Global Real-Time Memory-Centric Scheduling for Multicore Systems

Gang Yao[‡], Rodolfo Pellizzoni[⋆], Stanley Bak[§¶], Heechul Yun[†] and Marco Caccamo[‡]
[‡] University of Illinois at Urbana-Champaign, USA, {gangyao, mcaccamo}@illinois.edu
[†] University of Kansas, USA. heechul.yun@ku.edu
[⋆] University of Waterloo, Canada, rpellizz@uwaterloo.ca
[§] United States Air Force Reasearch Lab, Information Directorate, Rome, NY, USA, stanley.bak.1@us.af.mil

***Abstract*—As the number of cores increases, more master components can simultaneously access main memory. In real-time systems, this ongoing trend is leading to crippling pessimism when computing the worst-case cache miss time, since a memory request could potentially contend with other requests coming from every other core in the system. CPU-centric scheduling policies, therefore, are no longer sufficient to guarantee schedulability without introducing unacceptable pessimism for memory-intensive task sets. For this reason, we believe a shift is needed towards real-time scheduling approaches that can prevent timing interference from memory contention, while still making efficient use of the multicore platform.**

Previously, we have demonstrated the practicality of the *PREM* task model, where each job consists of a sequence of phases, some of which access memory and some of which perform only computation on cached data. In this work, we present the first global memory-centric scheduling policy for memory-intensive task sets whose jobs can be modeled as a sequence of memory-intensive (memory phase) and execution-intensive (execution phase) phases. The proposed policy is parameterizable based on the number of cores which are allowed to concurrently access main memory without saturating it. Building upon results from multicore response-time analysis, we introduce the notion of *virtual memory cores* as a fundamental technique for performing phase-based response time analysis for memory-intensive task sets. Finally, we use synthetic task set generation to demonstrate that proposed scheduling policy and related schedulability bound do indeed better schedule memory-intensive task sets when compared to state-of-art multicore scheduling.

I. Introduction

Multicore platforms are becoming increasingly popular in modern computing systems since they have a high processing capacity at a comparatively low cost. Shared resources on the multicore chip, such as main memory, are increasingly becoming a point of contention. For example, processing high resolution images on one core, while tracking objects from real-time vision data on another core, may cause a mutual slowdown due to interference for access to the shared memory, which was not a bottleneck when these tasks were run on a single-core system.

While the real-time scheduling problem has been studied for several decades, it has traditionally focused on scheduling CPU computation. One fundamental assumption is that we can estimate Worst-Case Execution Time (WCET) for each task when running alone in the system. However, when considering memory-intensive applications running concurrently on a multicore chip, the measured worst-case execution times can vary significantly, and may change depending on what is running on the other cores on the system. In the worst case, all the cores may simultaneously compete to access the main memory, and the worst-case task execution time can grow linearly with the number of cores in the system [23].

Classic multicore scheduling policies, therefore, are no longer sufficient to guarantee schedulability without introducing unacceptable pessimism for memory-intensive task sets. In this work, we aim at making the real-time scheduling policy aware of memory requirements of the running task set. By running a real-time system below its memory bandwidth saturation, tasks' WCETs are less sensitive to run-time memory access patterns and the pessimism of schedulability analysis is mitigated achieving higher schedulable utilization. We propose a novel memory-centric scheduling policy, in which memory phases are scheduled differently from execution phases: to do this, we require additional task information on when each task can request memory access. We leverage our earlier results on the PRedictable Execution Model (PREM) [22], where each task explicitly indicates regions (memory phases) of its execution where memory will be accessed, and regions (execution phases) of execution where computation will be done on cached-local data. Based on this information, the real-time scheduler promotes access to memory from different cores in order to produce a predictable and efficient schedule.

Earlier, we have presented a memory-centric multicore scheduling policy where memory access is granted based on a coarse-grained Time Division Multiple Access (TDMA) schedule [30]. This TDMA arbitration provides core isolation for the shared memory resource, and each core can be considered as if it were given a dedicated memory resource during its time slot. In another work [8], we used simulations to evaluate a large set of partitioned multicore scheduling algorithms which allowed any core to access memory at any time. In this work, we present the first global memory-centric scheduling policy for memory-intensive task sets whose jobs can be modeled as a sequence of memory and execution phases. The core idea of our approach is *memory promotion*, that is,

[¶]The majority of the work was performed while the author was at the University of Illinois at Urbana-Champaign. DISTRIBUTION A. Approved for public release; Distribution unlimited. (Approval AFRL PA case number 88ABW-2015-4173, 02 SEP 2015) raising the priority of memory access phases at run time. The advantage of memory promotion is that memory accesses do not suffer interference from the local computation (execution phase) of any other tasks. Hence, from an analysis perspective, memory phases and execution phases can be isolated without interfering with each other. Compared to the prior work which achieves isolation using TDMA, this work provides isolation by modeling the system as composed by two different types of resources (*virtual execution cores* and *virtual memory cores*) and isolation is enforced by controlling the tasks' priorities based on what type of phase is currently running. Along with core isolation, we also provide a *parametric response time analysis* which is parameterized based on the system's memory bandwidth. We demonstrate the benefits of memory promotion by generating synthetic task sets and using the proposed response time analysis to evaluate schedulability.

