MIRROR: Symmetric Timing Analysis for Real-Time Tasks on Multicore Platforms with Shared Resources

Wen-Hung Huang, Jian-Jia Chen
Department of Computer Science
TU Dortmund University, Germany
wen-hung.huang@tu-dortmund.de
jia.chen@tu-dortmund.de

Jan Reineke
Department of Computer Science
Saarland University, Germany
reineke@cs.uni-saarland.de

ABSTRACT

The emergence of multicore and manycore platforms poses a big challenge for the design of real-time embedded systems, especially for timing analysis. We observe in this paper that response-time analysis for multicore platforms with shared resources can be symmetrically approached from two perspectives: a *core-centric* and a *shared-resource-centric* perspective. The common "core-centric" perspective is that a task executes on a core until it suspends the execution due to shared resource accesses. The potentially less intuitive "shared-resource-centric" perspective is that a task performs requests on shared resources until suspending itself back to perform computation on its respective core.

Based on the above observation, we provide a pseudo-polynomial-time schedulability test and response-time analysis for constrained-deadline sporadic task systems. In addition, we propose a task partitioning algorithm that achieves a speedup factor of 7, compared to the optimal schedule. This constitutes the first result in this research line with a speedup factor guarantee. The experimental evaluation demonstrates that our approach can yield high acceptance ratios if the tasks have only a few resource access segments.

1. INTRODUCTION

To reduce power consumption and mitigate thermal dissipation issues in computing systems, there has been a trend towards using multicore platforms. Such multicore systems feature a number of shared resources, such as caches, memory banks, and on-chip busses, which have a strong influence on a task's execution time and response time. Traditionally, timing analysis consists of two separate steps: (*i*) worst-case execution time (WCET) analysis, which computes an upper bound on the execution time of a single job of a task running in isolation, and (*ii*) schedulability analysis, which determines whether multiple tasks are guaranteed to meet their deadlines, while sharing a processor. In multicores, however, tasks compete for other resources than just a processing core, and so characterizing their resource consumption by a single value is no longer sensible.

For a more in-depth treatment of literature on the impact of resource sharing on performance and worst-case timing analysis please consult the survey in [1]. One line of work in timing analysis for multicores is to assume structured execution models, e.g., [11, 14, 15, 17]. For example, the superblock execution model, proposed by Pellizzoni et al. [15], classifies the execution of a superblock into three phases: data acquisition, local execution, and data replication phases. This is also standardized in IEC 61131-3. For this structured model, the state-of-the-art results are restricted to the sequential execution of superblocks without any preemptions [11, 15, 17]. Recently, Melani et al. [14] have studied the problem of scheduling tasks consisting of one memory phase and one execution phase, called M/C (memory-computation) tasks, providing an exact response-time analysis under fixed-priority scheduling. Another direction is to consider structured resource arbitration methods, e.g., Time Division Multiple Access (TDMA) or Round-Robin (RR) arbitration [10, 17]. Since the resource sharing problem in multicore systems does not admit tight analytical solutions, approaches adopting timed automata, e.g., [11,13], have been also reported. Altmeyer et. al [2] present a framework to decouple response-time analysis from a reliance on context independent WCET values by separately analyzing the timing contribution of each resource to a task's response time. They implicitly assume that tasks spin while awaiting the response from a shared resource, rather than suspending.

In summary, existing schemes either make very strong assumptions about the tasks and their execution model [11,14, 15,17] or the analysis complexity is intractably high (the state explosion problem) [11, 13]. Further, mathematical guarantees about the quality of the scheduling policy and the schedulability analysis are usually not provided [2,10,15,17], in contrast to classical results for single-core scheduling. In this work, we introduce a new task model that is more permissive than the structured models proposed earlier, while at the same time enabling the derivation of hard mathematical guarantees in the form of a speedup factor.

The analysis presented in this paper is based on two rather simple observations: Observation 1: During its response time, a task is always either executing on its core (i.e., executing or waiting for access) *or* the shared resource (i.e., again either executing or waiting for access). Both contributions to the task's response time can be analyzed separately. Observation 2: The common *core-centric* perspective is that tasks are executed on a core, until they need to access a shared resource, and suspend their usage of the core. The potentially less intuitive *shared-resource-centric* perspective is that tasks perform requests on the shared resource, until suspending to perform computation on their respective cores. Both views yield very similar approaches to bound the time spent on the core and the shared resource, resulting in a *symmetric* analysis.

Our contributions are summarized as follows:

- By adopting the two observations above, we develop schedulability analysis in pseudo-polynomial time and even polynomial time in Section 3.
- We propose in Section 4 our task allocation algorithm.

DAC '16 June 05-09, 2016, Austin, TX, USA
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4236-0/16/06.
DOI: http://dx.doi.org/10.1145/2897937.2898046

In Section 5, we show that this algorithm offers non-trivial quantitative guarantees, including a speedup factor of 7 in polynomial-time complexity.

- The experimental evaluation in Section 6 demonstrates that our approach can yield good acceptance ratios. We also show that our analysis can utilize the structured information in the task models to provide tighter schedulability analysis.

