Noname manuscript No.
(will be inserted by the editor)

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

Yun Liang · Huping Ding · Tulika Mitra · Abhik Roychoudhury · Yan Li · Vivy Suhendra

the date of receipt and acceptance should be inserted later

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. 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.

Yun Liang
Advanced Digital Sciences Center, Singapore
E-mail: eric.liang@adsc.com.sg

Huping Ding
School of Computing, National University of Singapore
E-mail: dinghuping@comp.nus.edu.sg

Tulika Mitra
School of Computing, National University of Singapore
E-mail: tulika@comp.nus.edu.sg

Abhik Roychoudhury
School of Computing, National University of Singapore
E-mail: abhik@comp.nus.edu.sg

Yan Li
Washington University in St. Louis, USA
E-mail: liy@seas.wustl.edu

Vivy Suhendra
Institute for Infocomm Research, Singapore
E-mail: vsuhendra@i2r.a-star.edu.sg

We show that our method produces lower worst-case response time (WCRT) estimates than existing shared-cache analysis on a real-world embedded application. Furthermore, we also exploit instruction cache locking to improve WCRT. By locking some beneficial memory blocks into L1 cache, the WCET of the tasks and L2 cache conflicts are reduced, resulting in better WCRT. Experiments demonstrate that significant WCRT reduction is achieved through cache locking.

1 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.

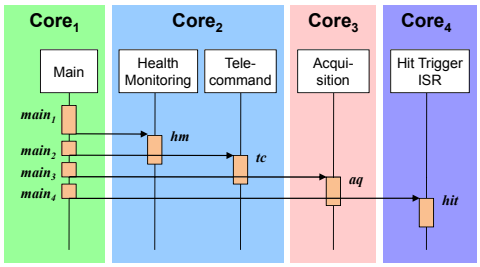


Fig. 1 A simple MSC and a mapping of its processes to cores.

Our *system model* consists of a concurrent program visualized as a graph, each node of which is a Message Sequence Chart or MSC [5]. 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.

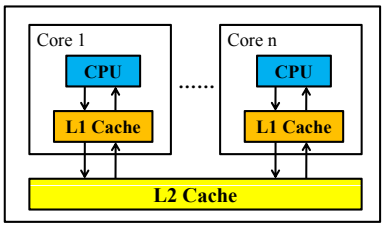


Fig. 2 A multi-core architecture with shared cache.

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 instruction cache interferences. Our method advances the state-of-the-art in shared cache multi-core timing analysis [39,40] 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. Second, 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. Finally, to further improve the worst-case execution time, we consider instruction cache locking to reduce the cache conflicts from tasks on different cores. In summary, (a)we develop a timing analysis method for shared cache multi-cores that enhances

the state-of-the-art and (b)we ensure more predictable memory accesses in the presence of shared instruction caches by a combination of analysis and system level changes via cache locking.

The rest of the paper is organized as follows. The next section presents the system model and architecture. Section 3 describes our WCRT analysis framework, and section 4 details the analysis components. In section 5, we introduce the cache locking optimization. Section 6 surveys related work. Experimental results appear in section 7, and section 8 concludes the paper.

2 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.

2.1 Message Sequence Charts

A Message Sequence Chart (MSC) [5] 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. 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. 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.

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.

2.2 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 [8].

2.3 DEBIE Case Study

Our case study consists of DEBIE-I DPU Software [15], 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

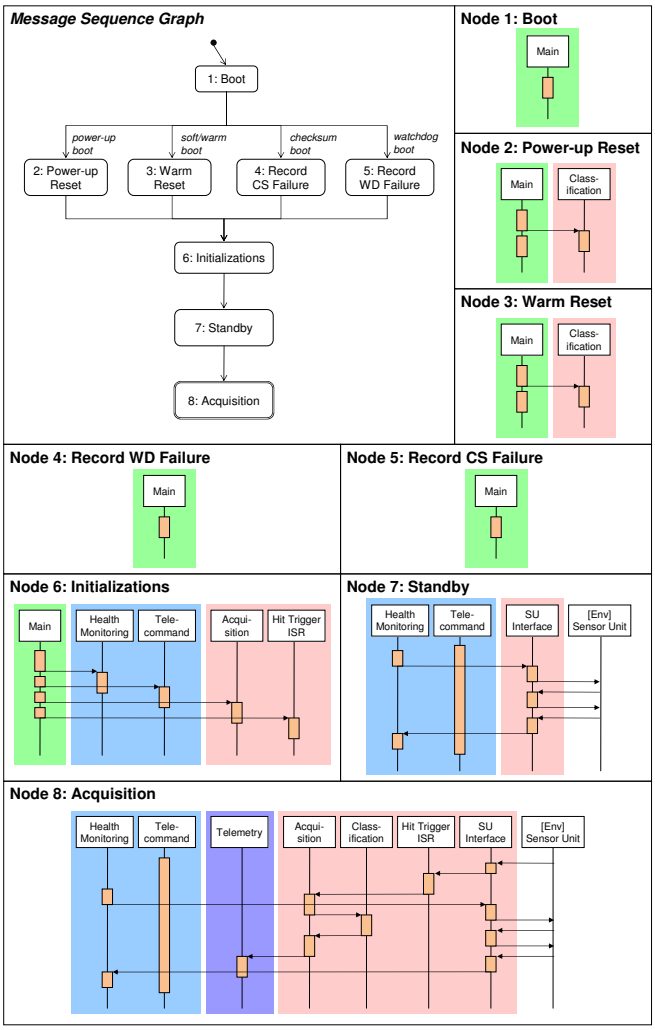


Fig. 3 DEBIE Case Study. Different colors are used to show the mapping of the processes to different processor cores.

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.

2.4 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 [19]. 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.

3 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. In fact the polynomial computational complexity (see section 4.5) allows our analysis to scale to a large number of cores. But shared cache itself may not always scale well to a large number of cores due to frequent inter-core evictions. Thus, in practice, we evaluate our technique using 2-core, 4-core and 8-core architectures as shown in section 7. 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 ([21,38,29,34]) 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 [19] 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

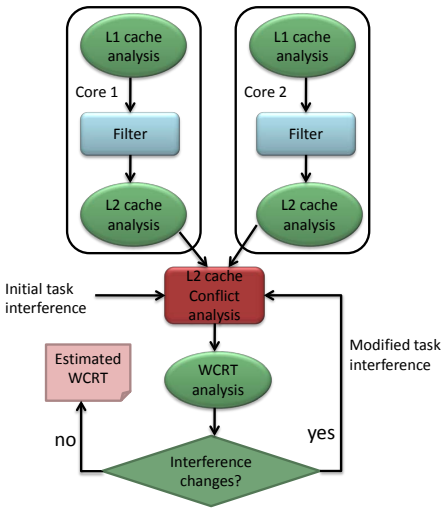


Fig. 4 Our Analysis Framework

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 shared L2 cache analysis proposed in the literature [39].

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 3.1 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.

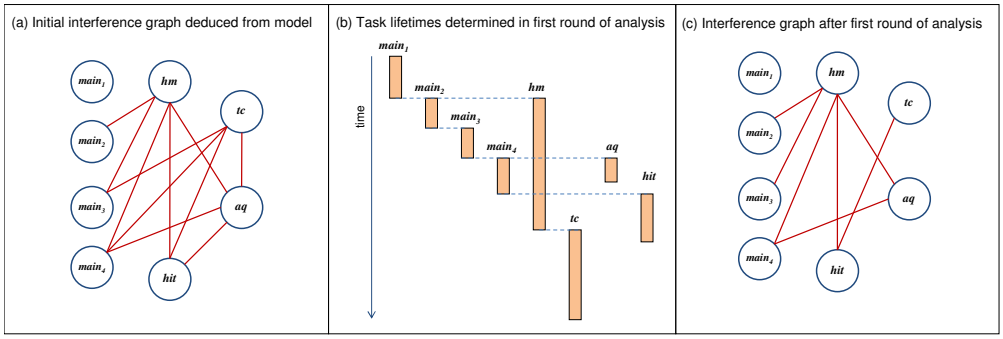


Fig. 5 The working of our shared-cache analysis technique on the example given in Figure 1

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.

3.1 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 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 anal-

ysis 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.

4 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 [19] 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.

4.1 Intra-Core Cache Analysis

The intra-core cache analysis step employs abstract interpretation method [37] 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 [37]. In this paper, we consider LRU replacement policy, but the cache analysis can be extended for other replacement polices as shown in [20].

– **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 [19]. 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. [19] 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.

4.2 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".

