Accounting for Cache-Related Preemption Delay in Dynamic Priority Schedulability Analysis

Lei Ju Samarjit Chakraborty Abhik Roychoudhury
Department of Computer Science, National University of Singapore
E-mail: {julei, samarjit, abhik}@comp.nus.edu.sg

Abstract

Recently there has been considerable interest in incorporating timing effects of microarchitectural features of processors (e.g. caches and pipelines) into the schedulability analysis of tasks running on them. Following this line of work, in this paper we show how to account for the effects of cache-related preemption delay (CRPD) in the standard schedulability tests for dynamic priority schedulers like EDF. Even if the memory space of tasks is disjoint, their memory blocks usually map into a shared cache. As a result, task preemption may introduce additional cache misses which are encountered when the preempted task resumes execution; the delay due to these additional misses is called CRPD. Previous work on accounting for CRPD was restricted to only static priority schedulers and periodic task models. Our work extends these results to dynamic priority schedulers and more general task models (e.g. sporadic, generalized multiframe and recurring real-time). We show that our schedulability tests are useful through extensive experiments using synthetic task sets, as well as through a detailed case study.

1 Introduction

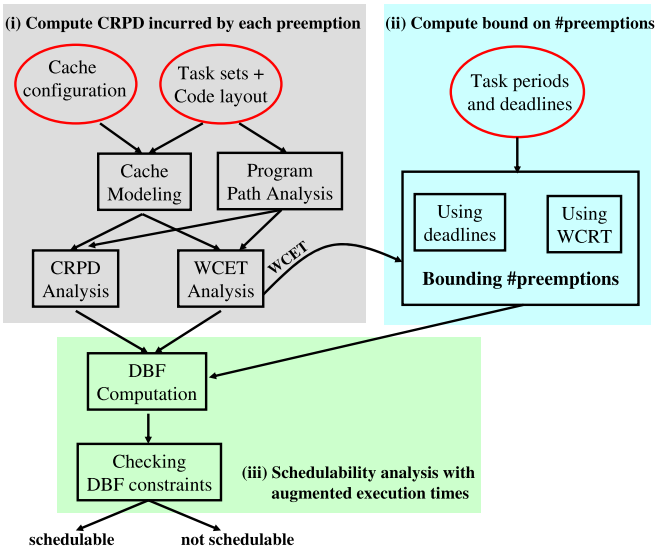
Model-based design is increasingly emerging as the key to tackle the growing complexity of modern real-time and embedded systems. It typically involves choosing an appropriate task model which accurately reflects the characteristics of the underlying application, annotating such a model with parameters such as execution times, deadlines and periods, followed by a schedulability analysis to verify whether all timing constraints are satisfied for all possible runs of the system. For such an analysis to be meaningful, it is important to accurately estimate the execution times of the different tasks constituting the system being designed. This has led to a lot of recent work on what is referred to as the worst-case execution time (WCET) analysis of programs [5, 10, 12, 16], which involves both program path analysis and modeling the timing effects of processor microarchitectural features (e.g. caches and pipelines).

However, such WCET analysis is usually carried out for each task in isolation and there has been relatively less emphasis on estimating the effects of inter-task interferences on the execution times of tasks. For example, depending on the cache blocks shared by two tasks T and T', the preemption of T by T' introduces additional cache misses when T resumes execution, thereby incurring an increase in its execution time. This additional execution time or delay is referred to as the cache-related preemption delay (CRPD) [7, 11]. Here, it may be noted that many timing analysis and system integration tools (e.g. SymTA/S which targets the automotive electronics domain [8, 15]) require the designer to annotate each task with its execution time (as discussed above), but currently do not provide any convenient mechanism to account for the CRPD. However, neglecting the CRPD may lead to unsafe execution time estimates for tasks (and hence incorrect schedulability tests).