To the best of our knowledge, this is the first result that provides a speedup factor in the presence of shared resources.

2. SYSTEM MODEL

Multicore Architecture Model: We assume in this paper that we have a multicore platform comprised of m identical cores connected by a shared resource, e.g., a communication fabric (bus) for accesses to a shared memory. We assume the following properties:

- Each data request (after it is granted) has a processing time (upper bound) of B.
- Each resource access request is atomic (non-split transaction). However, a segment of resource accesses with several consecutive requests can be preempted at any point at which a request finishes its transfer. This is illustrated in Figure 1 where task τ_1 requesting the shared resource at the beginning of clock cycle 1 experiences blocking of 4 clock cycles from task τ_2 granted at time 0, i.e., B is 5 clock cycles in this example, whereas the second request from τ_2 is preempted by τ_1 at time 5.
- We consider a fixed-priority resource arbiter in this paper: the arbiter of the shared resource always grants the request that is assigned the highest priority.
- We assume that data transfers from the local to the shared memory and vice versa are handled by direct memory access (DMA) units, so that the computation on the core may carry on during such activities. Such a feature is found in PTARM [16].

Task Model: We consider a real-time system to execute a set of n independent, preemptive, *resource access sporadic* (RAS), real-time tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task can release an infinite number of jobs under a given minimum inter-arrival time constraint. An RAS real-time task τ_i is characterized by a 5-tuple $(C_i, A_i, T_i, D_i, \sigma_i)$: T_i denotes the minimum inter-arrival time (also known as period) of τ_i; each job of τ_i has a relative deadline D_i; C_i denotes an upper bound on total execution time of each job of τ_i on a computing core (ignoring the latency due to shared resource accesses); A_i denotes an upper bound on the amount of execution time of shared resource accesses of τ_i; and σ_i denotes the maximum number of resource access segments, each of them may consist of several consecutive *data requests* to access the shared resource. That is, if the platform allows multiple requests to be directly processed without involving the core, a task can have multiple requests to the shared resource within a resource access segment, e.g., fetching (or writing back in the other direction) a collection of data and instructions from the shared main memory to the local scratchpad memory.

That is, if there is no interference on the shared resource accesses, the worst-case execution time of task τ_i is at most $C_i + A_i$. Note that, the derivation of C_i assumes that the shared resource accesses incur no timing cost. Moreover, the derivation of A_i considers only the worst-case shared resource access time of task τ_i. Figure 2 provides an example. Each resource access segment may consist of several requests on the shared resource. For example, given that $B = 1$ time units, there are twenty resource accesses in resource access segment (block) E in Figure 2. Note that A_i is assumed to be w_iB where w_i is an integer. Similar to the models in the literature, [2,11,15], we consider that the multicore platform is timing-compositional [8,18]. Therefore, we can use these parameters to bound the worst-case behaviour safely.

We assume that $C_i + A_i \leq D_i$ for any task $\tau_i \in \tau$. The utilizations of task τ_i on a core and on shared resource are defined as $U_i^C = C_i/T_i$ and $U_i^A = A_i/T_i$, respectively. We further assume that $U_\Sigma^C = \sum_{i=1}^n U_i^C \leq m$ and $U_\Sigma^A = \sum_{i=1}^n U_i^A \leq 1$. Otherwise, it cannot be feasibly scheduled.

Task system τ is said to be an *implicit-deadline* system if $D_i = T_i$ holds for each $\tau_i \in \tau$, and a *constrained-deadline* system if $D_i \leq T_i$ holds for each $\tau_i \in \tau$; otherwise, an *arbitrary-deadline* system. We restrict our attention here to *constrained-deadline* task systems. A system τ is said to be *feasible* if there exists a scheduling algorithm that can schedule the system without any deadlines being missed. A *schedulability test* of a scheduling algorithm is to verify whether the task system is feasible under the given algorithm.

In this paper, we consider *partitioned scheduling*: each task is statically assigned onto one core, and we assume that all the tasks allocated to a core are preemptively scheduled using fixed priority scheduling, for which each task is associated with a unique priority level for scheduling. Since each shared resource request is assumed *non-preemptive*, if task τ_i starts a resource access segment, it has to wait for a lower priority task to finish the access of the shared resource for at most B time units. Therefore, we know that the maximum blocking time of task τ_i due to non-preemptive blocking on the shared resource is at most $\sigma_i B$. We assume that $\sigma_i B \leq A_i$.

Definition of Speedup Factors: Speedup factors are used to quantify the quality of sufficient schedulability tests. A sufficient schedulability test $\mathcal{A}$ is said to have a speedup factor of α if for any task system that is not deemed schedulable by the test, it is the case that the task system is actually not schedulable upon a platform in which each core and the shared resource access is $\frac{1}{\alpha}$ times as fast.