Optimization for Set-Associativity:

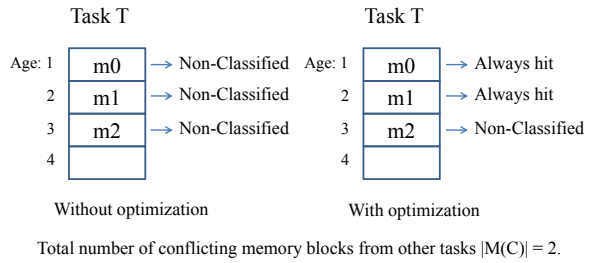


Fig. 6 An example of 4-way set associative L2 cache. The abstract cache state of task T for cache set C at a program point during *must analysis* is shown. Memory blocks are converted to either "Always Hit" or "Non-Classified" according to their ages and the number of conflicting memory blocks from interfering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

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 [37]. 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.

4.3 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 1 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 1 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 [35].

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

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

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.

4.4 Termination Guarantee

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

Theorem 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 1, 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 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 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.

4.5 Computational Complexity

In this subsection, we analyze the computational complexity of our approach. Our analysis framework consists of three components: intra-core cache analysis, L2 cache conflict analysis and WCRT analysis. As previously described, the last two components are invoked together in an iterative manner until the task interference does not change. In the following, we analyze the computational complexity of each component and the overall analysis. We assume there are M tasks $(t_1, \ldots, t_M)$ in the MSC. The complexity of our analysis is characterized by M and not the number of cores in the system.

Intra-core cache analysis. In this phase, we perform the must, may, and persistence analysis for each task in isolation. In other words, we assume that there are no conflicts among the tasks. In fact, must, may, and persistence analysis are fixed point data flow analysis and the computational complexity depends on the program control flow of the tasks and the cache architecture. The complexity of this phase is equivalent to standard single-core cache analysis for WCET estimation. Moreover, this phase is performed only once in our framework. Let w be the average cache analysis time per task. Then the computational complexity of this component is $T_{intra} = O(wM)$.

Iterative analysis. The L2 cache conflict analysis and WCRT analysis are invoked together iteratively as shown in Figure 4. We start with the worst-case task interference (i.e., tasks conflict with all the tasks on other cores except those excluded by partial order). The number of conflicting task pairs

(i.e., edges in the task interference graph) in the initial task interference graph can be bounded by $\frac{M\times(M-1)}{2}$. The iterative analysis terminates until there is no change to the task interference graph. Meanwhile, according to Theorem 2, *task interference monotonically decrease (strictly decrease or remain the same) across different iterations of our analysis framework.* Thus, the number of iterations of our iterative framework can be safely bounded by $\frac{M\times(M-1)}{2}$. Next, we first derive the computational complexity of one iteration and then summarize across all the iterations.

L2 cache conflict analysis. In this phase, for each pair of conflicting tasks (t_i, t_j), we compute the number of conflicting memory blocks introduced to task t_i by task t_j and vice versa. So for each task t we can collect the set of unique memory blocks from the tasks on other cores that conflict with t ($\mathcal{M}(C)$ in section 4.2). Let S be the number of sets in the shared cache and b be the average number of memory blocks per task. Then $\mathcal{M}(C)$ is upper bounded by $M \times \frac{b}{S}$. As we have S cache sets, the cost per task is $M \times \frac{b}{S} \times S = bM$. The process has to be repeated for all the tasks. So, the computational complexity of this component $T_{L2} = O(bM^2)$.

WCRT analysis. In this phase, we traverse the tasks in topology order. The topology sort of the tasks is performed only once before our iterative analysis. For each task t, we traverse all of its predecessors to compute the $EarliestReady[t]/LatestReady[t]$. The total number of predecessor relationship can be bounded by $\frac{M\times(M-1)}{2}$. To compute the $LatesteFinish[t]$, we need to sum the WCET of $t's$ peers. Recall that t_i and t_j are defined as peers if they are mapped to the same core and their execution lifetime overlaps and do not have dependencies. The total number of pairs of peers can be bounded by $\frac{M\times(M-1)}{2}$ too. Thus, the computational complexity of this component $T_{wcrt} = O(M^2)$.

Finally, we need to update the task interference graph: for each pair of conflicting tasks, we need to check whether their lifetime still overlap. This overhead $T_{update} = O(M^2)$.

The overall computational complexity of our analysis $T(M)$ is

$$\begin{aligned} T(M) &= T_{intra} + \sum_{X=1}^{X=\frac{M\times(M-1)}{2}} (T_{L2} + T_{wcrt} + T_{update}) \\ &= O(wM) + \sum_{X=1}^{X=\frac{M\times(M-1)}{2}} (O(bM^2) + O(M^2) + O(M^2)) \\ &= O(wM) + O(bM^4) \end{aligned}$$

In practice, our analysis converges after 2–3 iterations and thus the complexity is approximately $O(wM) + O(bM^2)$. We experimentally validate this using real-world and synthetic benchmarks on 2-core, 4-core and 8-core architectures (section 7) where the runtime is well under 6 minutes. Clearly, the complexity of our analysis is not dependent on the number of cores, but rather on the number of tasks. However, shared cache itself may not always scale well to a large number of cores due to the large number of conflicts in shared cache and

frequent inter-core evictions. As a result, the overestimation of timing analysis for such many-core shared cache architecture might be high.

5 Cache Locking Optimization

Cache locking was primarily designed to offer better timing predictability for hard real-time applications. Once a memory block is locked in the cache, it cannot be evicted from the cache under replacement policy. Thus, all the subsequent accesses to the locked memory blocks will be cache hits.

In the context of shared caches in multi-cores, a substantial source of unpredictability is the interference among the memory blocks accessed by the tasks executing on different cores but mapping to the same cache set. If the analysis cannot identify the task lifetimes to be disjoint, then we have to assume worst-case interference among the tasks in the shared cache. In this section, we resort to cache locking to avoid such interferences for memory blocks that are frequently accessed from the shared cache.

In our system architecture, each core has its private L1 cache and multiple cores share the same L2 cache. Thus we have the option of locking memory blocks either in the private L1 caches or the shared L2 cache. Locking the memory blocks into L1 caches certainly helps to improve the WCET of the current task (e.g., by locking memory blocks that cause a lot of cache misses on the WCET path in a task). In addition, more cache hits in the L1 cache implies less L2 cache accesses. Thus, the tasks running on the other cores may benefit as well due to the reduced L2 cache conflicts. It is also possible to lock memory blocks into the shared L2 cache. However, as L2 cache has longer latency compared to L1 cache, the WCET reduction is much less for the current task. More importantly, as the L2 cache size gets reduced after locking, the tasks on the other core might suffer considerably. Thus, we choose to lock memory blocks only in the L1 cache.

Locking Mechanisms: There exists two locking mechanisms — way locking and line locking. In way locking, the entire ways are locked for all the cache sets. Way-locking is employed by ARM processor series [3, 4]. Line locking is employed by Intel Xcale [1], ARM9 family and Blackfin 5xx family processors [2] allows different number of lines to be locked for different cache sets. Obviously, line locking is a fine grained locking mechanism compared to way locking. Thus, we consider line locking in this paper. Furthermore, recall that our system model is a message sequence graph (MSG) where each node is described as an MSC. We consider static instruction locking for each MSC. In other words, the memory blocks are locked in the cache at the beginning of execution of an MSC and remain locked throughout the execution of the MSC. The implication is that we need to pay for the time to load and lock the instructions into the cache before the execution of each MSC in the MSG.

5.1 Cost-Benefit Modeling for Cache Locking

The critical question that we need to answer for cache locking is how to select the memory blocks that should be locked. We need to perform a cost-benefit analysis to identify the most profitable memory blocks to lock in the cache.

To identify the profitable memory blocks, we first perform one round of our cache and WCRT analysis as described in Section 3. Then, we collect the profiles including the abstract cache states at L1 and L2 caches, task interference graph, memory access latency of each memory block (Table 1) and the execution frequency of each memory block on the WCET path. These information will be used in our cache locking modeling.