This paper is organized as follows. First, Section II explains the system and task model, validates the assumptions we make, and reviews essential results from response time analysis for globally-scheduled multicore systems. Section III then introduces the rules for our proposed global memory-centric scheduling policy. Building upon the results from multicore response-time analysis, Section IV uses the notion of *virtual memory cores* as a fundamental technique to perform phase-based response time analysis, and a worst-case response time is established for the proposed scheduling policy. Section V shows the benefits of memory promotion using our response-time bound in a simulation-based analysis. Finally, Section VI reviews prior research related to real-time multicore scheduling, and the paper concludes in Section VII.

II. System Model and Background

We now describe the task model used for our schedulability results and subsequent evaluation (Section II-A). This task model is justified as applicable to scheduling real-time multi-core systems built using currently-available hardware (Section II-B). Then, we review key response-time analysis results for multicore real-time systems, which will be built upon in our later analysis (Section II-C).

A. Task Model

In this work, we use a task model where each job is broken down into two types of phases. The first type of phase is a *memory phase*, where the CPU can perform main memory reads and writes. The second type of phase is a *execution phase*, where the CPU performs computation on already-cached data and does not access main memory. The practicality of this task model will be justified in the next subsection.

Tasks perform standard, non-DMA memory access; when a task is accessing memory, the core is occupied by this task and cannot do other computation. Furthermore, each individual task must be executed serially and cannot be run in parallel on multiple cores.

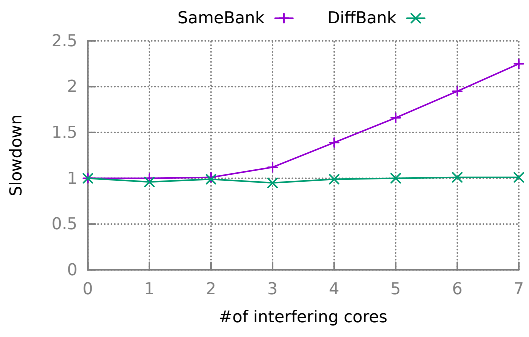
In this paper, we consider global CPU scheduling. The global scheduler maintains a pool of tasks which are ready to be executed, which can be run on any core. The system consists of a multi-core chip with m_{core} identical cores. Memory is a global shared resource, and can be accessed by any of the cores. However, if all the cores were allowed to access memory at same time, the interference would cause a slowdown for all tasks accessing memory [23]. Avoiding this interference and its effects on schedulability is the main justification for the proposed scheduling algorithm in this paper. In order to prevent memory interference, we limit the number of cores which can concurrently access memory without timing interference to m_{memo}. As long as m_{memo} or fewer cores access memory at the same time, memory bandwidth is not saturated and timing interference due to memory contention is negligible.

We use a fully preemptive scheduler, such that a task can be preempted at any time and resume execution afterwards. A mechanism needs to be in place which can guarantee that memory prefetched during a memory phase is not evicted from the last level cache, either by the same task (self-eviction), other tasks running on the same core (intracore eviction), or tasks on other cores which access the shared cache (intercore eviction). Existing mechanisms for achieving this are described in the next subsection. Additionally, cached memory needs to be accessible from any core (to permit migration), which implies a shared last-level cache. Many current-generation multicore chips have a shared last-level cache.

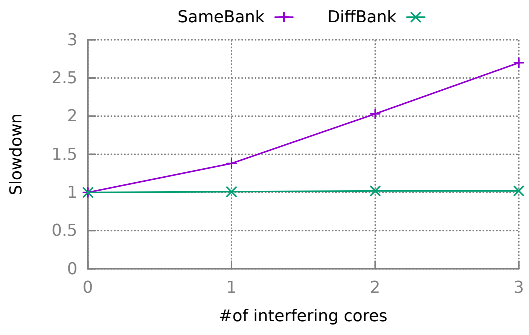
Formally, our focus in this paper is the scheduling policy and the corresponding response time analysis for a set of n periodic tasks $\mathcal{T} = \tau_1, \ldots, \tau_n$, each of which is modeled as $(T_i,\ C_i)$ in accordance with the classical Liu and Layland model [18]. The two tuple $(T_i,\ C_i)$ represents the period and worst-case execution time of each task respectively, and each task has its relative deadline D_i equal to its period. We consider a fixed priority global scheduling approach, where each task is assigned a fixed priority according to Rate Monotonic [18] with τ_1 being the highest priority. We denote the subset of tasks with priority higher than task τ_i as $hp(i)$.