Our contribution and relation to previous work: To address this shortcoming, there have been a number of recent attempts to integrate a CRPD estimation scheme within a schedulability analysis framework [6, 13, 14]. But all of these efforts were restricted to strictly periodic task sets and static priority schedulers. In this paper we show how CRPD can be accounted for within a dynamic priority schedulability analysis framework. Further, our proposed technique is not restricted to periodic task sets; it is applicable to more general task models such as sporadic [3], multiframe, generalized multiframe [2], and recurring real-time [1]. Our technique comprises three main steps: (i) using program analysis techniques to estimate the maximum CRPD incurred by *each* preemption of a task, due to possible preempting tasks. (ii) bounding the number of preemptions of each task. (iii) augmenting the execution time of each task with its total CRPD (due to *all* possible preemptions) and using these augmented execution times to perform a schedulability analysis using the processor demand criterion [2, 3].

As one might expect, our tests are safe (sufficient) but not tight (necessary). The pessimism arises from both steps (i) and (ii). However, through a number of experiments using both synthetic task sets and a case study, we show that our tests are useful. In particular, we show that many task sets which were originally schedulable, fail our tests when CRPD is taken into account. At the same time, our tests are not overly pessimistic.

978-3-9810801-2-4/DATE07 © 2007 EDAA



Overview of the proposed CRPD-aware dynamic priority schedulability analysis framework.

A schematic overview of our analysis framework is shown in Figure 1. Given the program code corresponding to each task and its layout in the memory, we use cache modeling and program analysis techniques to estimate the WCET and CRPD in step (i). This is followed by estimating the number of task preemptions in step (ii). Towards this, we propose two possible techniques. The first uses task deadlines and is more practical for reasons we describe later. The second relies on computing the worst case response times (WCRT) of tasks. Although for a restricted class of task sets this might lead to a less pessimistic test, such task sets would rarely arise in real-life applications. Finally, in step (iii) we use the augmented execution times of tasks to compute their *demand bound functions* (DBFs) and use these functions to perform a schedulability test.

The rest of the paper is organized as follows. In the next section we outline our program analysis technique to estimate the maximum CRPD incurred by each task of a task set (i.e. step (i)). In Section 3 we discuss the details of steps (ii) and (iii). Finally, an experimental evaluation of our framework is presented in Section 4, followed by a discussion on possible directions for future work in Section 5.

2 Cache-Related Preemption Delay

In the following, we consider the effects of instruction cache to concretely define and understand the concept of CRPD. As the reader will observe, the same definitions and analysis can be employed (with minor modifications) for data cache as well. Also, we implicitly assume a direct-mapped cache for simplicity of discussion; again this can be extended in a straightforward fashion to set-associative caches. We do not make any assumptions about whether the code memories of different tasks share memory blocks. Thus the memory space of different tasks can be considered disjoint, even though the memory blocks of different tasks get mapped to the same cache block.

Given a preempted task T and a preempting task T', the cache-related preemption delay is an upper bound on the delay due to additional cache misses caused by preemption of T by T'. Consequently we have to consider all possible program points of T where it can get preempted and capture the possible "*cache states*" at these preemption points. Further, in each such cache state we need to find which memory blocks in the cache will be used in subsequent execution of T. We can then consider all possible "cache states" when T' completes, and combine it with the possible "cache states" due to T at preemption, to get the maximum number of cache misses in T (after it resumes) due to the preemption by T'. This number multiplied by the cache miss penalty provides $CRPD(T, T')$, the cache-related preemption delay of T due to preemption by T'.

2.1 Abstract Cache States

Our preceding discussion on CRPD is a high-level one, since we did not discuss what "cache states" are and how they are computed. This indeed is a matter of choice since we can tune the level of abstraction at which we capture the cache states. This decides the precision of the analysis and the tightness of the CRPD estimation.