For memory block m, let lat_m be its worst-case access latency according to its classification in L1 and L2 caches (see Table 1) and f_m be its execution frequency on the WCET path, respectively. By locking memory block m into L1 cache, all the accesses to m will be cache hits; thus the WCET of the current task may be improved. We define the benefit for the current task as

$$Self_m = (lat_m - hit_{L1}) \times f_m$$

Meanwhile, locking m into the private L1 cache of the core eliminates all the accesses of m to the L2 cache. This may lead to reduced L2 cache conflicts for the tasks running on other cores with memory blocks mapped to the same L2 cache set as m. Let us assume m belongs to task T running on core P and it is mapped to cache set C in L2 cache before locking. Then, let $Conf(m)$ be the set of memory blocks belonging to the tasks running on other cores (not P) that can potentially access the cache set C in the L2 cache. Recall that in Section 4.2, for the memory block m, we will convert its access classification from "Always Hit" to "Non-Classified" if $|Conf(m)| > N_{L2} - age_{L2}(m)$ where $age_{L2}(m)$ is the age of m in the abstract cache state of L2 must analysis and N_{L2} is the associativity of the L2 cache. By locking m, we reduce the L2 cache conflicts for the tasks running on other cores. Thus we might be able to avoid the conversion of some "Always Hit" references to "Non-Classified" due to conflicts. If memory block m is converted from L2 "Non-Classified" to L2 "Always Hit", then the WCET reduction is

$$Gain_m = f_m \times (miss_{L2} - hit_{L2})$$

Therefore, the total benefit for the other cores after locking m is

$$Other_m = \sum_{m' \in Conf(m) \wedge (|Conf(m')| \; + \; age_{L2}(m') = N_{L2}+1)} Gain_{m'}$$

On the other hand, locking memory block m may have negative impact on the memory blocks mapped to the same set in the private L1 cache of the same core as the associativity for the private L1 cache is reduced through locking. Let $Same(m)$ be the set of memory blocks mapped to the same cache set as m in the private L1 cache of the core. From the L1 abstract cache states during must analysis, we can easily find the age of these memory blocks. If the age

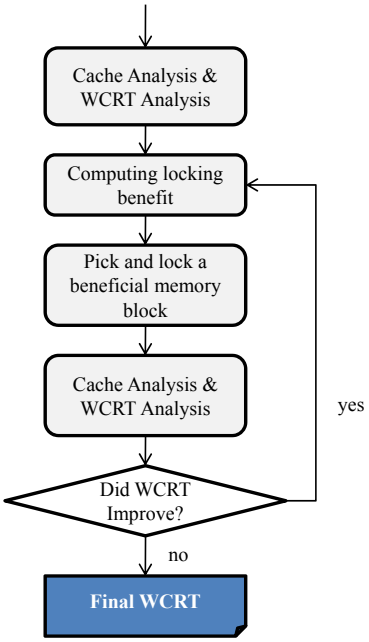


Fig. 7 Cache locking framework

of a conflicting memory block $m' \in Same(m)$ is $N_{L1} - 1$ where N_{L1} is the L1 cache associativity, then m' will be converted to L1 miss after locking m. In the worst case, m' will also be classified as "Non-Classified" L2 accesses. So, we define the cost of locking m as

$$Cost_m = \sum_{m' \in Same(m) \wedge age_{L1}(m') = N_{L1}-1} (miss_{L2} - hit_{L1}) \times f_m$$

where $age_{L1}(m')$ is the age of m' in the abstract cache state of L1 must analysis. Then, we define the overall benefit as

$$Benefit_m = Self_m + Other_m - Cost_m$$

Note that, we use $Benefit_m$ to evaluate and compare the benefit of locking different memory blocks such that we can quickly identify some beneficial memory blocks for locking. $Benefit_m$ may not be the actual WCRT reduction as both the BCET and the WCET path may change after cache locking. Thus, the actual WCET reduction may be more or less than we what predict. Also, the task interference graph may change due to the change of BCET and WCET values. But in practice, we find that $Benefit_m$ is a good metric to evaluate the benefit of locking different memory blocks.

The overall cache locking framework is shown in Figure 7. We first perform our cache and WCRT analysis before the iterative process. Then, in each

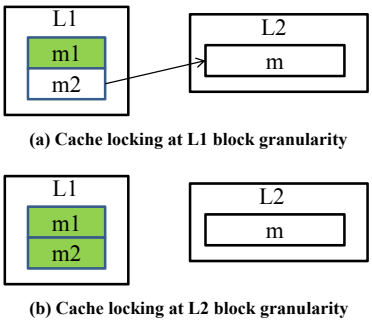


Fig. 8 Cache locking granularity

iteration, we compute the $Benefit_m$ for all the unlocked memory blocks so far. If a cache set is fully locked, we will not consider the memory blocks mapped to that cache set. Then, we select the memory block with the maximum $Benefit_m$ for locking. We break the ties arbitrarily. After that, we perform cache and WCRT analysis to derive the precise WCRT after cache locking. If WCRT is improved, we continue to lock; otherwise we stop the process.

Cache Locking Granularity: So far, we assumed that L1 and L2 caches have identical block size. However, in reality the block size of L2 cache can be greater than or equal to the block size of L1 cache. We can choose locking either at L1 or L2 block granularity. Figure 8 shows the differences between the two locking granularities. In this example, L2 block size is assumed to be twice as big as L1 block size. $m1$ and $m2$ are two consecutive memory blocks in L1 cache and both of them correspond to memory block m in L2 cache. If we choose to lock at L2 memory block granularity, then we have to lock both $m1$ and $m2$ in L1 cache simultaneously. More importantly, the references to memory block m will not access the L2 cache any more. However, we can not guarantee this if we choose to lock at L1 memory block granularity. For example, if we choose to lock $m1$ into L1 cache, there might still be accesses of m at L2 cache level due to miss of $m2$ in L1 cache. Thus, the L2 cache conflicts are not reduced. So, if we choose to lock at L1 granularity level, we will not include the benefit from other cores ($Other_m$) in the final $Benefit_m$. We will explore both locking granularities in the experiments.

6 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 ([7,37]) which also forms the foundation to the framework presented in this paper. Mueller [28] extends the technique for multi-level cache analysis; Hardy and Puaut [19] further adjust the method with a crucial observation to produce safe estimates for set-associative caches. Other pro-

posed methods that attempt exact classification of memory accesses for private caches include data-flow analysis [28], integer linear programming [24] and symbolic execution [27].

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 ([21,38,29]). 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 [34]. A different perspective in [36] 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 [33], [14] and [22]. 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 [32]. Gustavsson et al. [17] perform WCET analysis of multicore architectures through model checking.

One technique in literature that has addressed inter-core shared-cache analysis so far is the one proposed by Yan and Zhang [39,40]. 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 [19] (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 obtains improved estimates by exploiting the knowledge about overlap of time intervals for different tasks. Hardy et al. [18] bypass static single usage blocks from the shared caches, and only blocks statically known to be reused are cached. Their approach reduces the pollution in shared caches, thus, tightens the WCET estimates for multi-core processors with shared instruction caches. However, they also do not consider the execution time intervals of tasks.

Chattopadhyay and Roychoudhury [12] develop a compile-time scratchpad allocation framework for multi-processor platforms, where the processors virtually share on-chip scratchpad space and external memory is accessed through a shared bus. They adopt a static bus schedule scheme (Time Division Multiple Access) which is incorporated by scratchpad allocation method. Overall

WCRT is significantly reduced by appropriate content selection and overlay optimization (variables share the same scratchpad space due to disjoint lifetimes). There are also studies employing cache locking to improve the timing predicability. Puaut and Decotigny proposed two low-complexity algorithms for static cache locking in a multi-tasking environment [31]. System utilization or inter-task interferences are minimized through static cache locking [31]. Campoy et al. employed generic algorithms to select contents for locking in order to minimize system utilization [11]. However, the WCET path may change after some functions are locked into the instruction cache and the change of the WCET path is not handled in [31,11]. Falk et al. considered the change of the WCET path and showed better WCET reduction [16]. Liu et al. [26] formulated the instruction cache locking for minimizing WCET as linear programming model and showed that the problem is NP-Hard problem. However, all the techniques consider cache locking at function level (e.g. the entire function is either locked or not locked). In this paper we consider cache locking at finer granularity — cache line level, which provides more opportunities for optimization. More recently, Liang and Mitra [25] have shown that cache locking is quite effective for improving the execution time of general embedded applications as well.

7 Experimental Evaluation

In this section, we evaluate our WCRT analysis and cache locking technique with real-world and synthetic applications. We first perform a case study of DEBIE-I DPU Software [15], an in-situ space debris monitoring instrument developed by Space Systems Finland Ltd. 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. We further validate the effectiveness of our technique using Unmanned Aerial Vehicle (UAV) control application from PapaBench [30] and a synthetic benchmark. The tasks of the synthetic benchmark are from WCET benchmark suite and we use TGFF [6] to generate MSG. The MSG of PapaBench and synthetic benchmark are shown in Figure 9 and Figure 10.

We compile these benchmarks for SimpleScalar PISA (Portable ISA) instruction set [9] — 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 [23]. Details of the tasks in the DEBIE benchmark and their code-sizes appear in Figure 11 and Table 2. The table also shows the mapping of the tasks to the processor cores in a system with four cores. The details of the tasks in PapaBench and synthetic benchmarks are shown in Table 3 and 4, respectively. Some tasks of the synthetic benchmarks have very few L2 accesses due to their small code size. Therefore, we choose to partially unroll the loops for these tasks to increase the code size.