In this work, to simplify the analysis we make further assumptions about the task structure. Rather than allowing an arbitrary number of phases (memory / execution) per task, we instead have each job consisting of exactly three phases. Each task instance starts and ends with one memory phase and there is one execution phase between these two memory phases. Intuitively, this allows a job to fetch memory during its first phase, perform execution during its second phase, and write back its results to memory during its third phase. We index the memory and execution phases separately starting from 0: the first and the last memory phases are m_0 and m_1 and the execution phase in the middle is e_0. We will refer to these phases as m_0, e_0 and m_1. Since we will not allow more than m_{memo} cores to access memory at a time, the duration of the memory phases does not depend on the number of other cores concurrently accessing memory. Hence, the task's worst case execution time C is equal to the sum of these three phases, $C = m_0 + e_0 + m_1$. We will relax the assumption on the number of phases in future work and for now we focus on the 3-phase task model.

For the sake of simplicity, we do not consider now interference from peripheral access in this work, although the scheduling method and analysis can be easily integrated with



(a) Freescale P4080



(b) Intel Xeon

Fig. 1. Observed performance slowdowns on P4080 and Intel Xeon platforms. Each task accesses different banks in *DiffBank* while all tasks access the same bank in *SameBank*

our earlier work on I/O scheduling [7], [22].

B. Task Model Justification

A task model where each job consists of phases which perform memory access and phases which work on locally cached data was first proposed as part of our earlier work on the Predictable Execution Model (PREM) [22]. In this work, embedded benchmarks were modified in order to be made PREM-compliant, without significantly increasing total memory accesses time or total execution time. Since we target commercially-available platforms, we assume a cache-based system where the core fetches required instruction and data for the task from main memory into last level cache. While some embedded platforms [12] provide a cache-stashing mechanism that allows memory transfers performed by a separate DMA component to be optionally loaded into cache, such mechanism is not widely available nor generally guaranteed to succeed; hence, we do not employ it in this work.

In previous work on memory-access scheduling and multi-core scheduling for PREM-compliant tasks, only one core was allowed to access main memory at a time (m_{memo} =1) [22], [30]. However, this often turns out to be overly pessimistic because in reality there is significant memory-level parallelism. To illustrate this, we performed experiments on two multicore platforms using an engineered synthetic task that only accesses a selected memory bank. Figure 1 shows the performance slowdown of the mentioned task while varying the number of interfering tasks (each core runs a single instance of the synthetic task) on the two multicore platforms in two different settings: *SameBank* and *DiffBank*. For DiffBank, each task accesses its own memory bank and experiences almost no slowdown even as the number of interfering cores increases. For SameBank, which represents the worst-case scenario, all tasks access the same memory bank (namely bank 0) and the observed task suffers significant interference as the number of cores increases. Even in this worst-case scenario, however, the P4080 platform offers a certain degree of memory parallelism, while less so in the case of the Xeon platform.

The m_{memo} variable is introduced to account for such memory level parallelism into the analysis framework. As technology develops, and depending on the specific platform and the tasks used, the value of m_{memo} may be varied, so we perform a parameterized response time analysis. To precisely measure the parallelism of system memory capacity involves other works, for example, the assignment of tasks to memory banks, which is not the main focus of this work. Finally, our model, of course, is compatible with research work which assumes memory as being a single resource by setting m_{memo} =1.

A mechanism is needed to prevent the above-described problems of self-eviction, intracore eviction, and intercore eviction. The simplest approach is to partition the cache for each task. Better, colored cache lockdown, which is supported by many embedded processors, permits individual pages of memory to be placed in the last-level cache and locked, so that they can not be evicted until they are explicitly unlocked [19]. One potential problem with these approaches is the limited cache space. Recently, it was presented how to identify and lock in cache only the hot pages of each task [19], and use analysis to bound the effects of memory interference for non-locked pages.

In our analysis, we assume the length of each phase, m_0, e_1 and m_1, is an enforced constant. This can be done in implementation, for example, by implementing busy waiting at the end of a phase if execution proceeds faster than the worst case [22]. This modification will not degrade worst-case response time, but it may actually improve the scheduling bound. For example, consider a two-cores, two-tasks system (one task running on each core) where m_{memo} =1 and memory access is prioritized. A shorter-than-expected execution phase from the higher priority task will cause more memory access by the higher priority task in a given time window, which in turn can make a lower priority task have to wait longer before it can access memory (degrading its response time). If the execution phase time of the higher priority task is enforced as constant, however, the lower priority task will be able to get memory access while the higher priority task's execution phase is executing.

C. Summary of Response Time Analysis for Global Multicore Scheduling

While response time analysis [6] has been around for the single core systems for a long time, only recently this technique has been applied to the multi-core case [9], [10], [13], [16]. The formula for the multi-core case is expressed in the following lemma.

Lemma 1. *(Theorem 7 in [10]) An upper bound of the response time of a task τ_k in a multi-core system scheduled with global fixed priority can be derived by the fixed point iteration on R_k on the following expression, starting with $R_k = C_k$:*

$$R_k^{x+1} = C_k + \left\lfloor \frac{1}{m} \sum_{\tau_j \in hp(k)} \min\{I_k^j(R_k^x), R_k^x - C_k + 1\} \right\rfloor \tag{1}$$

where m is the number of cores and $I_k^j(R_k^x)$ is the upper bound of interference from task τ_j on time window R_k^x.