Assuming direct-mapped cache, it is possible to define a *concrete cache state* as a mapping $\{1, \ldots, n\} \rightarrow M \cup \{\perp\}$; where n is the number of blocks in the cache, M is the set of memory blocks (which get mapped to different cache blocks) and $\perp$ denotes the situation where a certain cache block is empty. Conceptually, an abstract cache state represents *a set of* concrete cache states. However, there can be differences in representation of an abstract cache state leading to different degrees of abstraction. Here we present two possible choices to illustrate this issue. The notation 2^S for a set S denotes the powerset of S.

- For each cache block c, the content of c is not a single memory block (as would be the case for a concrete cache state in a direct-mapped cache) but a set of memory blocks (see [6]). The *type* of such an abstract cache state is $\{1, \ldots, n\} \rightarrow 2^{M \cup \{\perp\}}$.
- The content of the cache is not given by one mapping from cache blocks to memory blocks, but a set of mappings from cache blocks to memory blocks (see [11]). The type of such an abstract cache state is $2^{\{1,\ldots,n\} \rightarrow M \cup \{\perp\}}$.

In the first choice, an abstract cache state can be given by

$$1 \rightarrow \{a, \perp\}, \quad 2 \rightarrow \{b, d\}$$

or $[\{a, \perp\}, \{b, d\}]$ for a direct-mapped cache with two cache blocks into which memory blocks $\{a, b, c, d\}$ get mapped

to. This abstract state represents four concrete cache states $[a, b]$, $[\perp, b]$, $a, d]$, $[\perp, d]$ — corresponding to the two choices in each of the two cache blocks. In the second choice, the abstract cache state is *directly* represented as a set of concrete cache states. We have adopted the second choice in this paper since it leads to more precise program analysis.

2.2 Associating Abstract Cache States with Program Points

Given a definition of abstract cache states, we can traverse the control flow graph of a task T to associate each program point of T with an *Incoming Abstract Cache State*.

Definition 1. *An incoming abstract cache state for a program point p must capture all the concrete cache states with which p can be reached.*[1]

Since the control flow graph contains loops, the computation of the *Incoming Abstract Cache State* will be iterative, where the *Incoming Abstract Cache State* estimate for each program point gets updated in every iteration. This is continued until a (least) fixed-point is reached. Convergence to a fixed point is guaranteed because the set of concrete cache states represented by the *Incoming Abstract Cache State* estimates monotonically increase and the domain of concrete cache states is finite.

Similar to the *Incoming Abstract Cache State* computation, we also compute *Outgoing Abstract Cache State* for each program point of a task T.

Definition 2. *An outgoing abstract cache state for a program point p must capture all concrete cache states at which any cache block can be first referenced after p.*

Again, the *Outgoing Abstract Cache State* for each point is also computed as a (least) fixed-point. The only difference between the two fixed point computations is that while computing the *Incoming Abstract Cache State* of a program point p, we (iteratively) propagate the abstract cache states for p's predecessors in the task's control flow graph. However, for computing the *Outgoing Abstract Cache State* of p, we (iteratively) propagate the abstract cache states of p's successors.

2.3 CRPD Estimation

Using the above notions, we can now compute $CRPD(T, T')$ — the cache related preemption delay due to the preemption of task T by task T' — as follows.

For each program point p in the preempted task T (there are only finitely many such points), we compute (i) the incoming abstract cache state of p, (ii) the outgoing abstract cache state of p, and (iii) a pointwise *intersection* (performed on a per cache block basis) between incoming and outgoing cache states of p. The intersection conservatively estimates the cache blocks which contain such memory blocks at program point p that they are referenced after p. We call such cache blocks as *Useful cache blocks* at program point p of task T, and denote this set as $UCB(p, T)$.