For any RAS task τ_i and any real number $t \geq 0$, the *demand bound function* $dbf_i(t)$ is the largest cumulative execution requirement of all jobs that can be generated by τ_i to have both their arrival times and their deadlines within a contiguous interval of length t [3]. The demand bound function of task τ_i for an interval of length t is $dbf_i^C(t) = max(0, (\left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1) \times C_i)$ and that for the shared resource accesses is $dbf_i^A(t) = max(0, (\left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1) \times A_i)$.

3. SCHEDULABILITY ANALYSIS

We present our response-time and schedulability analyses under the following conditions:

- We assume that the priority levels are assigned *a priori*.
- Each task τ_i is assigned to a core. Let Γ_p denote the *set* of tasks that are allocated on core p, on which task τ_k (that is under analysis) is allocated.
- We *only* test the schedulability of task τ_k assuming that all tasks with higher priority than task τ_k are already guaranteed to meet their deadlines.

The last condition also implies that we have to perform schedulability tests by considering all the tasks one by one. The task set $hp(k)$ consists of the tasks with higher priority than task τ_k. Therefore, the tasks in $hp(k)$ can interfere with task τ_k on the shared resource and the tasks in $hp(k) \cap \Gamma_p$ can interfere with task τ_k on core p. Throughout this section, we assume that we know an upper bound R_i on the worst-case response time of any higher-priority task τ_i in $hp(k)$, which can be derived in the previous iterations. By the last condition above and the assumption of constrained-deadline

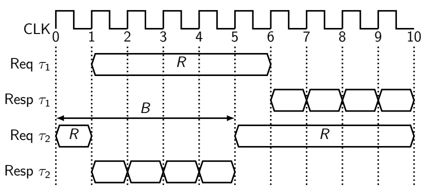

Figure 1: Two tasks access the share resource on a non-split-transaction bus.

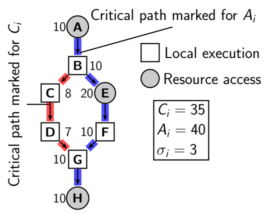

Figure 2: Deriving C_i and A_i.

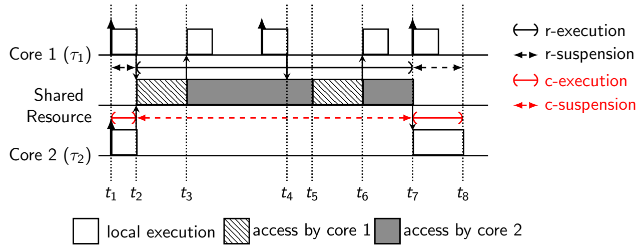

Figure 3: An example of accessing a shared resource by two cores, on which τ_1 and τ_2 are allocated separately.

task systems, we have $R_i \le D_i \le T_i$.

3.1 Our Strategies and Preliminaries

Consider the simplest case in which two tasks τ_1 and τ_2 are assigned on core 1 and core 2, respectively. That is, there is no multitasking on each core. We assume that τ_1 has higher priority than τ_2 and we are now analyzing task τ_2 in all the examples. The schedule of accessing a shared resource by these two tasks is indicated in Figure 3:

- At time t_1, tasks τ_1 and τ_2 start their computation.
- At time t_2, tasks τ_1 and τ_2 both attempt to access the shared resource. The request from task τ_1 is granted, while task τ_2 suspends itself on core 2.
- At time t_3, task τ_1 finishes its access to the shared resource and resumes its local computation. At the same time, task τ_2 starts to access the shared resource.
- At time t_4, task τ_1 again attempts to access the shared resource. Due to the minimum non-preemptive region, this request from task τ_1 is blocked by task τ_2.
- At time t_5, after leaving the non-preemptive region, task τ_2 on the shared resource is preempted by task τ_1.
- At time t_6, task τ_1 finishes its access to the shared resource and resumes its local computation. At the same time, task τ_2 continues to access the shared resource.
- At time t_7, task τ_2 finishes its access to the shared resource and resumes its local computation.
- At time t_8, task τ_2 finishes its execution.

From the point of view of the shared resource, task τ_2 suspends its resource accesses in time intervals $[t_1, t_2)$ and $[t_7, t_8)$ due to local computations, both of which are called *r-suspension* intervals in this paper. Moreover, task τ_2 *executes* on the shared resource in time intervals $[t_3, t_4)$ and $[t_6, t_7)$ and *awaits* (in the queue) to access the shared resource in time intervals $[t_2, t_3)$ and $[t_5, t_6)$, all of which are called *r-execution* intervals in this paper. Symmetrically, from the point of view of core 2, task τ_2 executes/suspends its computation on the core, called *c-execution*/*c-suspension*, in time intervals $[t_1, t_2)$ and $[t_7, t_8)$/ time intervals $[t_2, t_7)$. In either perspective, the feasibility of τ_2 can be examined by answering whether the summation of r-execution time and r-suspension time (or c-execution time and c-suspension time) of a job of task τ_2 is never more than its relative deadline.