Observation 1. *When computing task response time, the sum of interference is divided by the number of cores, as all cores can execute in parallel. This is one main difference compared to the single core case, as we will show our scheduling analysis in next section, we will change this number depending on the execution or memory phase.*

Observation 2. *Computing the interference from one task in a specific time window is in general intractable and hence, we compute the workload, denoted by $W(L)$, this task can generate during this window as the upper bound of interference. Workload captures the amount of execution that can occur in a fixed window of time and clearly it holds that: $I_k^i(x) \leq W_i(x)$.*

Computing the workload is based on three terms: *carry-in*, *body* and *carry-out*. A carry-in job has its arrival time before the considered time window and its deadline within the time window. A carry-out job has its arrival time within the window but its deadline is outside the window. Body jobs have both an arrival time and deadline within the considered time window. To compute a workload upper bound, the computation has the carry-in job arriving as late as possible and the carry-out job as early as possible. This is shown in Fig.2. Here, two cases are shown, one where the time window $L = 6$, and one where the time window $L = 1$. The first job finishes just before its worst case response time and the second job starts execution right on arrival. The slack of this task, which is the amount of time the task could be delayed without causing a deadline miss, is denoted as s. For example in Fig.2, the first instance of task has slack 1, meaning that it can be delayed at most 1 time unit before missing the deadline.

The workload from task τ_i during time window L is upper bounded by the following equation (Eqn.(8) in [10]):

$$W_i(L) = N_i(L) * C_i + \min\{C_i, L + D_i - s - C_i - N_i(L) * T_i\}. \tag{2}$$

where $N_i(L)$ is the count of carry-in and body jobs. It can be calculated as (Eqn.(3) in [10] and Eqn.(5) in [16]):

$$N_i(L) = \left\lfloor \frac{L + D_i - s - C_i}{T_i} \right\rfloor. \tag{3}$$

For example, consider again the example task showed in Fig.2. The task has its relative deadline D equal to the period $T = 5$, its worst case execution time $C = 2$ and the slack $s = 1$. In first case, we are trying to compute the workload with a time window of length $L = 6$. The carry-in includes

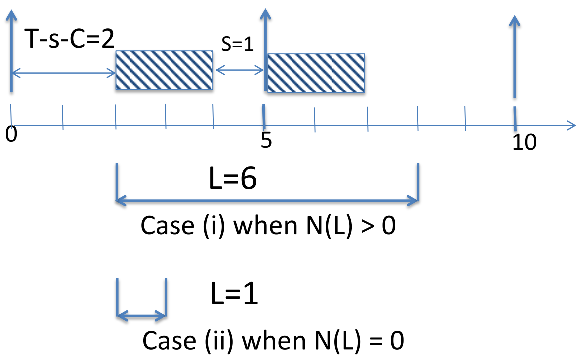


Fig. 2. The computation of the workload for two time windows of different lengths L for one task. There are two different cases depending on the value of $N(L)$, as in Eqn.(3)

one whole job and $N_i(L) = \lfloor \frac{6+5-1-2}{5} \rfloor = 1$. Hence, the workload generated by this task during this time window can be computed as $W_i(L) = 2 + \min\{2, 6 + 5 - 1 - 2 - 5\} = 4$, which is exactly showed in the figure. The interference from this task, then, can be upper bounded by the computed workload.

Now we introduce one minor improvement on the traditional computation of workload. Consider this same example task in Fig.2, assume the time window is short and $N_i(L)$ is equal to 0. For example, as in case (ii) in the figure, when $L = 1$ and $N_i(L) = 0$, the workload is upper bounded by 1, while Eqn.(2) would yield the result of $\min\{2, 1 + 5 - 1 - 2\} = 2$.

The key observation in the computing of workload is that $N_i(L) = 0$ *should be considered a special case and it is different from the Eqn.*(2). When $N_i(L) = 0$, the time window ends within the task period and it does not stretch out into the next period. Therefore, we can directly use the minimum of this time window length and the task execution time as the workload.

Based on this, we use the following equation to calculate the workload in the rest of the paper.

$$W_i(L) = \begin{cases} \min\{C_i, L\} & \text{if } N_i(L) = 0 \\ Eqn.(2) & \text{otherwise} \end{cases} \tag{4}$$

III. The Global Memory-Centric Scheduling Policy

In this section, we introduce a global memory-centric scheduling policy and illustrate it with an example task set. We will elaborate on the motivation, the rules and the effects of the proposed approach. In the next section we will consider its schedulability bound.

Traditional real-time scheduling methods focus on CPU scheduling, and do not consider memory access. As the number of cores increases, the effects of the comparatively limited memory bandwidth will increasingly become apparent in real-time systems. The proposed scheduler, therefore, takes memory access into account when deciding which task to execute.

We now list the three rules of the proposed global memory-centric scheduling approach:

1) In order to prevent slowdowns due to memory interference, a maximum of m_{memo} cores are allowed to access memory at the same time. If there are more than m_{memo} tasks ready to execute a memory phase, the lower priority ones are blocked and do not execute.

