Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores

Yan Li Vivy Suhendra Yun Liang Tulika Mitra Abhik Roychoudhury
School of Computing
National University of Singapore
{*yanli,vivy,liangyun,tulika,abhik*}*@comp.nus.edu.sg*

***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 paper, we develop a timing analysis method for concurrent software running on multi-cores with a shared instruction cache. Communication across tasks is by message passing where the message mailboxes are accessed via interrupt service routines. We do not handle data cache, shared memory synchronization and code sharing across tasks. Our method progressively improves 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 lower worst-case response time (WCRT) estimates than existing shared-cache analysis on a real-world embedded application.**

I. Introduction

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.

Our *system model* consists of a concurrent program visualized as a graph, each node of which is a Message Sequence Chart or MSC [1]. MSC 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 1 shows a simple MSC with five processes (*Main*, *Health Monitoring* etc.) executing the tasks $main_1, \ldots, main_4$, hm etc. Note that an MSC denotes a labeled partial order of tasks.

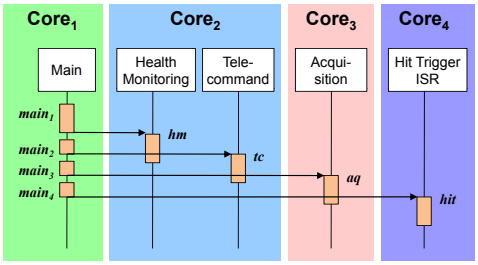

Figure 1. A simple MSC and a mapping of its processes to cores.

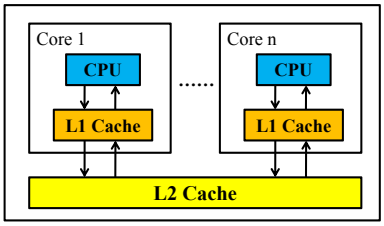

Figure 2. A multi-core architecture with shared cache.

Our *system architecture* consists of a multi-core where the individual processes in the program (the vertical lines of the MSCs) are mapped to the different cores (see Figure 1). With such a mapping, an MSC provides a natural specification of interactions among the processes in a concurrent program running on multi-cores. As multi-cores are increasingly adopted in high-performance embedded systems, the on-chip cache hierarchy becomes more complex. We consider an architecture where each processor core has private first-level (L1) cache. However, a second-level (L2) cache is shared across the processor cores (see Figure 2).

Certainly, the analysis effort required for capturing the timing effects in the presence of a shared cache is complex, as memory contention across the multiple cores significantly affects the shared cache behavior. In particular, accesses to the L2 cache originating from different cores may conflict with each other. Thus, isolated cache analysis of each task that does not account for these conflicts will not safely bound the execution time of the task.

Contributions: In this paper, we develop a worst-case response time (WCRT) analysis of concurrent programs, where the concurrent execution of the tasks are analyzed to bound the shared cache interferences. Our method advances the state-of-the-art in shared cache multi-core timing analysis [23] in several ways. First of all, our iterative analysis estimates which tasks (running on two different cores) can have overlapping lifetimes. If two tasks cannot overlap, they cannot affect each other in terms of conflict misses and thus we can reduce the number of estimated conflict misses in the shared cache. This leads to improved timing estimates. Moreover, we consider set-associative caches in our analysis as opposed to only direct mapped caches and this creates additional opportunities for improving the timing estimation. In summary, we develop a timing analysis method for shared cache multi-cores that enhances the state-of-the-art.

Assumptions: Our analysis framework has the following assumptions.

- *Data Cache:* We only handle the instruction memory hierarchy in this work. 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.
- *Cache Architecture:* We consider Least Recently Used (LRU) cache replacement policy for set-associative caches. The L2 cache block size is assumed to be larger than or equal to the L1 cache block size. Finally, we are analyzing non-inclusive multi-level caches [7].
- *Other architectural features:* We only consider architectures without timing anomalies caused by interactions between caches and other architecture features.
- *Shared code across tasks:* We assume that the tasks do not share any common code. In case two tasks share a function f — we create two separate copies of function f, one for each task. This is because we do not handle constructive effects of shared cache in this work.
- *Inter-task communication:* In our framework, the tasks communicate with each other through message passing via mailboxes. The tasks deposit or receive messages from the mailbox through interrupt service routines (ISR). Exclusive access to the mailbox is ensured by disabling interrupts within ISR. A task waiting on a message is notified by the ISR once the message is available in the mailbox. Finally, we assume that there is no overflow in any mailbox, that is, mailboxes are of unbounded length.

II. System Model and Architecture

In this section, we give some background on 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.