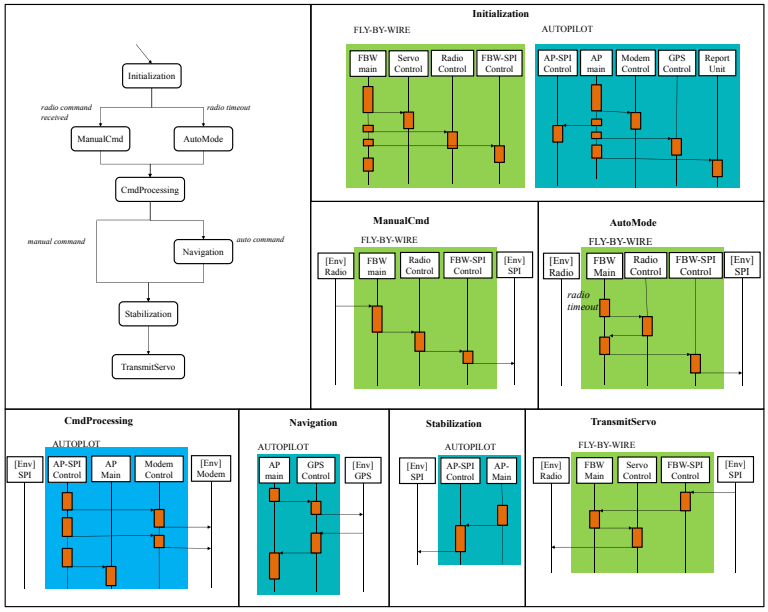


Fig. 9 Message Sequence Graph of the PapaBench application.

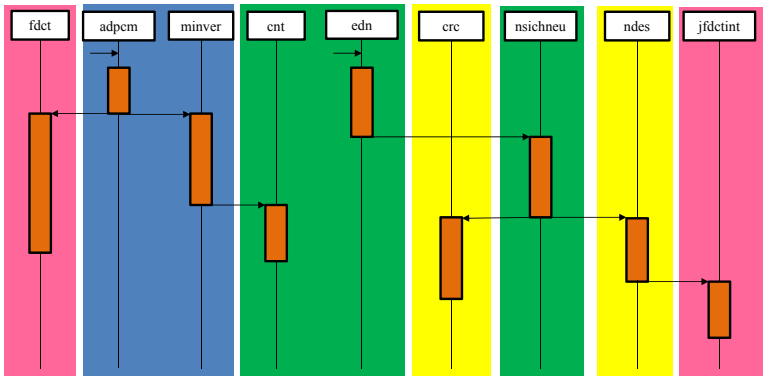


Fig. 10 Message Sequence Chart of synthetic benchmark.

As we are modeling the cache, we assume a simple in-order processor with unit-latency for all data memory references. We perform all the experiments on a 2.5GHz 8-core Xeon CPU with 16GB memory.

In the following, we perform four sets of experiments. First, we compare our analysis with Yan-Zhang's method using DEBIE case study. We also provide experimental results for PapaBench and synthetic benchmarks. Then, we present the cache locking results. Finally, we show the accuracy results of our analysis.

Table 2 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	3
3	wr_main_1	3,408	1
	wr_main_2	5,952	1
	wr_class	1,648	3
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	4
	$init\_telecm$	4,408	3
	$init\_acqui$	200	4
	$init\_hit$	616	2
7	sby_health_1	16,992	4
	sby_health_2	448	4
	sby_telecm	23,288	3
	sby_su_1	6,512	2
	sby_su_2	4,392	2
	sby_su_3	1,320	2
8	acq_health_1	16,992	4
	acq_health_2	448	4
	acq_telecm	23,288	3
	acq_acqui_1	3,136	4
	acq_acqui_2	3,024	4
	acq_telemt	3,768	1
	acq_class	3,064	3
	acq_hit	8,016	2
	acq_su_0	2,536	2
	acq_su_1	6,512	2
	acq_su_2	4,392	2
	acq_su_3	1,320	2

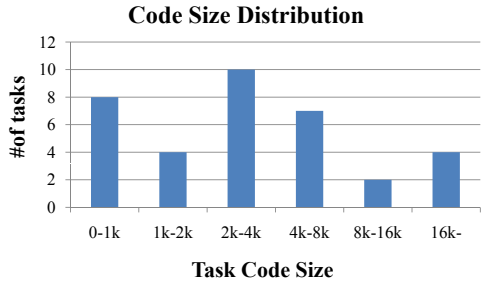


Fig. 11 Code size distribution of DEBIE benchmark.

7.1 Case Study of DEBIE Benchmark

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

Table 3 Characteristics and settings of the PapaBench.

MSC	Task	Codesize (bytes)	Core
1	*fbw_main_initservo*	816	1
	fbw_main_initradio	96	1
	fbw_main_initspi	96	1
	fbw_main_initend	1,696	1
	fbw_servo_init	528	1
	fbw_radio_init	384	1
	fbw_spi_init	272	1
	ap_main_initmodem	768	2
	ap_main_initspi	96	2
	ap_main_initgps	96	2
	ap_main_initreport	1,264	2
	ap_modem_init	352	2
	ap_spi_init	560	2
	ap_gps_init	392	2
	ap_report_init	5,520	2
2	*fbw_main_mancmd*	144	1
	fbw_radio_mancmd	464	1
	fbw_spi_mancmd	992	1
3	*fbw_main_autocmd*	208	1
	fbw_radio_autocmd	464	1
	fbw_main_gencmd	144	1
	fbw_spi_autocmd	992	1
4	*ap_spi_cmd*1	2,752	2
	*ap_spi_cmd*2	1,728	2
	*ap_spi_cmd*3	176	2
	*ap_modem_update*1	2,304	2
	*ap_modem_update*2	6,496	2
	ap_main_getcmd	1,536	2
5	*ap_main_nav*1	20,240	2
	*ap_gps_nav*1	1,040	2
	*ap_gps_nav*2	1,296	2
	*ap_main_nav*2	13,920	2
6	*ap_main_stabil*	96	2
	ap_spi_sendnav	656	2
7	*fbw_spi_rcvnav*	1,840	1
	fbw_main_getnav	256	1
	fbw_servo_nav	656	1

Table 4 Characteristic of synthetic benchmark.

Benchmarks	Code Size (bytes)	LoC	Core
adpcm	12,480	876	3
cnt	5.368	224	4
crc	4,200	162	1
edn	11,000	336	4
fdct	9,936	401	2
jfdctint	12,048	511	2
minver	6,256	201	3
ndes	6,352	230	1
nsichneu	63,744	4,032	4

estimation result following Yan-Zhang's technique [39,40]. 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.

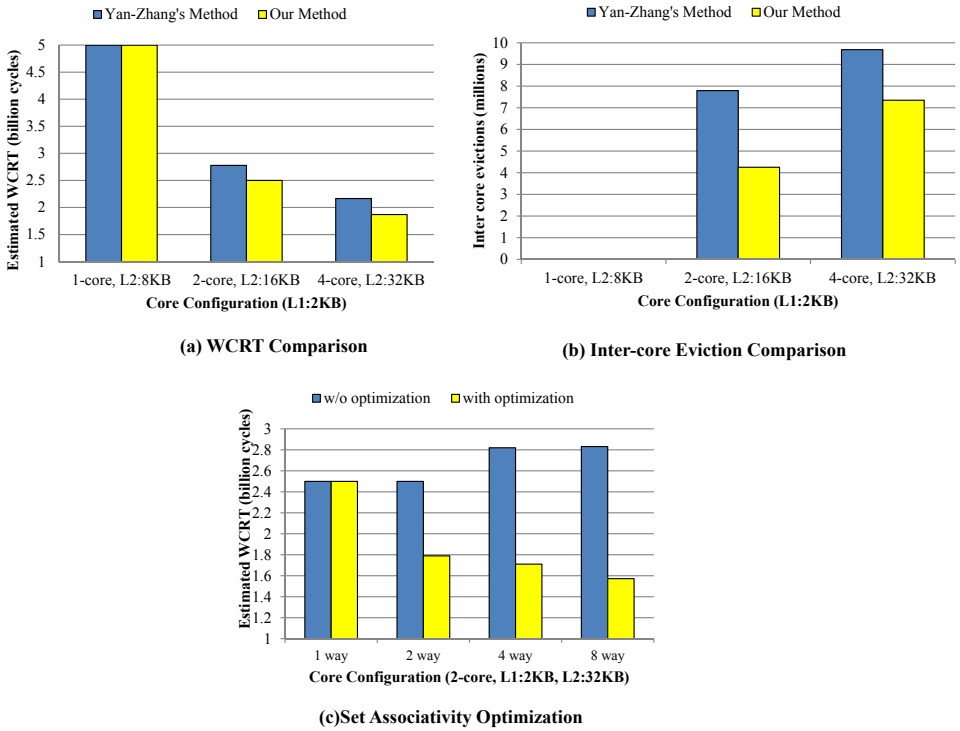
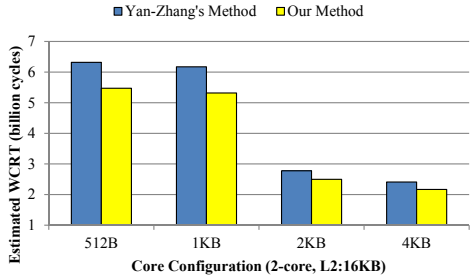


Fig. 12 Comparison between Yan-Zhang's method and our method and the improvement of set associativity optimization of DEBIE benchmark).