2) Priority of memory phases is promoted over execution phases. Tasks which are executing memory phases dynamically have their priority increased above all tasks not accessing memory. Relative priorities among tasks running memory phases are maintained. Therefore, a task with a pending memory phase will always preempt any running task which is in an execution phase, except in cases when this would conflict with the first rule.

3) Tasks priorities are assigned according to task periods (rate monotonic). Tasks with shorter periods are given higher system priority and will therefore always execute ahead of all pending tasks with longer periods, except in cases when this would conflict with the first or second rule.

Given these three rules, when a task needs to start a memory phase, there are several possible outcomes depending on the state of currently-running tasks. In the first case, if all the cores are busy running other tasks but the memory is not fully loaded (the number of tasks running memory phases is less than the memory capacity m_{memo}), the memory phase starts immediately by preempting the lowest active execution phase across all cores. This is because, according to rule 2, a memory phase runtime priority is always higher than any execution phase. In the second case, if memory is fully loaded and the priority of the pending memory phase is higher than the lowest priority of one of the running memory phases, the task will preempt the lowest priority memory phase. Notice the number of running memory phases is bounded by m_{memo} in accordance with rule 1. In the third case, if the memory is not fully loaded and one or more cores are idle, the memory phase can start immediately on one of the idle cores. In all other cases, the current memory phase cannot execute and the task is blocked.

Our proposed memory-centric scheduling policy leads to an important property at runtime: *the isolation of memory and execution phases*. Since all memory phases have higher priority than execution phases, free memory resources are always available for memory phases, if there are any active ones. All memory phases can therefore be considered as if they were running on one of m_{memo} *virtual memory cores*, without being interfered by the execution phases. Likewise, there are always m_{core}- m_{memo} *virtual execution cores* available to exclusively run execution phases. This key realization is illustrated in the following example, and it is the cornerstone of the schedulability analysis in the next section.

Another note about our proposed memory-centric scheduling is that this method is targeting memory intensive applications: by promoting the memory access and isolating the memory access and local execution, we can better bound the task stall time due to memory contention and estimate the worst-case response time. When the system memory load is low, our method loses its main benefits as the virtual memory cores are under utilized and wasted. This will be better shown in the evaluation section.

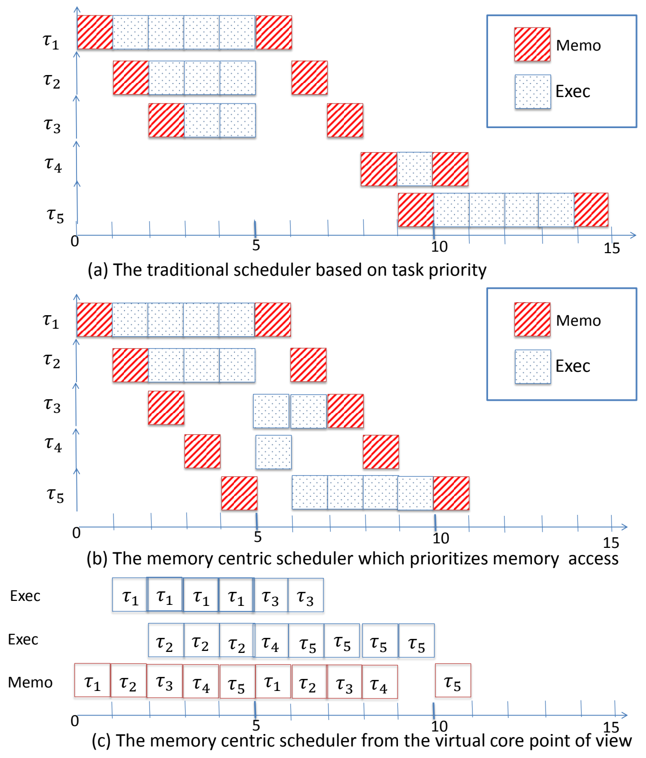


Fig. 3. A scheduling trace for five tasks on a system with 3 cores (m_{core} = 3) and memory capacity 1 (m_{memo} = 1). Compared to an approach without memory promotion (without rule 2, shown in (a)), our proposed memory-centric policy (shown in (b) increases the parallelism of the system and reduces the task set response time. The trace in (c) shows the same schedule trace as in (b), but from the view of the virtual memory core and virtual execution cores. The traditional scheduler in (a) represents the one that does not arbitrate access to memory, but suffers the effects of bus contention.

A concrete task set example is presented to better explain the rules of our proposed memory-centric scheduling. Fig.3 depicts a set of five tasks which are released at time 0, each row in the figure shows the trace of a single task. Here, we show each task with starting and end memory phase length 1 ($m_0 = m_1 = 1$) and an execution phase length between 1 to 4. We assume that the system consists of three cores (m_{core} = 3) so that at any time instance there are at most three tasks running concurrently. We further assume the system memory capacity is 1 core's memory bandwidth (m_{memo} = 1), hence, only one task can access memory at a time. The tasks are assigned global priority with task 1 having the highest priority. In Fig.3(a) and (b), each line represents the execution progress of each task: we do not show the trace for each core but only limit the number of concurrently running tasks. Fig.3(c) shows the same trace as in (b), but we group the execution and memory phases from all tasks.