A. Message Sequence Charts

A Message Sequence Chart (MSC) [1] is a variant of an UML sequence diagram with a formal semantics. Figure 1 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 1). Each process executes a sequence of "tasks" shown via shaded rectangles (e.g., $main_1$, hm, tc are tasks in Figure 1). 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 1, 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 1), 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.

B. 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

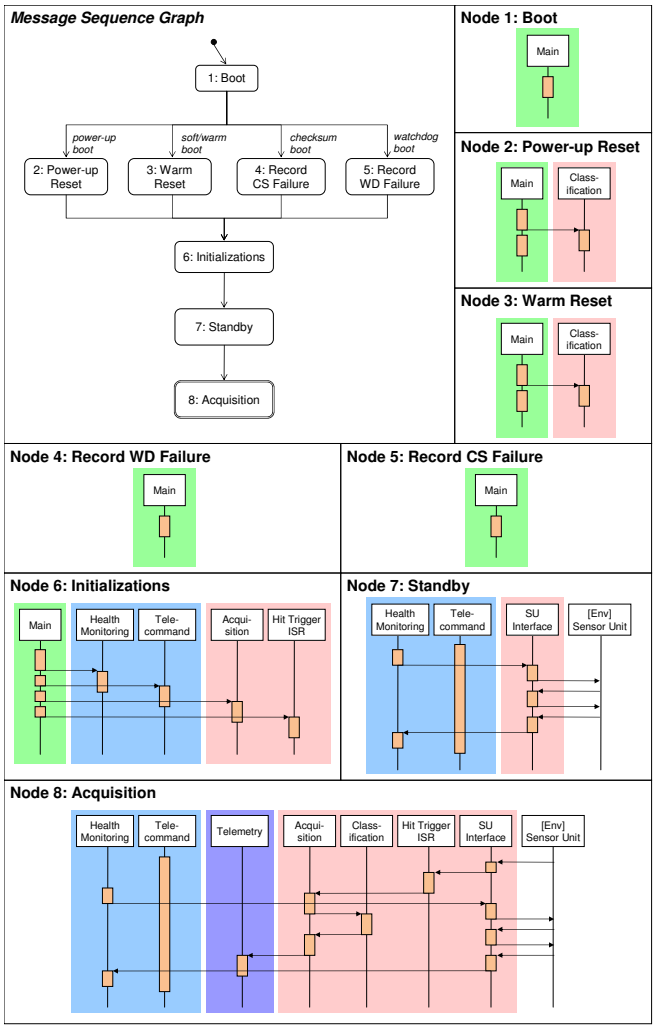

Figure 3. DEBIE Case Study. Different colors are used to show the mapping of the processes to different processor cores.

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].

C. DEBIE Case Study

Our case study consists of DEBIE-I DPU Software [6], 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 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 3. 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.

D. System Architecture

The generic multi-core architecture we target here is quite representative of the current generation multi-core systems as shown in Figure 2. 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).

Each cache can be either direct-mapped or set-associative. In this paper, we consider Least Recently Used (LRU) cache replacement policy for set-associative caches. Also, we consider architectures without timing anomalies caused by interactions between caches and other architecture features. The L2 cache block size is assumed to be larger than or equal to the L1 cache block size. Finally, we are analyzing non-inclusive multi-level caches [7]. Even though we consider two levels of caches here, our approach can be easily extended to handle more levels of cache hierarchy using the same propagation principle from L1 cache to L2 cache presented in this paper.

III. Analysis Framework

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 4 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 ([9], [22], [15], [18]) 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 [7] for intra-core analysis.

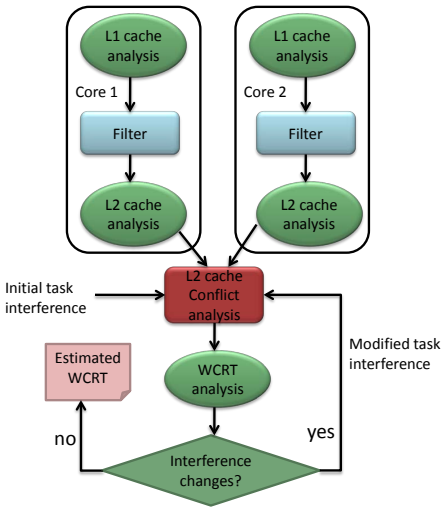

Figure 4. Our Analysis Framework

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 [23].

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 III-A 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 4 that allows us to improve the lifetime bounds with each iteration of the analysis.