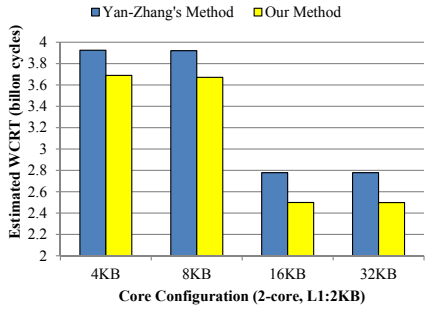
7.1.1 Comparison with Yan-Zhang's method

Yan-Zhang's analysis [39,40] is restricted to direct mapped cache. Thus, to make a fair comparison, we first configure both L1 and L2 as direct mapped caches. Figure 12(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 32-byte block size. The L2 cache has 64-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 14% more accuracy over Yan-Zhang's method. As tasks are distributed on more cores, the parallelization of task execution help to reduce the overall runtime as shown in Figure 12(a).

In Figure 12(b), we compare the number of inter-core cache evictions estimated by both methods for the same configurations as in Figure 12(a). When



(a) Varying L1 Size



(b) Varying L2 Size

Fig. 13 Comparison of estimated WCRT between Yan-Zhang's method and our method for varying L1 and L2 cache sizes of DEBIE benchmark).

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 12(b). This explains the WCRT improvement in Figure 12(a).

7.1.2 Set associative caches

Our method is able to handle set-associative caches accurately by taking into account the age of the memory blocks. Figure 12(c) compares the estimated WCRT with and without the optimization for set-associativity (see Section 4.2) 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 32-byte block size and L2 cache is configured as a 32KB set-associative cache with 64-byte block size, but varied associativity (1, 2, 4, 8). As shown in Figure 12(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.

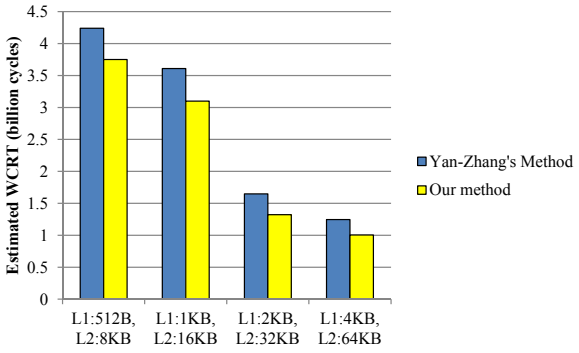


Fig. 14 WCRT estimation results on 8-core processor and comparison with Yan-Zhang's method under various configurations.

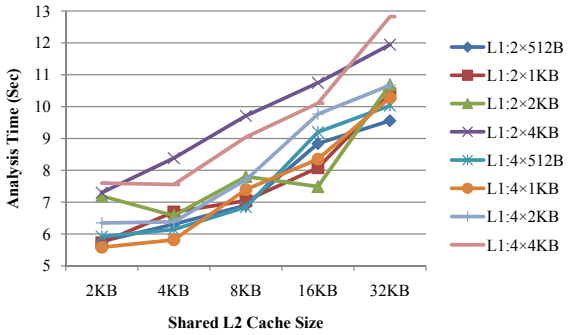


Fig. 15 Runtime of our iterative analysis.

7.1.3 Sensitivity to L1 cache size

Figure 13(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 14% more accuracy over Yan-Zhang's method.

7.1.4 Sensitivity to L2 cache size

Figure 13(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

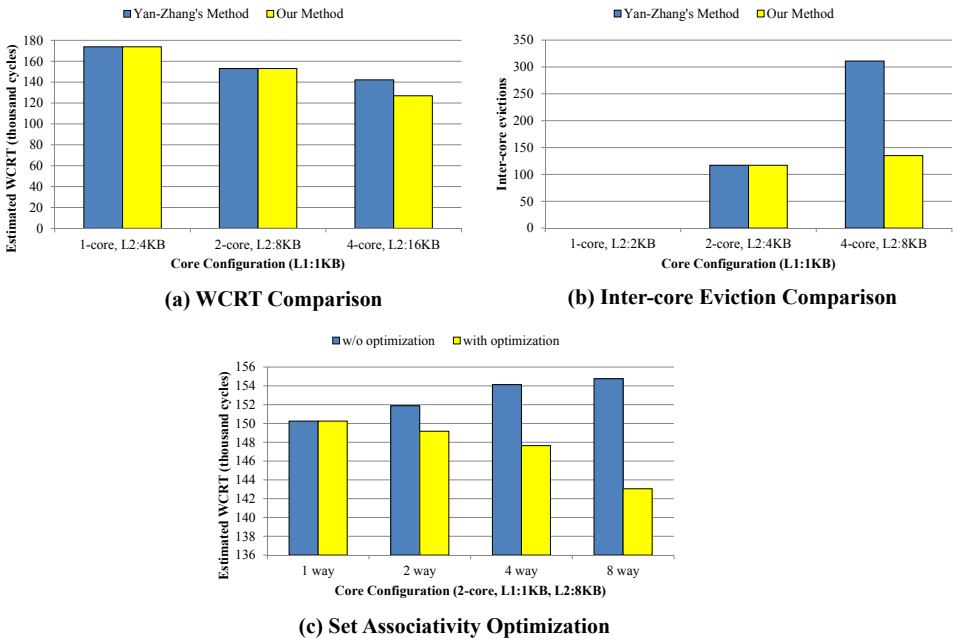


Fig. 16 Comparison between Yan-Zhang's method and our method and the improvement of set associativity optimization for PapaBench.

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.

7.1.5 8-core Setting

Figure 14 presents the comparison of estimated WCRT between Yan-Zhang's analysis and ours for a 8-core setting with various cache configurations. As tasks are distributed to 8 cores, there are more inter-core evictions than 4-core settings. Our method achieves up to 20% improvement over Yan-Zhang's method.

7.1.6 Runtime

Figure 15 sketches the runtime of our complete iterative analysis (L2 cache and WCRT analysis) for various configurations (e.g. different cores and cache sizes) of DEBIE benchmark. It takes less than 13 seconds to complete our analysis for any considered settings.

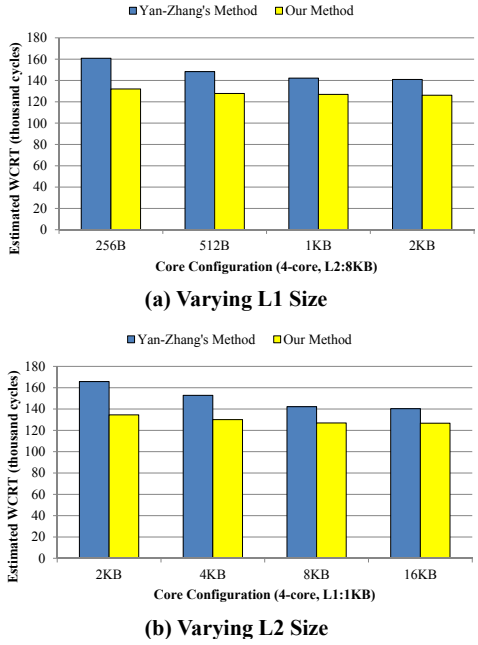


Fig. 17 Comparison of estimated WCRT between Yan-Zhang's method and our method for varying L1 and L2 cache sizes for PapaBench.

7.2 PapaBench and Synthetic Benchmarks

For PapaBench, we first evaluate our analysis in terms of the aforementioned three perspectives: WCRT comparison, inter-core eviction comparison, and set associativity optimization. The results are shown in Figure 16. For PapaBench, our method achieves up to 11% more accuracy over Yan-Zhang's method in terms of WCRT estimation. The set associativity optimization also improves the estimated WCRT significantly for high associative cache. The sensitivity to L1 and L2 cache size results are shown in Figure 17. As shown, our analysis achieves lower WCRT estimation for various settings compared to Yan-Zhang's method. All the experiments are performed using the same setting as in section 7.1.

The results for the synthetic benchmark are shown in Figure 18 and 19. Figure 18 shows the WCRT comparison, inter-core eviction comparison, and set associativity optimization, respectively. Our method achieves up to 8% improvement in WCRT estimation compared to Yan-Zhang's method. The set associativity optimization again improves the estimated WCRT significantly for highly associative caches. Figure 19 presents the sensitivity of our analysis to different L1 and L2 cache configurations, and our method produces lower estimated WCRT in all configurations.