Rule 1 limits the number of concurrent cores which can access memory. It is imposed by the architecture and is necessary to prevent a slowdown due to memory interference. Rule 3, priority assignment from task rates, is the standard global RM scheduling rule. We therefore more closely examine Rule 2, which promotes memory phases above execution phases. The effects of this rule can be seen by comparing the trace without rule 2 in Fig.3(a) with the trace using rule 2 in Fig.3(b).

Observation 3. *Without memory promotion (rule 2), we may not fully utilize the parallelism of the hardware system and this results in a long response time.*

In Fig.3(a), task 4 can start only after the three higher priority tasks finish, and since one task cannot run in parallel on more than one core, during the time window [5, 9], only a single core is actively running. This leads to a long response time for task 5. The underlying reasoning is that the memory phase is delayed by both the memory and execution phases of higher priority tasks. In turn, this memory phase delays its subsequent execution phase.

Observation 4. *Memory promotion makes the memory phase finish earlier and increases the system parallelism.*

In Fig.3(b), at time 3, the memory phase of task 4 starts by preempting the execution phase of task 3 (the lowest priority at that time among the 3 active tasks). Similarly, task 5 at time 4 preempts the execution phase of task 4. This makes the memory phases of these two tasks finish earlier compared to the case without memory promotion in Fig.3(a).

Observation 5. *Memory-centric scheduling provides isolation between memory accesses and local computation.*

Here isolation means that memory phases are not subjected to interference from execution phases. As shown in Fig.3(b), from time 0 to time 9, the memory is continuously utilized. We can consider the memory resource as m_{memo} virtual cores even though in reality the memory phases are running physically on different cores. The memory phases are scheduled on this virtual core according to their priority level. Fig.3(c) provides a per-virtual-core based view of the same trace as in (b). Since memory can only be accessed by one core at the time (m_{memo} =1), the memory phases can be considered as running on this virtual memory core, while the remaining execution phases are running on the two virtual execution cores (m_{core}-m_{memo} =2). This provides the basic idea for the proposed schedulability analysis which will be detailed in the next section. Notice that considering having only m_{core}-m_{memo} =2 virtual cores which run execution phases can be pessimistic; in fact, if no memory phases are executing, up to m_{core} cores could concurrently run execution phases.

Observation 6. *Under certain conditions, memory promotion can improve task response time.*

Since the memory phase of a low priority task is promoted above the execution phase of any other tasks, after it finishes the subsequent execution phase can take advantage of the system parallelism and run concurrently with the execution phase of other tasks. This is what exactly happened in Fig.3(b): the first memory phases of task 4 and task 5 preempt the execution phase of task 3. At time 5 after these two memory phases finish, the execution phases of task 4 and task 5 can run concurrently with task 3, reducing the task set response time.

Observation 7. *The virtual memory core can be idle even though there are unfinished but not ready memory phases.*

This case happens in Fig.3(c) when the memory core is suspended during time [9, 10], even though task τ_5 still has an unfinished memory phase. This is because there are precedence constraints between memory phases and execution phases; in fact, the memory phase can start only after the completion of the preceding execution phase. This imposes new challenges in the response time analysis for the memory cores: we cannot compute the delay from memory phases by simply summing up the length of memory phases and performing standard response time analysis. We will elaborate on this in the next section.

IV. The schedulability analysis

Having introduced the memory-centric scheduling policy, we provide the corresponding schedulability analysis in this section. We will first give an overview of the methodology and break down the analysis into two main problems, then we will introduce the solution to both of them in detail.

A. Methodology overview

As consequence of memory promotion, all memory phases can be considered as if they were running on m_{memo} virtual memory cores, and in the worst case, the execution phases can run only on the remaining m_{core}-m_{memo} cores. Although analyzing the system like this will be pessimistic in terms of the amount of execution that can occur in the system, the motivation for this work is that memory is increasingly becoming the bottleneck in multicore real-time systems, whereas execution is continuing to scale up according to Moore's law. Each task is composed of a sequence of alternating memory and execution phases, hence, we can apply the global response time analysis on these m_{memo} and m_{core}-m_{memo} cores separately to compute the response time of one memory/execution phase. However, existing CPU-only global response time analysis cannot be directly applied here, as each task consists of two different type of requests which interfere in different ways. First, from the interfering task point of view, interference can only be caused when its phase type matches the phase type of the task under analysis. Secondly, from the point of view of analyzed task, even though computing the response time of one phase is possible by exploiting memory/execution isolation, the job response time is usually less than the sum of each phase response time, and to get a tighter bound we must take some combination of these phases response times. These challenges can be summarized in the following two questions which will, in turn, be addressed in the next two subsections.