Our analysis strategy to safely bound R_k of task τ_k is as follows: Suppose that task τ_k releases a job at time t_0. We check whether this job of task τ_k is guaranteed to be finished by $t_0 + t$. Therefore, we have to determine the cumulative amount of time in the time interval $(t_0, t_0 + t]$ for r-execution and r-suspension while executing this job of task τ_k. They are defined as *r-execution time* and *r-suspension time* of task τ_k in an interval of length t, respectively.

Without multitasking, the total time of resource access suspension from the bus (r-suspension) is trivially upper bounded by the execution time due to local computation. However, with multitasking, this time increases due to interference from other tasks allocated on the same core p. To calculate an upper bound on the r-execution and r-suspension time of task τ_k, we first provide two lemmas:

LEMMA 1. *The cumulative resource access time of task $\tau_i \in hp(k)$ on the shared resource within an interval of length t is upper bounded by:* $E_i(t) = \left\lceil \frac{t + R_i - A_i}{T_i} \right\rceil A_i$

PROOF. In the presence of suspension, we need to consider the *carry-in* effect, as pointed out in [9, 12]. Nevertheless, the interference due to such an effect can be quantified as *jitter*, and thereafter the access times can be upper bounded by releasing all the accesses from the first job (that arrives before t_0 and has an absolute deadline after t_0) as late as possible and the following accesses as soon as possible. □

LEMMA 2. *The cumulative execution time of task $\tau_i \in hp(k)$ on the core, on which τ_i is allocated, within an interval of length t is upper bounded by:* $W_i(t) = \left\lceil \frac{t + R_i - C_i}{T_i} \right\rceil C_i$

PROOF. This is due to the same argument in Lemma 1. □

The rest of this section is organized as follows: (1) We first derive a bound on the r-execution time (denoted as $X_k(t)$) and r-suspension time (denoted as $S_k(t)$) in Section 3.2 under the assumption that the worst-case response time of task τ_k is no more than t for some $0 < t \le T_k$. (2) We then derive a bound on the r-execution time (denoted as X_k^*) and r-suspension time (denoted as S_k^*) that is *independent* of the task's response time in Section 3.3 by referring to the number of resource access segments. (3) Section 3.4 provides the schedulability test, combining the results from (1) and (2).

3.2 Calculating $S_k(t)$ and $X_k(t)$

To evaluate the maximum r-suspension time, provided that the interval length of interest is $t \le T_k$, we can simply sum over the computation workload $W_i(t)$ from all the higher-priority tasks that are allocated on core p plus C_k. The proofs are relatively straightforward.

LEMMA 3. *The r-suspension time of task τ_k due to its executions on core p in an interval of length t is at most*

$$S_k(t) = C_k + \sum_{\tau_i \in hp(k) \cap \Gamma_p} W_i(t) \tag{1}$$

Similarly, we have the corresponding lemma for the maximum r-execution time as follows.

LEMMA 4. *The r-execution time for task τ_k in an interval of length t is at most*

$$X_k(t) = A_k + \sigma_k B + \sum_{\tau_i \in hp(k)} E_i(t) \tag{2}$$

3.3 Calculating X_k^* and S_k^*

We now show how to bound the r-execution time and the r-suspension time, *independently* of the task's overall response time. To this end, we use σ_k to calculate X_k^* and S_k^*.

LEMMA 5. *X_k^* for task τ_k is the smallest t satisfying the following inequality:*

$$A_k+\sigma_k B+\sum_{\tau_i\in hp(k)}\left(\sigma_k-1+\left\lceil\frac{t+\sigma_k\cdot(R_i-A_i)}{T_i}\right\rceil\right)A_i\le t \quad (3)$$

PROOF. We consider any arbitrary program path with $z\le\sigma_k$ shared resource access segments. W.l.o.g., we only have to consider z as an integer and ≥ 1. Let A_k^j denote the amount of execution times on the jth resource access segment from τ_k in the program path. By definition, $\sum_{j=1}^{z}A_k^j\le A_k$. Let r_j be the smallest t satisfying $A_k^j+B+\sum_{\tau_i\in hp(k)}E_i(t)\le t$. In other words, we know that the condition $\forall 0\le t<r_j, A_k^j+B+\sum_{\tau_i\in hp(k)}E_i(t)>t$ holds for all $j=1,\ldots,z$. By adopting Lemma 1, the worst-case response of A_k^j can be easily proved to be upper bounded by r_j for all $j=1,\ldots,z$.

Thus, the maximum r-execution time of this program path for task τ_k is upper bounded by $r_1+r_2+\cdots+r_z$. For a specified $0\le\theta<r_1+r_2+\cdots+r_z$, there always exists a combination of $t_1,t_2,\ldots,t_z$ with $0\le t_j<r_j$ such that $\theta=t_1+t_2+\cdots+t_z$. Therefore, we know that

$$(\sum_{j=1}^{z}A_k^j)+zB+\sum_{\tau_i\in hp(k)}\sum_{j=1}^{z}E_i(t_j)>\sum_{j=1}^{z}t_j=\theta. \quad (4)$$