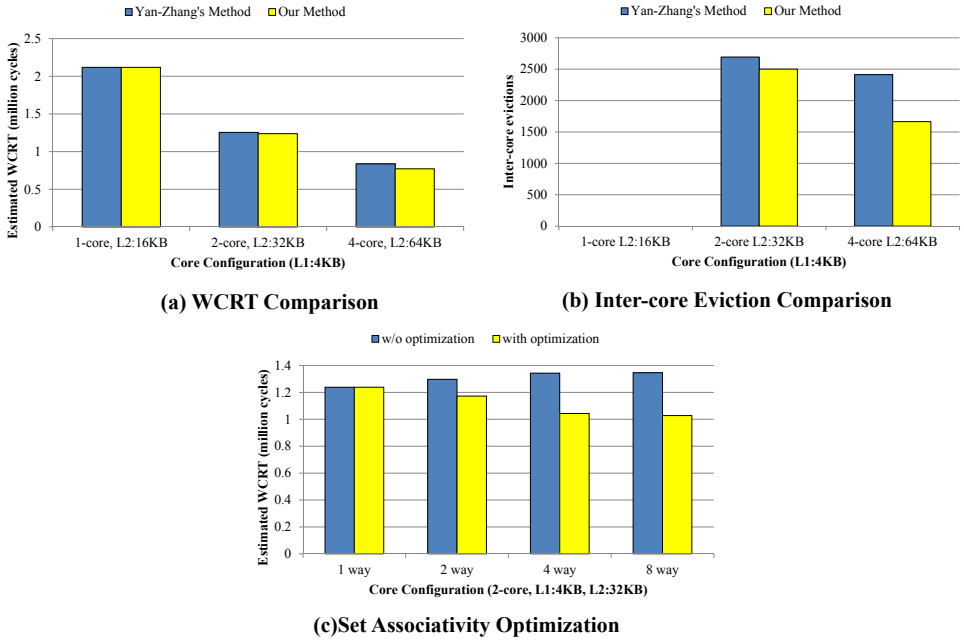
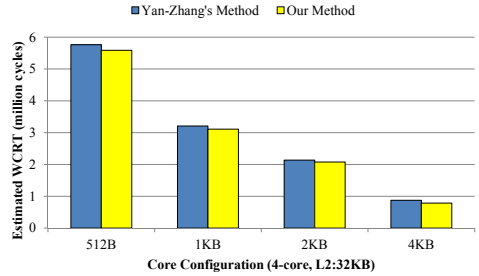


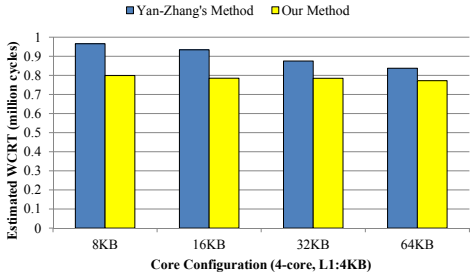
Fig. 18 Comparison between Yan-Zhang's method and our method and the improvement of set associativity optimization for synthetic benchmark.

Results without Synchronization. The improved accuracy of our analysis stems from both dependency analysis among the tasks as well as lifetime analysis among the tasks without dependency. In order to evaluate the strength of our lifetime analysis, we removed all the synchronization edges in the synthetic benchmark shown in Figure 10, i.e., there is no data dependency among the tasks. The task mapping on 4 cores remains the same, as shown in Table 4. L1 cache is configured as 4KB direct mapped cache with 32-byte block size and L2 cache is configured as a 64KB direct mapped cache with 64-byte block size.

The comparison between Yan-Zhang's method and our method is shown in Figure 20 (note that the scale of this graph is different from Figure 18 and 19). Our method has lower estimated WCRT due to fewer estimated inter-core evictions in the L2 cache. In our method, lifetime of each task is considered. Two tasks whose lifetime does not overlap should not conflict with each other in L2 cache. Compared to the results with synchronization for the same configuration (Figure 18 (b)), Yan-Zhang's method estimates the same number of inter-core evictions (2,413). While for our method, the number of evictions increases from 1,664 to 2,129 after the elimination of synchronization, as more tasks may conflict with each other now. The absolute WCRT value, however, decreases as the elimination of synchronization creates more parallelism opportunity.

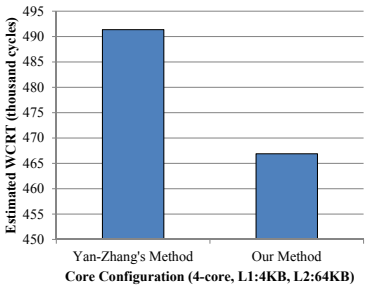


(a) Varying L1 Size

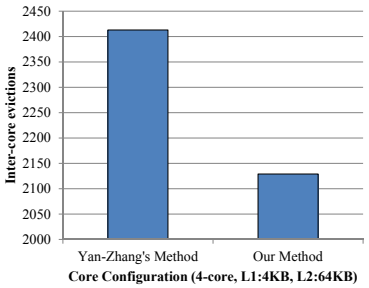


(b) Varying L2 Size

Fig. 19 Comparison of estimated WCRT between Yan-Zhang's method and our method for varying L1 and L2 cache sizes for synthetic benchmark.



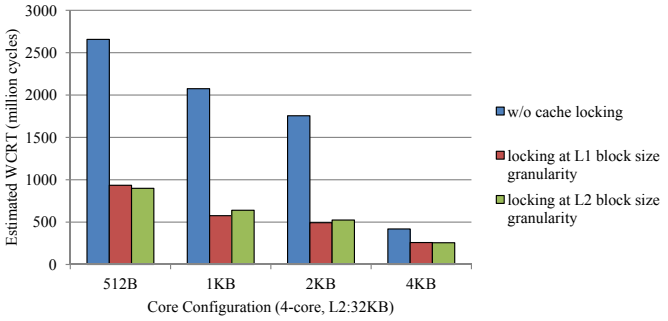
(a) WCRT Comparison



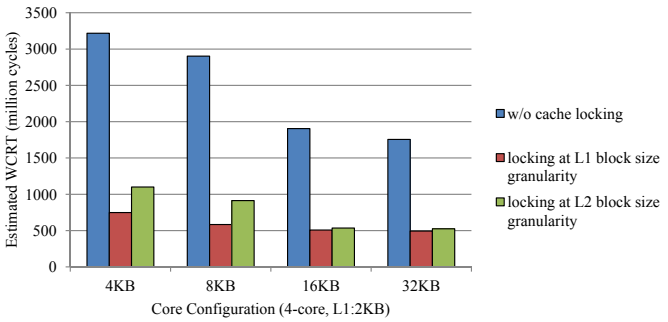
(b) Inter-core Eviction Comparison

Fig. 20 Comparison between Yan-Zhang's method and our method for synthetic benchmarks without synchronization.

Our complete iterative analysis (L2 cache and WCRT analysis) runs very fast for PapaBench and synthetic benchmarks. For any tested settings, it takes only a few seconds to complete the analysis.



(a) Varying L1 size



(b) Varying L2 size

Fig. 21 Estimated WCRT with and w/o cache locking at different locking granularities for DEBIE benchmark.

7.3 Cache Locking Results

In this section, we evaluate the effectiveness of cache locking in WCRT improvement. We employ cache locking at both L1 and L2 block size granularities as discussed in Section 5. We consider a 4-core system. L1 cache is configured as 2-way set associative cache with 32-byte block size and L2 cache is configured as 4-way set associative with 64-byte block size. Note that the cache locking overhead (i.e. the execution of lock instructions) are included in the WCRT computation.

Figure 21 shows the estimated WCRT with and without cache locking. In Figure 21(a), we vary the L1 cache size but keep the L2 cache size constant at 32KB. In Figure 21(b), we vary the L2 cache size but keep the L1 cache size constant at 2KB. In Table 5, we provide more detailed results for each individual MSC (Table 2) for a particular cache size setting: 2KB L1 cache and 8KB shared L2 cache. As shown, cache locking significantly improves the WCRT by up to 80% across various cache settings. Both locking granularities show promising WCRT improvement. Overall L1 block size granularity per-

Table 5 Estimated WCRT with and w/o cache locking for individual MSCs DEBIE benchmark.