After having computed the *Useful Cache Blocks* at every program point of the preempted task T, we compute the *Incoming Abstract Cache state* at the end of the preempting task T'. This will be done by a fixed-point analysis over the control flow graph of T'. From the incoming abstract cache state of the termination point of T' we can find out the number of cache blocks used by some memory block of T'. Let us call these the *Used Cache blocks* of T' and denote this set as $usedCB(T')$. We can now compute $CRPD(T, T')$ as follows.

$$CRPD(T, T') = max_{p \in Prog(T)} \mid UCB(p, T) \cap usedCB(T') \mid$$

Here, $Prog(T)$ is the set of all program points in task T. Thus we want to capture those cache blocks which could be useful at some program point p of task T, *and* are used by task T' when it preempts T — thereby resulting in addition cache misses when task T resumes execution from program point p.

Indirect preemptions: In the preceding discussion, we sketched a method for estimating $CRPD(T, T')$. But in a system with more than two tasks, T may be preempted by T', which further gets preempted by another task T''. Since all tasks share the same cache, the execution of T'' can also potentially introduce additional cache misses which are encountered when T resumes. To solve this problem in a clean way, we always define the CRPD between a pair of tasks, and conservatively estimate the delay due to indirect preemptions. Thus, the cache-related delay in the execution of T owing to the preemption scenario where T gets preempted by T' and T' gets preempted by T'' is conservatively estimated to be $CRPD(T, T') + CRPD(T, T'')$. Hence, given a task set, it is sufficient to compute the CRPD for all possible (ordered) task pairs only.

3 CRPD-aware Schedulability Analysis

In what follows, for simplicity of exposition we only consider sporadic task sets being preemptively scheduled using the Earliest Deadline First (EDF) scheduler. However, it will not be difficult to see that our framework can be used for more general task models as well.

Each task T in a sporadic task set τ, is characterized by a Worst Case Execution Time e, a deadline d and p, which is the minimum seperation in time between two consecutive releases of T [3]. In order to account for cache-related inter-task interferences, we need to augment e with the CRPD

[1] Depending on the precision of the analysis, it could also represent some concrete cache states with which p is never reached in concrete program executions.

that may be incurred by T due to all preempting tasks. This is given by:

$$\hat{e} = e + \sum_{T' \in pr(T)} CRPD(T, T') \times n(T, T')$$

where, $\hat{e}$ is the augmented execution time of T, $pr(T)$ is the set of all tasks that may preempt T under EDF scheduling policy and $n(T, T')$ is the number of preemptions of T due to T'. In the following subsection, we discuss how to compute $pr(T)$ and $n(T, T')$, before presenting our schedulability test.

3.1 Computing the Set of Preempting Tasks

Whereas computing the set of preempting tasks is straightforward for static priority scheduling disciplines, computing $pr(T)$ is less obvious for EDF.

Theorem 1. *A task T can preempt a task T' under EDF scheduling policy only if T has a smaller deadline than that of T' (i.e. $d < d'$).*

Proof. Suppose T has a deadline d, which is greater than or equal to T''s deadline d'. At any execution point, when T' has been executing for some time t and T becomes ready, the remaining deadline for T (which is d) will always be greater than the remaining deadline for T' (which is $d' - t$). Hence, T can never preempt T'. □

Hence, given a task set τ, any task $T \in \tau$ can only be preempted by tasks belonging to the set $pr(T) = \{T' \mid T' \in \tau \wedge d' < d\}$. It may be noted that with static priority schedulers, if T has a higher priority than T', then every instance of T will preempt the execution of T'. However, with dynamic priority schedulers, in particular EDF, $T \in pr(T')$ only implies that *some* instances of T may preempt T', depending on their remaining deadlines. However, a task that does not belong to $pr(T')$ will never be able to preempt T'.