By the fact that $\lceil x+y\rceil+1\ge\lceil x\rceil+\lceil y\rceil$, we have $\sum_{j=1}^{z}E_i(t_j)=\sum_{j=1}^{z}(\left\lceil\frac{t_j+R_i-A_i}{T_i}\right\rceil)A_i\le(z-1+\left\lceil\frac{z\cdot(R_i-A_i)+\sum_{j=1}^{z}t_j}{T_i}\right\rceil)A_i=(z-1+\left\lceil\frac{z\cdot(R_i-A_i)+\theta}{T_i}\right\rceil)A_i$. The left hand side in Eq. (4) is at most $A_k+\sigma_kB+\sum_{\tau_i\in hp(k)}\left(\sigma_k-1+\left\lceil\frac{\theta+\sigma_k\cdot(R_i-A_i)}{T_i}\right\rceil\right)A_i$. That is, the smallest t solved by Eq. (3) is at least $r_1+r_2+\cdots+r_z$. □

Similarly, we have the following lemma.

LEMMA 6. *S_k^{r*} for task τ_k is the smallest t satisfying the following inequality:*

$$C_k+\sum_{\tau_i\in hp(k)\cap\Gamma_p}\left(\sigma_k+\left\lceil\frac{t+(\sigma_k+1)\cdot(R_i-C_i)}{T_i}\right\rceil\right)C_i\le t \quad (5)$$

PROOF. Since the number of shared resource access segments of τ_k is at most σ_k, the maximum number of execution segments of τ_k on core p will be potentially σ_k+1, interleaved the τ_k resource access segments. The rest of the proof for Eq. (5) is similar to Lemma 5. □

Note that it is also possible that resource accesses start both at the beginning and at the end of a program. Hence, there are at most σ_k-1 execution segments of τ_k in this case, which in turn reduces pessimism, to be discussed in Section 6.

3.4 Response-Time Analysis

The following theorem concludes the schedulability test and response-time analysis directly from Lemmas 3 to 6.

THEOREM 1. *The smallest t satisfying:*

$$\min\{X_k(t),X_k^*\}+\min\{S_k(t),S_k^*\}\le t \quad (6)$$

is a safe upper bound of R_k if t is no more than T_k.

PROOF. We prove this by contraposition. Suppose that task τ_k releases a job at time t_0. (Since we assume constrained-deadline task systems, we can safely consider that there is no job of task τ_k in the ready queue at time t_0 if $R_k\le D_k\le T_k$.) If this job has not yet finished by time t_0+t, then, at any point in the time interval $(t_0,t_0+t]$, task τ_k either executes (or is queued) on the shared resource or suspends on the shared resource. Therefore, if the job of task τ_k is not finished at time t_0+t, a necessary condition is that its r-execution time plus its r-suspension time from t_0 to time t_0+t has been strictly larger than t. Such a condition implies $\min\{X_k(t),X_k^*\}+\min\{S_k(t),S_k^*\}>t$, which contradicts the assumption that t satisfies Eq. (6). □

COROLLARY 1. *An RAS task τ_k allocated on core p is schedulable in fixed-priority partitioned scheduling if there exists $0\le t\le D_k$ s.t. Eq. (6) holds.*

The response-time analysis in Theorem 1 and the schedulability test in Corollary 1 require pseudo-polynomial time complexity. Note that we can also simply test $t=D_k$ requiring only polynomial time complexity at the expense of potentially less precise results.

Response-Time Analysis: Spinning versus Suspension. One might assume that Lemmas 3 and 4 also apply to a scenario in which tasks spin rather than suspending upon resource accesses. Unfortunately, this is not true, as it may underestimate delays due to lower-priority blocking: In a spinning scenario the response time of task τ_k is also affected by lower-priority blocking of higher-priority task that have preempted τ_k. To safely account for such delays, one would have to adapt Lemma 1 to account for lower-priority blocking of each resource access segment of task τ_i. A response-time analysis obtained in this manner roughly corresponds to the analysis described by Altmeyer et al. [2] adapted to the setting explored in this paper. For the special case that B is 0, employing Lemmas 3 and 4 without adaptation is correct for a model in which tasks spin upon resource accesses:

PROPOSITION 1. *For $B=0$, the smallest t satisfying:*

$$X_k(t)+S_k(t)\le t \quad (7)$$

is a safe upper bound of R_k if t is no more than T_k, even if tasks spin upon resource accesses rather than suspending.

4. TASK ALLOCATION

We now present a task allocation algorithm that is compatible with the schedulability test in Corollary 1. Our strategy is to first sort the tasks according to their relative deadlines such that $D_1\le D_2\le\cdots\le D_n$. Then, we allocate the tasks by using a simple heuristic reasonable allocation (RA) algorithm [7], i.e., either First-Fit (FF), Best-Fit (BF), or Worst-Fit (WF). The First-Fit algorithm places the task in the first core that can accommodate the task. If no core is found, it opens a new core and places the task in the new core. The Best-Fit (Worst-Fit, respectively) algorithm places each task in the core with the *smallest* (*largest*, respectively) *remaining capacity* among all the core with sufficient capacity to accommodate the item. The *remaining capacity* is defined as the relative deadline minus the upper bound on the worst-case response time by Theorem 1. Algorithm 1 presents the pseudocode for the first-fit packing. Due to Corollary 1, if Algorithm 1 returns a feasible allocation, this results in a feasible schedule under the deadline-monotonic strategy (a task with a shorter relative deadline has higher priority).