A. Illustration

We illustrate our iterative analysis framework on the MSC depicted in Figure 1. 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

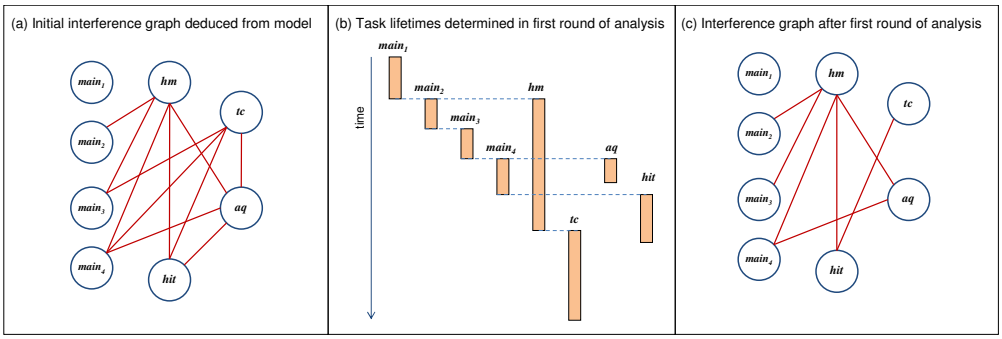

Figure 5. The working of our shared-cache analysis technique on the example given in Figure 1

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 relations among tasks in an *interference graph* as shown in Figure 5(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 4), 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 5(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 tells 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 5(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.

IV. Analysis Components

The first step of our analysis framework is the independent cache analysis for each core (see Figure 4). As mentioned before, we use the multi-level non-inclusive cache analysis proposed by Hardy and Puaut [7] 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.

A. Intra-Core Cache Analysis

The intra-core cache analysis step employs abstract interpretation method [21] at both L1 and L2 cache levels. The additional step for multi-level caches is the filter function (see Figure 4) 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 [21]. In this paper, we consider LRU replacement policy, but the cache analysis can be extended for other replacement polices as shown in [8].

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

The analysis results can be used to classify the memory blocks in the following manner.

- **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.
- **Persistent (PS):** If a memory block is guaranteed never to be evicted from the cache, 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.

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 [7]. The filter function is shown below.

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

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. [7] 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.

B. L2 Cache Conflict Analysis

Shared L2 cache conflict analysis is the central component of our framework. It takes in two inputs, namely the task interference graph (see Figure 5) generated by the WCRT analysis step and the abstract cache states plus the classification corresponding to L2 cache analysis for each task in each core. 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 that 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 references 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 block 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".

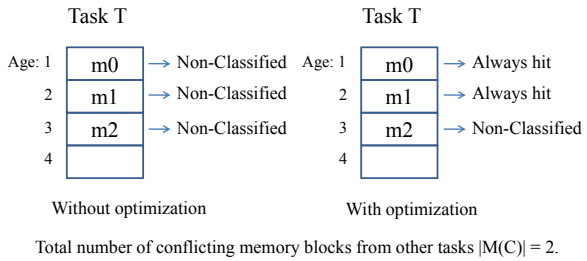

Figure 6. An example of 4-way set associative L2 cache. The abstract cache state of task T for cache set C at a program point during *must analysis* is shown. Memory blocks are converted to either "Always Hit" or "Non-Classified" according to their ages and the number of conflicting memory blocks from interfering tasks.

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 to cache set C where N is the associativity of the cache [21]. 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". Figure 6 shows an example. Memory blocks $m0$ and $m1$ are kept as "Always Hit" because the number of conflicting memory blocks from interfering tasks ($\mathcal{M}(C) = 2$) are not enough to evict them. However, memory block $m2$ is converted to "Non-Classified" due to its old age.

C. 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 I 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 $l : 1 \dots L$, 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 I
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}	$miss_{L2}$

Note that 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 [19].

In order to compute the WCRT of MSG, we need to know the time interval of each task. The task ordering within a node (denoting an MSC) of the MSG model 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 execution interval do not overlap or if tasks 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

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

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 false if tasks t and u do not have any dependency and true otherwise. When iterative process terminates, we are able to derive the final application WCRT as

$$\begin{aligned} WCRT &= max_t\ LatestFinish(t) \\ &\quad - min_{t'}\ EarliestReady(t') \end{aligned}$$

that is, the duration from the earliest start time of any task until the latest completion time of any task. Note that this iterative process within WCRT analysis is different from the iterative process shown in Figure 4.

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.

D. Termination Guarantee

Now we proceed to prove that the iterative L2 cache conflict analysis framework shown in Figure 4 terminates.