Using WCRT to bound the number of preemptions: Accurately determining $n(T, T')$ under EDF is not possible without unrolling a concrete schedule. Hence, we use an upper bound on the number of possible preemptions of T to approximate $n(T, T')$. Towards this, we first exploit the observation that the maximum number of preemptions of T due to T' under a static priority assignment can serve as an upper bound on the number of preemptions under EDF. Such a static priority assignment may be obtained using a deadline monotonic scheduler (i.e. a task having a smaller deadline has a higher priority). Let us denote the resulting bound on the number of preemptions of T due to T' as $n_{WCRT}(T, T')$, where

$$n_{WCRT}(T, T') = \lceil \frac{R(T)}{p'} \rceil$$

Here, $R(T)$ is the Worst Case Response Time (WCRT) of T under the above-mentioned static priority assignment and p' is the minimum separation time of T'. $R(T)$ may be obtained using well-known techniques for WCRT computation for periodic and sporadic task models.

Using deadlines to bound the number of preemptions: There are two problems with the above approach: (i) it might lead to overestimation, and (ii) computing the WCRT for more general task models (e.g. generalized multiframe and recurring teal-time) is non-trivial. To avoid these drawbacks, an alternative approach is to bound $n(T, T')$ using the task deadlines.

It is easy to see that under EDF, for T' to preempt T n times, the following inequality must hold: $d' + (n-1)p' < d$. This inequality holds irrespective of whether the task set is feasible or not. Using this inequality, it is possible to obtain the following upper bound on the number of preemptions of T by T':

$$n_{deadline}(T, T') = \lceil \frac{d - d'}{p'} \rceil$$

Although this bound on $n(T, T')$ might also be pessimistic at times, it will often be tighter than $n_{WCRT}(T, T')$. More importantly, this technique is applicable to a much wider variety of task models.

3.2 Putting Everything Together

We are now ready to describe our schedulability test which takes into account the CRPD incurred by tasks. As mentioned in Section 1, we use the **processor demand criterion-based test** [2, 3], where for each task T we compute its *demand bound function* $T.dbf(t)$ and check whether the following set of inequalities hold:

$$\sum_{T \in \tau} T.dbf(t) \leq t, \quad \forall 0 \leq t \leq t(\tau)$$

where $t(\tau)$ is a bound that we will derive shortly. Now, if we use the deadline-based approach to bound $n(T, T')$ then the augmented execution time of any task T is given by:

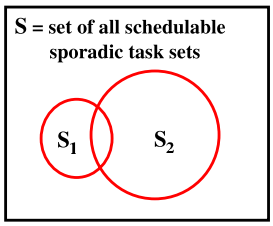
$$\hat{e} = e + \sum_{T' \in pr(T)} CRPD(T, T') \times (\lceil \frac{d - d'}{p'} \rceil)$$

and $T.dbf(t) = \hat{e} \times \max\{0, \lfloor \frac{t-d}{p} \rfloor + 1\}$. Alternatively, the WCRT-based approach may also be used to compute T's augmented execution time. Finally, the bound $t(\tau)$ on the number of tests is given by the following lemma.

Lemma 1. *If a task set τ is not schedulable and $\sum_{T \in \tau} \frac{\hat{e}}{p} \leq 1$, then*

$$\exists \ t \leq (\max_{T \in \tau}\{p - d\}) \times \frac{\sum_{T \in \tau} \frac{\hat{e}}{p}}{1 - \sum_{T \in \tau} \frac{\hat{e}}{p}}$$

for which $\sum_{T \in \tau} T.dbf(t) > t$.



Relationship between the schedulability tests based on $n_{WCRT}(T, T')$ and $n_{deadline}(T, T')$.

Proof sketch. If τ is not schedulable then it follows from the processor demand criterion that there exists some t' for which $t' > \sum_{T \in \tau} T.dbf(t')$. Solving this inequality leads to an upper bound on t'. □

3.3 Discussion

It should be noted that the two bounds we obtained for $n(T, T')$ are both meaningful. However, in the following discussion we aim to show that $n_{deadline}(T, T')$ is more relevant from a practical point of view, apart from it being easy to derive for a wide variety of task models. Figure 2 shows three sets of task sets S, S_1 and S_2, where S is the set of all schedulable sporadic task sets, S_1 is the set of sporadic task sets which pass our schedulability test using WCRT to bound the number of preemptions, and S_2 is the set of sporadic task sets which pass the $n_{deadline}$-based schedulability test.