- Given an interfering task and a fixed interval of time, how can we compute the maximum interference (workload on memory or execution) generated by this task?
- Given the computed workload from interfering tasks and the response time of the phases, how can we compute a task response time that is tighter than a direct sum of phase response times?

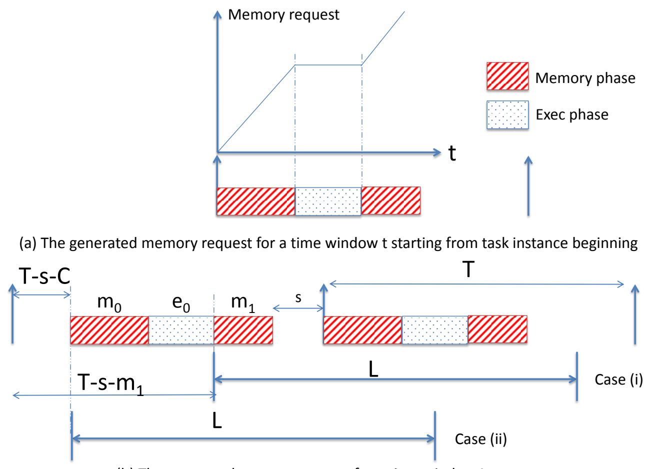

Memory request
t
Memory phase
Exec phase
(a) The generated memory request for a time window t starting from task instance beginning
T-s-C
T
m_0
e_0
m_1
s
T-s-m_1
L
Case (i)
L
Case (ii)
(b) The generated memory request for a time window L

Fig. 4. The calculation of memory workload from one task during a time window of a certain length requires iteration on its phases.

B. The workload during a length of time

In this subsection, we will first consider the memory workload, and then address execution phase interference. Our goal is to, given a fixed interval of time L, provide functions $W_m(L)$ and $W_e(L)$ which compute the maximum memory workload and maximum execution workload, respectively. Since these functions are defined per task, we compute them with respect to an arbitrary task with phases m_0, e_0 and m_1, task period T and computation time C.

In order to get the maximum memory workload generated by a task in a given period of time, we assume a worst-case task arrival pattern [10], where the carry-in job (the job that straddles the start of the interval of time) finishes as late as possible and the carry-out job (the job that straddles the end of the interval of time) starts as early as possible. This can be imagined as pushing the two end tasks toward the center in order to maximize the amount of memory access in the time window being considered. In this way, more memory phases occur during the time window, and therefore the interference is maximized.

Each interfering task instance has two memory phases. The maximal memory workload can be obtained by aligning the start of the time window with the start of either one of the two interfering memory phases. We therefore compute the workload for each of these two cases and take the maximum to get an the upper bound on possible interference.

When aligning the time window with the start of a memory phase, the carry-in job memory workload is the sum of memory phases starting from this phase to the end of this task instance. The body jobs, if any, contribute both of their memory phases to the workload being computed. Finally, the carry-out job contributes to the memory load from its first memory phase until the end of the time window. In order to compute the memory workload of the carry-out job, we define a function $\mathcal{F}_m : \mathbb{R} \to \mathbb{R}$, which takes as input the time window of length t starting from the task beginning, and outputs the total memory load during this time window. This function, with respect to our sample task with phases m_0, e_0 and m_1, and can be expressed as follows:

$$\mathcal{F}_m(t) = \begin{cases} t & \text{if } t \le m_0 \\ m_0 & \text{if } m_0 < t \le m_0 + e_0 \\ t - e_0 & \text{if } m_0 + e_0 < t \le C \\ m_0 + m_1 & \text{if } C < t \le T \end{cases} \tag{5}$$

Figure 4(a) shows an example of this function $\mathcal{F}_m(t)$. The x-axis is the length of the time interval and the y-axis is the maximal memory request generated during the time interval. With the function $\mathcal{F}_m(t)$, we can formalize the memory workload in a way similar to Eqn.(4).

For the case where the window is aligned with the start of the *second* memory phase, the workload generated by the sample task during a period of time L is:

$$w_m(L) = \begin{cases} \min\{m_1, L\}; \text{ if } N(L) = 0 \\ m_1 + (N(L) - 1) * (m_0 + m_1) + \mathcal{F}_m(lco); \text{ else} \end{cases} \tag{6}$$

where $N(L)$, the number of carry-in and body jobs in an interval of time L for our sample task, is computed similar to Eqn.(3)

$$N(L) = \lfloor \frac{L + T - s - m_1}{T} \rfloor,$$

and lco is the Length within the Carry-Out job:

$$lco = L + T - s - m_1 - N(L) * T.$$

When $N(L)$ is 0, the time window finishes within the first task instance, and the memory workload is no larger than m_1 or L (the first part of Eqn.(6)). Otherwise, the carry-in job contributes m_1, the $N(L) - 1$ body jobs contribute $m_0 + m_1$ each, and the final carry-out portion is computed using the $\mathcal{F}_m$ function.