Theorem IV.1. *For any task t, its BCET and EarliestReady[t] do not change across different iterations of L2 cache conflict and WCRT analysis.*

Proof: Our level 2 cache conflict analysis only considers the memory blocks classified as "Always Hit" for L2 cache. Some of these memory blocks might be changed to "Non-Classified" due to interference from conflicting tasks while others remain as "Always Hit". An "Always Hit" memory block in L2 cache should have "Always Miss" or "Non-Classified" status in L1 cache. According to Table I, a memory block classified as L1 "Always Miss" is considered as L2 cache hit in the best case irrespective of whether is it AH or NC in L2 cache. Similarly, a "Non-classified"

memory block in L1 is considered as L1 cache hit in the best case irrespective of its classification in the L2 cache. Hence, L2 cache conflict analysis cannot reduce the best case access time of a memory reference and hence a task's BCET does not change across different iterations of our analysis.

We prove that $EarlistReady[t]$ does not change through contradiction. Let us assume that for a task t, its $EarlistReady[t]$ changes. This must be due to a change in its predecessors's $EarliestReady[t]$ because a task's BCET remains unchanged. Proceeding backwards, $EarliestReady[src]$ must have changed where src is a task without any predecessor, contradicting the fact that $EarliestReady[src] = 0$. Hence, for a task t its $EarliestReady[t]$ does not change. ■

Theorem IV.2. *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 IV.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 interferences 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.

■

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

V. Related Work

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], [21]) which also forms the foundation to the framework presented in this paper. Mueller [14] extends the technique for multi-level cache analysis; Hardy and Puaut [7] 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 [14], 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 ([9], [22], [15]). 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 [18]. A different perspective in [20] 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 [17], [5] and [10]. 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 [16].

The only technique in literature that has addressed inter-core shared-cache analysis so far is the one proposed by Yan and Zhang [23]. 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 [7] (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

Table II
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

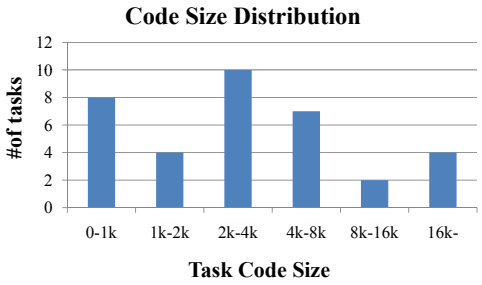

Figure 7. Code size distribution of DEBIE benchmark.

to obtain improved estimates by exploiting the knowledge about interaction among tasks in the multiprocessor.

VI. ESTIMATION RESULTS

Setup: We compile our benchmark for SimpleScalar PISA (Portable ISA) instruction set [4] — a MIPS like instruction set architecture. The individual tasks are compiled into SimpleScalar PISA compliant binaries, and their control flow graphs (CFGs) are extracted as input to the cache analysis framework. The cache analysis framework is built on top of the open-source WCET analysis tool Chronos [11]. Details of the tasks in the DEBIE benchmark and their code-sizes appear in Figure 7 and Table II. The table also shows the mapping of the tasks to the processor cores in a system with four cores.

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 4 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 [23] — the only available shared-cache analysis method in the literature (see Section V). 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.

Comparison with Yan-Zhang's method: Yan-Zhang's analysis [23] is restricted to direct mapped cache. Thus, to make a fair comparison, we first configure both L1 and L2 as direct mapped caches. Figure 8(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 experiment, 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 8(b), we compare the number of inter-core cache evictions estimated by both methods for the same configurations as in Figure 8(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 8(b). This explains the WCRT improvement in Figure 8(a).

Set associative caches: Our method is able to handle set-associative caches accurately by taking into account the age of the memory blocks. Figure 8(c) compares the estimated WCRT with and without the optimization for set-associativity (see Section IV-B) in a 2-core system. Without