5. SPEEDUP FACTOR

In this section, we derive a speedup factor for Algorithm 1.

LEMMA 7. *Any constrained-deadline RAS task system τ that is feasible upon a multicore platform comprised of m cores and a shared resource must satisfy*

$$\max\left\{\max_{t>0}\frac{\sum_{\tau_i\in\tau}dbf_i^C(t)}{mt},\max_{t>0}\frac{\sum_{\tau_i\in\tau}dbf_i^A(t)}{t},\max_{\tau_i\in\tau}\frac{C_i+A_i}{D_i}\right\}\le 1 \quad (8)$$

Algorithm 1: MIRROR First-Fit Deadline-Monotonic

```
input  : A set $\tau$ of RAS tasks and $m$ identical cores
output: Task allocations $\Gamma_j$ and the feasibility of system $\tau$
sort the given $n$ tasks in $\tau$ s.t. $D_1 \leq D_2 \leq \cdots \leq D_n$;
$\Gamma_j \leftarrow \emptyset, \forall j = 1, 2, \ldots, m$;
for $k = 1, 2, \ldots, n$ do
    for $j = 1, 2, \ldots, m$ do
        if task $\tau_k$ is schedulable according to Corollary 1 then
            $\Gamma_j \leftarrow \Gamma_j \cup \{\tau_k\}$; // assign $\tau_k$ to core $j$
            break (continue the outer loop) ;
    return "infeasible allocation";
return "feasible allocation";
```

Proof. These conditions come from the definition of the demand bound functions, as also used in traditional multiprocessor scheduling [5] (without considering resource sharing). Note that all the demand for accesses to the shared resource, i.e., $dbf_i^A(t)$, must be sequentially executed on a single computing unit, i.e., shared bus. □

Theorem 2. *Algorithm 1 has a speedup factor of* 7.

Proof. Suppose that Algorithm 1 fails to obtain a partition for τ: there exists task τ_k which cannot be mapped to any core. Note that due to the sorting of the tasks in Algorithm 1, all the tasks before task τ_k mapped onto cores have been ensured $R_i \leq D_i$ for $i = 1, 2, \ldots, k-1$. Since τ_k fails the test of Corollary 1 on each of the m cores, for each core $p = 1, 2, \ldots, m$, we have $S_k(D_k) + X_k(D_k) > D_k$. By Lemma 3 and Lemma 4, we have

$$C_k + A_k + \sigma_k B + \sum_{\tau_i \in hp(k)} E_i(D_k) + \sum_{\tau_i \in hp(k) \cap \Gamma_p} W_i(D_k) > D_k$$

Summing over all m such cores, we obtain

$$m\left(C_k + A_k + \sigma_k B + \sum_{\tau_i \in hp(k)} E_i(D_k)\right) + \sum_{\tau_i \in hp(k)} W_i(D_k) > mD_k \tag{9}$$

By definition, $R_i \leq D_i \leq D_k$ for $\tau_i \in hp(k)$. To conclude the speedup factor, we need to prove $E_i(D_k) \leq 3dbf_i^A(D_k)$. If $T_i > D_k$, then $E_i(D_k) \leq (\left\lceil \frac{D_k}{T_i} \right\rceil + 1)A_i \leq 2A_i = 2dbf_i^A(D_i) \leq 2dbf_i^A(D_k)$. Otherwise, if $T_i \leq D_k$, then $E_i(D_k) \leq (\left\lceil \frac{D_k}{T_i} \right\rceil + 1)A_i \leq (\left\lfloor \frac{D_k}{T_i} \right\rfloor + 2)A_i \leq 3\left\lfloor \frac{D_k}{T_i} \right\rfloor A_i \leq 3dbf_i^A(D_k)$. Similarly, $W_i(D_k) \leq 3dbf_i^C(D_k)$. Dividing both sides by mD_k in Eq. (9), together with our assumption $\sigma_k B \leq A_k$ and the facts $E_i(D_k) \leq 3dfb_i^A(D_k)$ and $W_i(D_k) \leq 3dbf_i^C(D_k)$, we have

$$\frac{C_k}{D_k} + \frac{2A_k + 3\sum_{\tau_i \in hp(k)} dbf_i^A(D_k)}{D_k} + \frac{3\sum_{\tau_i \in hp(k)} dbf_i^C(D_k)}{mD_k} > 1. \tag{10}$$

Recall that $\sum_{\tau_i \in \tau} dbf_i^A(D_k) \geq A_k + \sum_{\tau_i \in hp(k)} dbf_i^A(D_k)$. Therefore, we have

$$\frac{C_k}{D_k} + \frac{3\sum_{\tau_i \in \tau} dbf_i^C(D_k)}{mD_k} + \frac{3\sum_{\tau_i \in \tau} dbf_i^A(D_k)}{D_k} > 1. \tag{11}$$