Task sets in $S - (S_1 \cup S_2)$ are schedulable task sets which fail both our tests. As mentioned in Section 1, this is due to the pessimism introduced by the CRPD estimation and the bound on the number of task preemptions. A more interesting set is $S_1 - S_2$. This set comprises task sets which pass the n_{WCRT}-based test, but fail the $n_{deadline}$-based test. This would happen for task sets with tasks having large deadlines but small execution times (and hence small response times). This results in $n_{deadline}(T, T')$ being overly pessimistic compared to $n_{WCRT}(T, T')$. However, this pessimism alone is not sufficient for such task sets to fail the $n_{deadline}$-based test. This is because such task sets are schedulable under a deadline monotonic scheduler and hence pass our processor demand criterion-based test. For such task sets to fail the $n_{deadline}$-based test, the CRPD of the constituent tasks must contribute to a large fraction of the tasks's execution time (i.e. $\hat{e}$), which for most realistic applications in not true. In summary, $S_1 - S_2$ consists of task sets whose tasks have small execution times, large deadlines, and relatively large CRPD compared to the original WCET.

Finally, $S_2 - S_1$ consists of task sets whose tasks have large (possibly infinite) worst case response times, incur high processor utilization and cannot be scheduled using a static priority scheduler. These task sets would fail the CRPD-aware static priority schedulability tests proposed in [6, 13, 14].

		$PDC_{deadline}$	
		pass	fail
SP_{crpd}	pass	498	0
	fail	279	223

Results for SP_{crpd} versus $PDC_{deadline}$.

		$PDC_{deadline}$	
		pass	fail
PDC	pass	763	42
	fail	0	195

Results for PDC versus $PDC_{deadline}$.

4 Experimental Evaluation

To evaluate the usefulness of our analysis framework, we applied it to both synthetic task sets, as well as to a real-life case study. The results obtained show that (i) many task sets which were originally schedulable, fail our tests when the CRPD is taken into account, and (ii) a number of task sets which failed a CRPD-aware static priority (deadline monotonic) schedulability test, passed our test (meaning that they are schedulable under EDF). These show that accounting for CRPD within a schedulability analysis framework might often be necessary, depending on how critical are the real-time constraints. Secondly, our proposed tests are not overly pessimistic; more specifically, they can distinguish between task sets which are feasible under a dynamic priority scheduler, but infeasible with static priority scheduling.

4.1 Using Synthetic Task Sets

We randomly generated 1000 sporadic task sets with the number of tasks in each set varying between 2 and 6. The execution times of these tasks varied between 1000 and 5000, and the minimum separation time p of each task $T = (e, p, d)$ in τ was set to $\alpha \times |\tau| \times e$, where α was uniformly chosen from the range $[1.0, 2.0]$. Hence, the processor utilization due to τ varied between 0.5 and 1. The deadline d was chosen to lie between e and p and $CRPD(T, T')$ was randomly chosen to be approximately 5% of T's worst case execution time.

We subjected these task sets to three different schedulability tests: (i) SP_{crpd} – which is a CRPD-aware schedulability test for static priority (deadline monotonic) schedulers (as proposed in [6, 13, 14]), (ii) PDC – processor demand criterion-based dynamic priority schedulability analysis, which ignores CRPD, and (iii) $PDC_{deadline}$ – CRPD-aware dynamic priority schedulability analysis which uses task deadlines to bound the number of task preemptions (proposed by us in this paper). The results we obtained are shown in Tables 1 and 2. From Table 1, note that 279 task sets pass the $PDC_{deadline}$ test but fail under SP_{crpd}; these are task sets which are schedulable under EDF but not using a deadline monotonic scheduler, which shows that our proposed test is not overly pessimistic. Table 2 shows the result of accounting for CRPD; 42 task

sets (i.e. 4.2% of the total task sets) which were originally schedulable fail when the effects of CRPD are taken into account, thereby pointing to the importance of CRPD-aware dynamic priority schedulability analysis.