MSC	**w/o cache locking**	**locking at L1 block size granularity**	**improvement** (%)	**locking at L2 block size granularity**	**improvement (%)**
1	7,741	6328	18.25	4,933	36.27
2	29,790	27,590	7.39	28,404	4.65
3	11,524	6,520	43.42	6,202	46.18
4	11,524	6,520	43.42	6,202	46.18
5	19,742	17,374	11.99	16,028	18.81
6	137,176,212	39,646,903	71.10	40,783,962	70.27
7	979,875,684	154,698,198	84.21	228,064,576	76.73
8	1,785,859,974	389,844,371	78.17	644,563,187	63.91

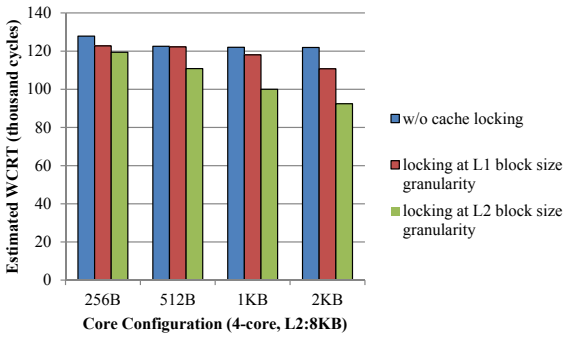
Table 6 Estimated cache misses with and w/o cache locking for DEBIE benchmark.

Cache	**w/o cache locking**	**granularity at L1 block size**	**improvement** (%)	**granularity at L2 block size**	**improvement (%)**
core 1	492	430	12.60	417	15.24
core 2	3,410,485	2,375,979	30.33	2,827,827	17.08
core 3	112	116	-3.57	116	-3.57
core 4	7,202,040	6,492,711	9.85	5,140,950	28.62
shared L2 cache	19,399,164	2,390,423	87.68	4,878,932	74.85

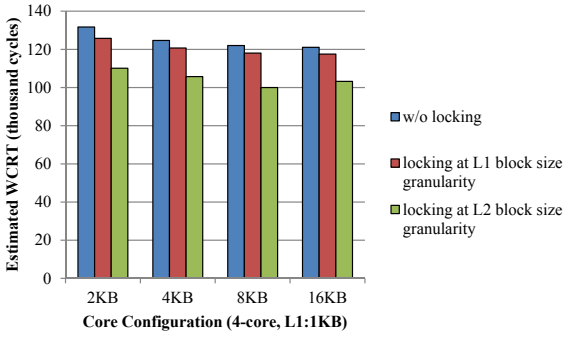
forms better than L2 block size granularity. This is because we may lock some memory blocks that are not on the WCET path if we lock at coarser granularity. Furthermore, locking bigger memory blocks into L1 cache implies less locking opportunities for the remaining memory blocks given the fixed cache size.

Table 6 presents the detailed cache misses with and without cache locking for DEBIE benchmark. The size of L1 and L2 caches are 2KB and 8KB, respectively. Rows (core 1, core 2, core 3, and core 4) show the estimated number of memory accesses, which are L1 cache misses but L2 cache hits, for each core. The last row (shared L2 cache) presents the number of shared L2 cache misses. As shown in Table 6, our cache locking significantly reduces the L2 cache misses and slightly improves the L1 cache misses for most of the cores. Recall that the performance metric in section 5 models the overall access latency of the memory hierarchy. The reduction of L2 cache misses can effectively improves the overall WCRT as the latency of L2 miss is much longer than that of L1 miss. It is possible that our cache locking slightly increases the L1 cache misses for some cores (see core 2).

The Estimated WCRT with and without locking for PapaBench and synthetic benchmarks are shown in Figure 22 and 23, respectively. The corresponding cache misses improvement are shown in Table 7 (L1:1KB, L2:8KB) and Table 8 (L1:2KB, L2:32KB).



(a) Varying L1 size



(b) Varying L2 size

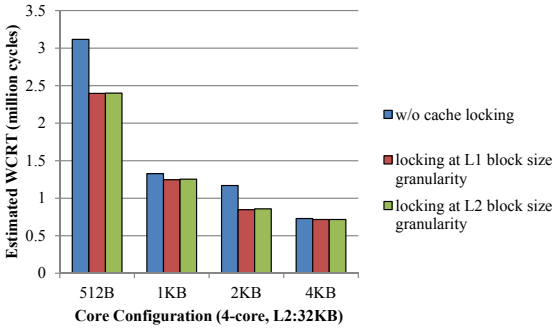
Fig. 22 Estimated WCRT with and w/o cache locking at different locking granularities for PapaBench.

Table 7 Estimated cache misses with and w/o cache locking for PapaBench.

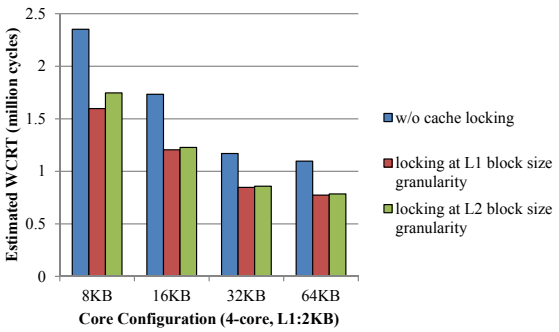
Cache	**w/o cache locking**	**granularity at L1 block size**	**improvement** (%)	**granularity at L2 block size**	**improvement** (%)
core 1	37	25	32.43	24	35.14
core 2	166	128	22.89	176	-6.02
core 3	479	475	0.84	570	-19.00
core 4	151	137	9.27	153	-1.32
shared L2 cache	1190	1108	6.89	934	21.51

Table 8 Estimated cache misses with and w/o cache locking for synthetic benchmark.

Cache	**w/o cache locking**	**granularity at L1 block size**	**improvement** (%)	**granularity at L2 block size**	**improvement** (%)
core 1	9112	4509	50.52	5819	36.14
core 2	928	928	0.00	928	0.00
core 3	1146	1146	0.00	1146	0.00
core 4	2012	2012	0.00	2012	0.00
shared L2 cache	8249	5371	34.89	5400	34.54



(a) Varying L1 size



(b) Varying L2 size

Fig. 23 Estimated WCRT with and w/o cache locking at different locking granularities for synthetic benchmark.

7.3.1 8-core Setting

Figure 24 presents the estimated WCRT improvement for a 8-core setting. As shown, cache locking significantly improves the WCRT by up to 50% across various cache settings.

7.3.2 Runtime

Figure 25 shows the runtime of our iterative cache locking process for both locking granularities across various settings for DEBIE benchmark. As shown, our cache locking algorithm is very efficient. It takes less than 6 minutes for any considered setting. The cache locking analysis for PapaBench and synthetic benchmark completes within 1 minute.

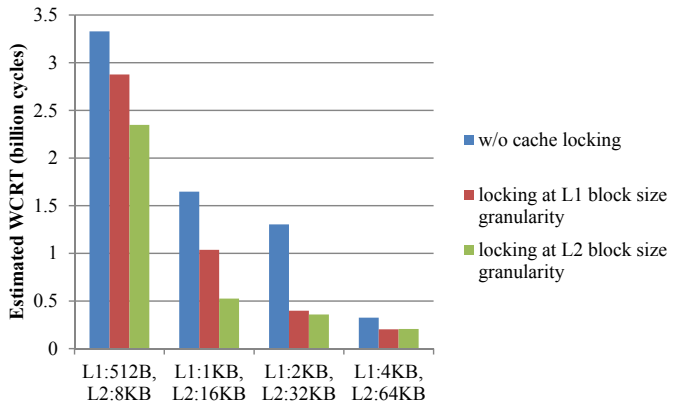


Fig. 24 Cache locking results for 8-core processor of various configurations for DEBIE benchmark.

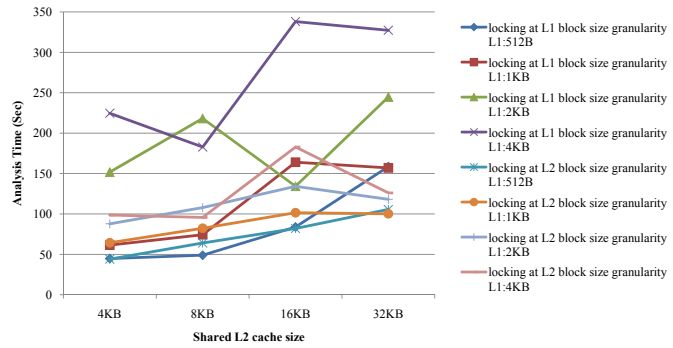


Fig. 25 Runtime of WCRT analysis with cache locking.