Assume for a contradiction that the task set is feasible on a multicore with speed $\frac{1}{7}$. Then, considering the adjusted speed, we know by Lemma 7 that $\frac{C_k}{D_k} \leq \frac{1}{7}$, $\frac{\sum_{\tau_i \in \tau} dbf_i^C(D_k)}{mD_k} \leq \frac{1}{7}$, and $\frac{\sum_{\tau_i \in \tau} dbf_i^A(D_k)}{D_k} \leq \frac{1}{7}$. This, however, clearly contradicts Eq. (11), implying a speedup factor of 7. □

We would like emphasize that the factor 7 in Theorem 2 can already be obtained in polynomial time if we only test t at D_k in Corollary 1. Moreover, for implicit-deadline, harmonic task systems, i.e., $T_i = D_i$ and $\frac{D_k}{T_i}$ is an integer for $\tau_i \in hp(k)$, the speedup factor is 5, since $E_i(D_k) \leq 2dbf_i^A(D_k)$ and $W_i(D_k) \leq 2dbf_i^C(D_k)$ in such cases.

6. EXPERIMENTAL RESULTS

In this section, we conduct extensive experiments using synthesized task sets. Due to space limitations, only a subset of the results is presented. We evaluate these tests on a 4-core system, i.e., $m = 4$. We generate 100 task sets for each utilization level, from $0.01m$ to $0.99m$, in steps of $0.01m$. The metric to compare the results is to measure the *acceptance ratio*. The acceptance ratio of a level is said to be the number of task sets that are deemed schedulable by the test divided by the number of task sets for this level, i.e., 100.

The cardinality of the task set was 5 times the number of cores, i.e., 20. The UUniFast-Discard method [4] was adopted to generate a set of utilization values with the given goal. We here used the approach suggested by Davis et al. [6] to generate the task periods according to the exponential distribution. The distribution of periods is within two orders of magnitude, i.e., 10ms-1000ms. The execution time was set accordingly, i.e., $C_i = T_i U_i$. Task relative deadlines were implicit, i.e., $D_i = T_i$. We then generated a set of access utilization values U_i^A for tasks with the given U_Σ^A, according to the uniform distribution, generated by the UUniFast method. We consider task sets with total access utilization U_Σ^A of 40% and 70%. We ensure that for every task τ_i, $U_i^A + U_i^C \leq 1$. Then, the upper bound on the access time to shared resources A_i was set accordingly, i.e., $A_i = T_i U_i^A$.

The maximum number of resource access segments σ_i was set depending on the following types of access: 1 (rare access, type=R), 2 (moderate access, type=M), and 10 (frequent access, type=F). The evaluated tests are listed as follows:

- MIRROR: the test proposed in Corollary 1.
- MIRROR-SPIN: the test following Proposition 1 using only $S_k(t)$ and $X_k(t)$, which applies also to tasks that spin while awaiting access to the shared resource. This resembles the test from [2] in our model when B is 0.
- exact-MC: the exact test proposed in [14] for M/C tasks, which is the special case of the RAS task model.

To fairly compare the results with respect to [2, 14], we assume that $B = 0$ in all the tests; otherwise, our tests can benefit from the short blocking time, and the other results can be significantly impacted by their considerations of blocking time as explained in Section 3.4.

Result I. Figures 4a and 4b depict the results (by using the FF allocation) with frequent access (F) for 40% and 70% bus utilizations. As shown in Figures 4a and 4b, the performance by MIRROR-SPIN and MIRROR are identical in the case of frequent accesses. The reason behind this is that Lemma 5 and 6, by which MIRROR are advantageous to MIRROR-SPIN, become ineffective when the number of resource access segments is large.

Result II. In Figures 4c and 4d we consider performance differences between different reasonable allocations: FF, BF, and WF, denoted by MIRROR-FF, MIRROR-BF, and MIRROR-WF, respectively. The number of resource access segments is 2 (type=M), which is aligned with the superblock model used in the literature [11, 15] and the IEC61131-3 standard. This models the execution of a job into three phases, i.e., read, execution, and write. Therefore, we only consider one computation segment in Lemma 6. The performance of FF is identical to that of BF, and noticeably