4.2 Case Study: A 3G Phone Application

Our setup is motivated by a 3G mobile phone application which involves audio and video decoding (of incoming streams) as well as encoding (for transmission over a network). For audio, we chose the well-known `adpcm` program; the *mediabench* suite [9] contains source codes for the `adpcm` encoder as well as decoder. For video, we chose one representative task from the MPEG encoder/decoder. In particular, for MPEG encoding (decoding), we selected the `dct` (`idct`) program performing discrete cosine transform (inverse discrete cosine transform). Out of these four programs, we constructed different task sets by varying parameters such as the resolution and frame rate (for video encoding and decoding) and sampling rate (for audio encoding and decoding). For video, we considered resolution choices of 120×90 and 160×120 pixels; the frame rates were varied from $15 - 25$ frames per second. For audio, we considered sampling rates from $25 - 44.1$ KHz. This resulted in as many as 900 different task sets. The execution times of the four tasks remained constant across the different task sets. But their deadlines varied depending on the choices of the frame resolution, frame rate and audio sampling rate. The minimum separation time for all tasks were always equal to their deadlines.

To study the impact of CRPD on schedulability anlysis we considered two possible processor configurations, which were (deliberately) made to differ *only* in their number of cache blocks. We chose cache sizes of 32 blocks in the first processor (call it P_1), and 128 blocks in the second (call it P_2). Both P_1 and P_2 ran at 500 Mhz, and had a direct-mapped cache with cache miss penalty of 20 clock cycles. We used the Chronos WCET analyzer [4] to estimate the WCET of each of the four programs (uninterrupted execution time not considering CRPD) running on P_1 and P_2.

Once again, we subjected the different task sets to the three different schedulability tests listed in Section 4.1. Whereas only 356 task sets passed the SP_{crpd} test, 545 task sets passed our proposed $PDC_{deadline}$ test on the processor P_1. On the processor P_2 these numbers were 392 and 510 respectively. The increase in the number of schedulable task sets can be attributed to the larger number of cache blocks in P_2.

Finally, 728 task sets in P_1 passed the PDC test, compared to only 545 sets passing the $PDC_{deadline}$ test. Again, on P_2 these numbers were 556 and 510 respectively, thereby showing the importance of accounting for CRPD within a schedulability analysis framework.

5 Concluding Remarks

In this paper we presented a dynamic priority schedulability analysis framework that takes into account the CRPD incurred by tasks. This framework extends previous work, which considered only static priority schedulers and periodic task models. Note that we associated the worst case CRPD with each preemption of a task. One possible improvement would be to account for the fact that different preemptions might incur different cache penalties, and factor this into the schedulability test. As a long-term goal, it would be meaningful to model the timing impacts of other microarchitectural features like pipelines and branch prediction within a schedulability analysis framework.

References

[1] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003.

[2] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.

[3] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.

[4] Chronos WCET analyzer, `www.comp.nus.edu.sg/~rpembed/chronos/`.

[5] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3):249–274, 2000.

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

[7] C.-G. Lee *et al.* Bounding cache-related preemption delay for real-time systems. *IEEE Trans. Software Engineering*, 27(9):805–826, 2001.

[8] A. Hamann, M. Jersak, K. Richter, and R. Ernst. Design space exploration and system optimization with SymTA/S–Symbolic Timing Analysis for Systems. In *RTSS*, 2004.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO*, 1997.

[10] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Trans. Design Automation of Electronic Systems*, 4(3):257–279, 1999.

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

[12] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.

[13] J. Staschulat and R. Ernst. Scalable precision cache analysis for preemptive scheduling. In *LCTES*, 2005.

[14] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS*, 2005.

[15] SymTA/S Tool, `http://www.symtavision.com/`.

[16] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.