A free real-time benchmark. In *WCET Workshop*, 2006.

[18] P. Puschner and M. Schoeberl. On composable system timing, task timing, and WCET analysis. In *WCET Workshop*, 2008.

[19] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *CODES+ISSS*, 2008.

[20] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 System Microarchitecture. Available on: `http://researchweb.watson.ibm.com/journal/rd/494/sinharoy.html`, 2005.

[21] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT*, 2004.

[22] Sun Microsystems, Inc. UltraSPARC T1 Overview. Available on: `http://www.sun.com/processors/UltraSPARC-T1/index.xml`, 2006.

[23] Y. Tan and V. Mooney. WCRT analysis for a uniprocessor with a unified prioritized cache. In *LCTES*, 2005.

[24] 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.

[25] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES*, 2000.

[26] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *RTAS*, 2008.