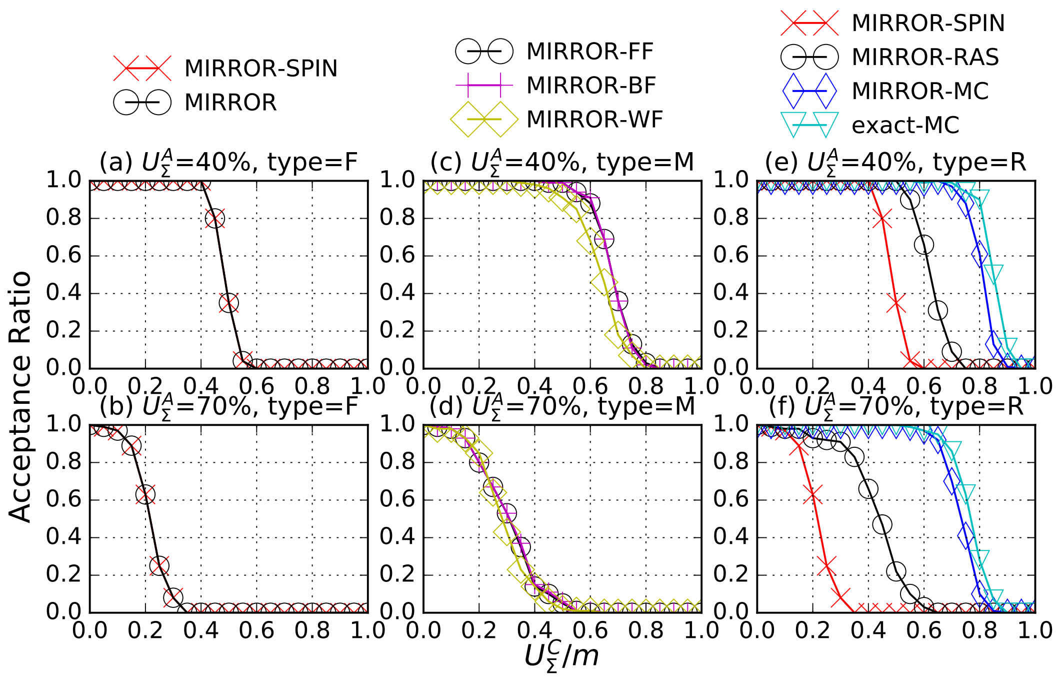

Figure 4: Comparison with different access utilizations U_Σ^A and different types of access frequency (type). In the first (and second, respectively) row the access utilization is 40% (and 70%, respectively). In the first (respectively, second and third) column, frequent (respectively, moderate and rare) access frequency is assumed.

better than that of WF. In such a case, the effectiveness of MIRROR is sustainable for the case of 40% bus utilization, smoothly dropping down at around 50% core utilization.

Result III. In Figures 4e and 4f, we show the effectiveness by our proposed MIRROR (by using the FF allocation) for M/C tasks, in which each task has only one computation segment and one resource access segment [14]. We denote the proposed approach for such tasks as MIRROR-MC, distinct from MIRROR-RAS for RAS tasks which needs two segments in Lemma 6. Note that in the presence of the M/C task model, we do not have to consider the arrival jitter on the interference due to memory accesses, quantified in Lemma 1. We first notice that without knowing the program structure, both MIRROR-SPIN and MIRROR-RAS perform relatively poorly, even though MIRROR still outperforms MIRROR-SPIN. In the presence of M/C tasks, the proposed MIRROR achieves high schedulability, compared to exact-MC.

More importantly, the number of resource access segments adversely affects schedulability, for instance, as reported in Figures 4a, 4c, and 4e.

7. CONCLUSIONS

This paper presents a symmetric approach to analyze the schedulability of a set of resource access sporadic real-time tasks. Our key observation is to consider *suspension-aware* response-time analysis in both core-centric and shared-resource-centric perspectives. Our schedulability analysis can also be seamlessly adopted for shared resource models, e.g., Time Division Multiple Access (TDMA), First-Come First-Serve (FCFS), and Round-Robin (RR) protocols. Together with reasonable task allocations, we show that the speedup factor of our approach using fixed-priority arbitration, compared to the optimal schedule, is at most 7. This is the first result that provides a speedup factor for the scheduling problem in the presence of shared resources. In future work, we would like to better understand the interplay between the execution segments and the resource access segments, to provide tighter analysis.

Acknowledgments: This paper is supported by the DFG, as part of the SFB 876 (http://sfb876.tu-dortmund.de/).

References

[1] A. Abel, F. Benz, et al. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR 2013 - Concurrency Theory*, pages 25–43. 2013.

[2] S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke. A generic and compositional framework for multicore response time analysis. In *RTNS*, 2015.

[3] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[4] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[5] J.-J. Chen and S. Chakraborty. Resource augmentation bounds for approximate demand bound functions. In *RTSS*, 2011.

[6] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *Computers, IEEE Transactions on*, 57(9):1261–1276, 2008.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979.

[8] S. Hahn et al. Towards compositionality in execution time analysis – definition and challenges. In *CRTS*, 2013.

[9] W.-H. Hung, J.-J. Chen, H. Zhou, and C. Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *DAC*, 2015.

[10] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 50(2):185–229, 2014.

[11] K. Lampka et al. A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets. *Real-Time Systems*, 50(5-6):736–773, Nov. 2014.

[12] C. Liu and J. Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *RTSS*, 2014.

[13] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS*, pages 339–349, 2010.

[14] A. Melani et al. Memory-processor co-scheduling in fixed priority systems. In *RTNS*, 2015.

[15] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE*, pages 741–746, 2010.

[16] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS*, 2011.

[17] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *DAC*, 2010.

[18] R. Wilhelm et al. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):966–978, 2009.