7.4 WCRT Analysis Accuracy

In this subsection, we evaluate the accuracy of our WCRT estimates. Evaluating the accuracy of the WCRT estimate requires us to obtain observed WCRT through simulation. Even though we can compile and analyze DEBIE benchmark and estimate its WCRT, there are difficulties in carrying out cycle-accurate simulation of the entire benchmark program. The original DEBIE software receives its input from multiple sensors as well as uses multiple timer interrupts. When porting to other platforms that may not have these peripherals, the behavior of the peripherals are simulated through some "harness" functions. In other words, we need to use simulated peripherals to run this benchmark. Unfortunately, we cannot port these simulated peripheral codes to SimpleScalar infrastructure that we use as our platform. The simulated pe-

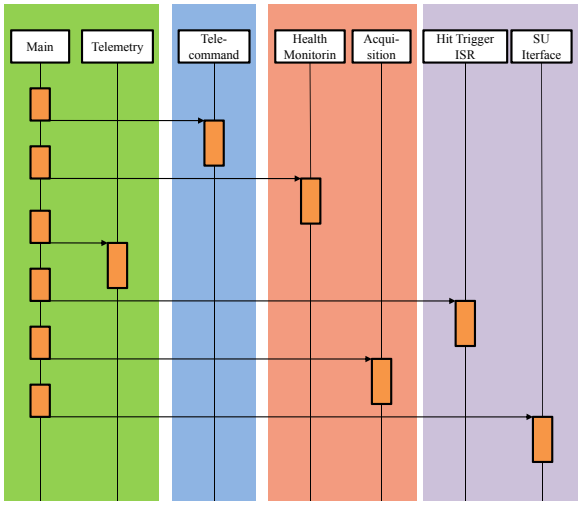


Fig. 26 MSC used to obtain simulation results.

Table 9 Characteristic and settings of the MSC used in simulation.

Task	Codesize (bytes)	Core
main-test-tc	240	1
main-test-hm	240	1
main-test-tm	240	1
main-test-hit	240	1
main-test-aq	240	1
main-test-su	240	1
tm-test	56, 960	1
hit-test	10,776	2
su-test	50,176	2
tc-test	45,368	3
hm-test	44,176	4
aq-test	44,128	4

ripherals communicate via RTX kernel with proprietary code. Thus it is not possible for us to simulate the entire DEBIE benchmark.

However, the DEBIE benchmark includes another program that tests the different software components. This particular testing program does not require any peripherals to generate input data as the test inputs are generated internally. Thus we choose to use the MSC corresponding to this testing program to evaluate the accuracy of our WCRT analysis. Details of this MSC are shown in Figure 26 and Table 9. In addition to DEBIE benchmark, we can obtain the simulation results for PapaBench and synthetic benchmarks. We compare the estimation with simulation for all the them.

We consider a 4-core system and the corresponding mapping of the tasks to the processor cores appear in Figure 26. The L1 cache is configured as 2-way set associative cache with 32-byte block size. The shared L2 cache is configured as 4-way set associative cache with 64-byte block size.

Table 10 WCRT estimation accuracy with cache locking for DEBIE benchmark.

Configuration	locking at L1 block size granularity			locking at L2 block size granularity		
	Est. WCRT (cycles)	Obs. WCRT (cycles)	Over Est.	Est. WCRT (cycles)	Obs. WCRT (cycles)	Over Est.
L1:2KB L2:4KB	195,365,114	170,997,708	1.14	196,953,375	170,723,730	1.15
L1:2KB L2:8KB	190,462,524	155,539,731	1.22	194,421,313	155,411,040	1.25
L1:2KB L2:16KB	189,618,571	152,009,715	1.25	192,322,099	151,895,604	1.27
L1:2KB L2:32KB	189,211,591	151,577,355	1.25	191,123,029	151,479,444	1.26
L1:4KB L2:8KB	167,275,370	151,513,653	1.10	169,693,178	151,310,262	1.12
L1:4KB L2:16KB	167,820,768	147,722,133	1.14	170,242,896	147,708,012	1.15
L1:4KB L2:32KB	167,127,572	147,398,223	1.13	169,559,690	147,393,552	1.15

Table 11 WCRT estimation accuracy with cache locking for PapaBench.

Configuration	locking at L1 block size granularity			locking at L2 block size granularity		
	Est. WCRT (cycles)	Obs. WCRT (cycles)	Over Est.	Est. WCRT (cycles)	Obs. WCRT (cycles)	Over Est.
L1:512B L2:4KB	125,700	61,380	2.05	112,726	74,145	1.52
L1:1KB L2:8KB	118,032	63,592	1.86	100,007	67,944	1.47
L1:1KB L2:4KB	120,687	60,012	2.01	105,722	66,789	1.58
L1:2KB L2:8KB	110,736	55,350	2.00	92,452	57,720	1.60
L1:2KB L2:16KB	110,232	55,368	1.99	96,259	58,302	1.65

Table 12 WCRT estimation accuracy with cache locking for synthetic benchmark.

Configuration	locking at L1 block size granularity			locking at L2 block size granularity		
	Est. WCRT (cycles)	Obs. WCRT (cycles)	Over Est.	Est. WCRT (cycles)	Obs. WCRT (cycles)	Over Est.
L1:2KB L2:32KB	846,924	467,111	1.81	858,500	476,669	1.80
L1:2KB L2:64KB	774,204	432,101	1.79	785,060	441,659	1.78
L1:4KB L2:32KB	717,265	425,639	1.69	717,067	426,476	1.68
L1:4KB L2:64KB	696,655	390,629	1.78	696,277	391,466	1.78

We obtain the simulation results using a cycle-accurate simulation infrastructure [13] built on top of the CMP-SIM simulator [10]. The bus is not modeled in this simulator. The estimated and observed (simulation) WCRT of DEBIE with cache locking are shown in Table 10. Overestimation ratio is calculated as *Estimation cycles/Observed cycles*. We note that the overestimation is not high, and it varies between 1.10 and 1.27.

Table 11 and Table 12 present the results of PapaBench and synthetic benchmark. The overestimation varies between 1.47 and 2.05.

8 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. Moreover, we exploit our analysis results to lock some memory blocks into the private L1 caches so as to reduce the impact of interference among the memory blocks in the shared L2 cache. Our cache locking algorithm improves the WCRT estimates by up to 80%.

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 Singapore Ministry of Education Academic Research Fund Tier 2, MOE2009-T2-1-033.

References

1. 3rd Generation Intel Xscale Microarchitecture Developers's Manual. Intel, May 2007. `http://www.intel.com/design/intelxscale`.
2. ADSP-BF533 Processor Hardware Reference. Analog Devices, April 2009. `http://www.analog.com/static/imported-files/processor_manuals/bf533_hwr_Rev3.4.pdf`.
3. ARM Cortex A-8 Technical reference Manual. Arm, Revised March 2004. `http://www.arm.com/products/CPUs/families/ARMCortexFamily.html`.
4. ARM1156T2-S Technical reference Manual. Arm, Revised July 2007. `http://www.arm.com/products/CPUs/families/ARM11Family.html`.
5. Message Sequence Charts. ITU-TS Recommendation Z.120, 1996.
6. Task Graphs For Free. http://ziyang.eecs.umich.edu/ dickrp/tgff/.
7. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. *Lecture Notes in Computer Science*, 1145:52–66, 1996.
8. R. Alur and M. Yannakakis. Model checking message sequence charts. In *Proceedings of the International Conference on Concurrency Theory*, 1999.
9. T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.
10. S. Baldawa. CMPSIM: A Flexible Multiprocessor Simulation Environment. Master's thesis, The University of Texas at Dallas, 2007.
11. A. M. Campoy et al. Cache contents selection for statically-locked instruction caches: An algorithm comparison. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005.
12. S. Chattopadhyay and A. Roychoudhury. Static bus schedule aware scratchpad allocation in multiprocessors. In *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, LCTES '11, pages 11–20.
13. S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, 2010.
14. 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.
15. European Space Agency. DEBIE – First standard space debris monitoring instrument, 2008. Available at: http://gate.etamax.de/edid/publicaccess/debie1.php.

16. H. Falk, S. Plazar, and H. Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2007.
17. A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards wcet analysis of multicaore architecures using uppaal. In *Proceedings of 10th International Workshop on Worst-Case Execution-Time Analysis*, WCET '10.
18. D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, 2009.
19. 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.
20. 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.
21. 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.
22. 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.
23. 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/.
24. 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.
25. Y. Liang and T. Mitra. Instruction cache locking using temporal reuse profile. In *DAC '10: Proceedings of the 47th annual Design Automation Conference*.
26. T. Liu, M. Li, and C. J. Xue. Minimizing WCET for real-time embedded systems via static instruction cache locking. In *RTAS '09: Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009.
27. 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.
28. F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2-3), 2000.
29. 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.
30. Fadia Nemer, Hugues Cass, Pascal Sainrat, Jean paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In *In WCET 06*, 2006.
31. I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium*, 2002.
32. P. Puschner and M. Schoeberl. On composable system timing, task timing, and WCET analysis. In *International Workshop on Worst-Case Execution Time Analysis*, 2008.
33. 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.
34. 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.
35. 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.
36. 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.
37. 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.

38. 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.
39. 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.
40. W. Zhang and J. Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '09.