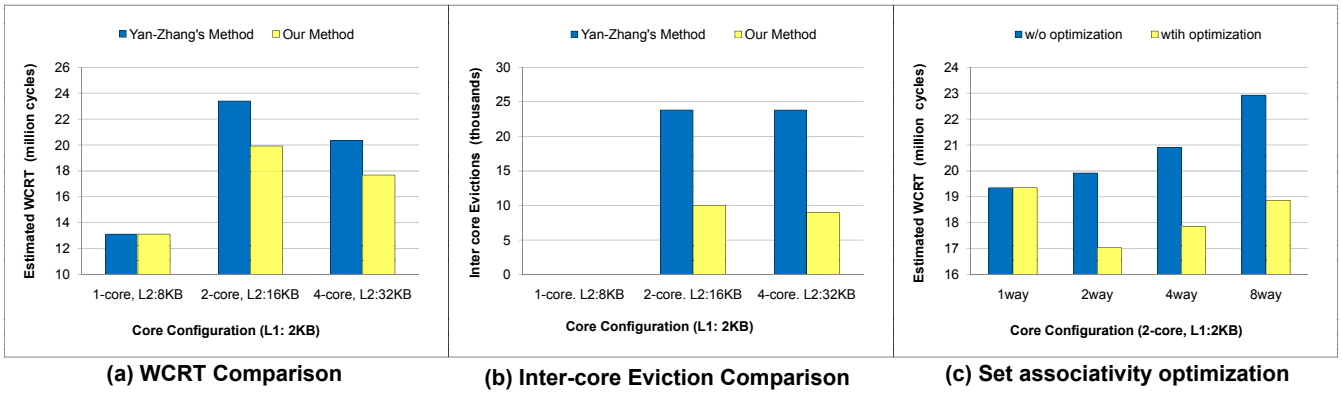

Figure 8. Comparison between Yan-Zhang's method and our method and the improvement of set associativity optimization.

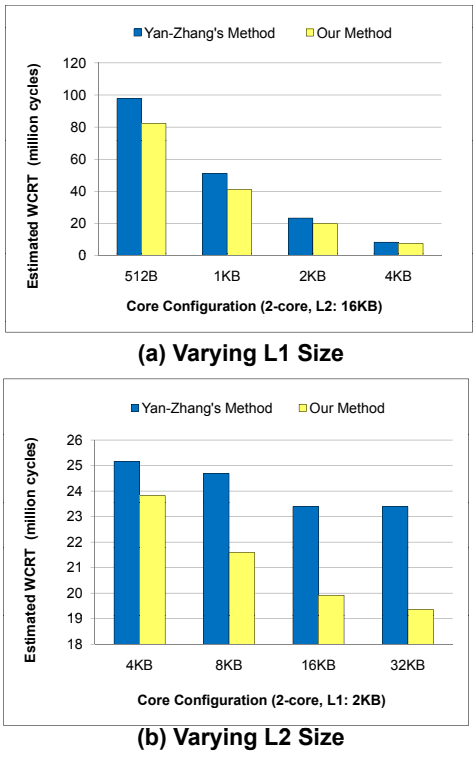

Figure 9. Comparison of estimated WCRT between Yan-Zhang's method and our method for varying L1 and L2 cache sizes.

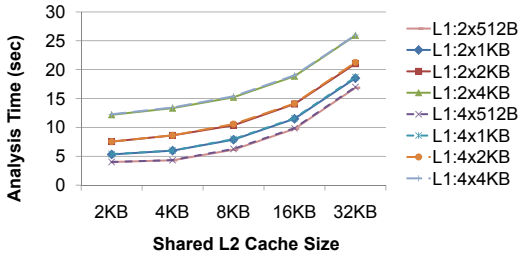

Figure 10. Runtime of our iterative analysis.

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 8(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.

Sensitivity to L1 cache size: Figure 9(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.

Sensitivity to L2 cache size: Figure 9(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 caches 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.

Scalability: Finally, Figure 10 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.

VII. Concluding Remarks

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.

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, namely - (i) handling of data caches, (ii) handling cache replacement policies other than LRU, (iii) directly capturing the constructive effect of shared code (such as libraries) across tasks, and (iv) allowing tasks to communicate via message passing as well as shared memory.

Acknowledgments

This work was partially supported by NUS University Research Council grant R-252-000-321-112 and NUS Faculty Research Council grant R-252-000-387-112.

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 *Proceedings of the International Conference on Concurrency Theory*, 1999.

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

[5] 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.

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

[7] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the Real-Time Systems Symposium*, 2008.

[8] 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.

[9] 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.

[10] J. W. Lee and K. Asanovic. METERG: Measurement-based end-to-end performance estimation technique in QoS-capable multiprocessors. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[11] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007. Available at: http://www.comp.nus.edu.sg/∼rpembed/chronos/.

[12] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proceedings of the Real-Time Systems Symposium*, 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 analysis for instruction caches. *Real-Time Systems*, 18(2-3), 2000.

[15] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of the IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, 2003.

[16] P. Puschner and M. Schoeberl. On composable system timing, task timing, and WCET analysis. In *International Workshop on Worst-Case Execution Time Analysis*, 2008.

[17] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, 2008.

[18] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *Proceedings of the 4th ACM international conference on Embedded software*, 2004.

[19] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the Design Automation Conference*, 2006.

[20] Y. Tan and V. Mooney. WCRT analysis for a uniprocessor with a unified prioritized cache. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, 2005.

[21] 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.

[22] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of the eighth international workshop on Hardware/software codesign*, 2000.

[23] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2008.