For the other case, when the time window is aligned with the start of the *first* memory phase, the memory workload can be computed as:

$$w'_m(L) = \begin{cases} \mathcal{F}_m(L); \text{ if } N(L) = 0 \\ N(L) * (m_0 + m_1) + \mathcal{F}_m(lco); \text{ else} \end{cases} \tag{7}$$

where the number of carry-in and body jobs $N(L)$ is

$$N(L) = \lfloor \frac{L + T - s - C}{T} \rfloor,$$

and lco is

$$lco = L + T - s - C - N(L) * T.$$

Finally, the resultant memory workload during a period of time L, is computed by taking the maximum of two previously computed workloads:

$$W_m(L) = \max\ (w_m(L), w'_m(L)) \tag{8}$$

An example of computing the memory workload is shown in Fig.4. As mentioned before, Fig.4(a) depicts the $\mathcal{F}_m(t)$ function for the 3-phase task. This function increases when the end of time window falls inside a memory phase, either m_0 or m_1, and it remains constant when it is inside the execution phase, as this does not contribute to the memory workload. Fig.4(b) shows that the workload is computed for two different alignment cases. In case (i), the time window is aligned with the start of second memory phase and the total memory workload is $m_0 + 2 * m_1$. In case (ii) the workload is $2 * m_0 + m_1$. Since in general we cannot assume the specific arrival pattern of the interfering tasks, the final result for memory workload would be the maximum of these two values.

The calculation of execution workload can be done in a similar way. In our 3 phases task model, there is only one alignment situation, and the workload is expressed as:

$$W_e(L) = \begin{cases} \mathcal{F}_e(L); \text{ if } N(L) = 0 \\ N(L) * e_0 + \mathcal{F}_e(lco); \text{ else} \end{cases} \tag{9}$$

where the number of carry-in and body jobs $N(L)$ is

$$N(L) = \lfloor \frac{L + T - s - C + m_0}{T} \rfloor,$$

and lco is

$$lco = L + T - s - C + m_0 - N(L) * T.$$

and function $\mathcal{F}_e(x)$ is similar to $\mathcal{F}_m(x)$ but computes the workload based on the execution phase length. The function $\mathcal{F}_e(x)$ can be formalized as

$$\mathcal{F}_e(t) = \begin{cases} 0 & \text{if } t \leq m_0 \\ t - m_0 & \text{if } m_0 < t \leq m_0 + e_0 \\ e_0 & \text{if } m_0 + e_0 < t \leq T \end{cases} \tag{10}$$

C. Improving task response time

Having shown how to compute the upper bound on memory and execution phase workloads, we can proceed to compute the response time for a single phase using the method in Eqn.(1). Since a job is composed of a sequence of phases, and one phase can start only after the previous one finishes, one simple way to compute a bound on job response time is by summing up the response time of each phase.

Consider computing the response time of a memory phase m of task τ_k. Its worst case response time, denoted as $R_m(m)$, can be computed as the fixed point iteration on the following equation:

$$R_m^{x+1} = m + \left\lfloor \frac{1}{m_{memo}} \sum_{\tau_j \in hp(k)} \min\{W_m(R_m^x), R_m^x - m + 1\} \right\rfloor$$

where W_m is computed using Eqn.(8).

Notice here the total interference is divided by m_{memo}, the number of virtual memory cores.

Likewise, the response time of the execution phase e, denoted as $R_e(e)$, can be computed by the fixed point iteration:

$$R_e^{x+1} = e + \left\lfloor \frac{1}{m_{exe}} \sum_{\tau_j \in hp(k)} \min\{W_e(R_e^x), R_e^x - e + 1\} \right\rfloor$$

where the number of available cores for execution m_{exe} = m_{core} - m_{memo} .

As mentioned before, one simple way to calculate the response time of 3 phases task, denoted as R_{τ_k}, is to sum up the response time of each phase: $R_m(m_0)+R_e(e_0)+R_m(m_1)$.

However, the result of this computation may be unnecessarily pessimistic. This is because this bound accounts for the worst-case interference pattern for each phase individually, whereas during execution all three worst-cases may not be able to occur at the same time. To try to improve the computed response time, we consider combining phases and computing the response time as one *merged* memory phase. Since both of these approaches are valid bounds, we take the minimum of this bound, and the direct summation bound.

In order to consider the whole job as a single memory phase, we need to decide what is its *equivalent* memory access time. After the completion of the first memory phase (m_0), the second memory phase can be delayed in the worst case as much as the response time of e_0, which can be computed as $R_e(e_0)$. The equivalent memory length of merged phase, therefore, is equal to $m_0 + R_e(e_0) + m_1$.

Fig.5 illustrates one example of this. The top line shows the m_{core}-m_{memo} cores serving the execution phase and the bottom line is for the m_{memo} virtual memory cores. Here, the task execution and the corresponding interference are clustered in one box to simplify the figure; in reality they may happen at different time instances. The interference is represented using a higher box compared to the task execution as it may use all the m_{memo} and m_{core} virtual cores, while the task execution and memory access cannot be parallelized. This figure exactly captures the task worst-case response time, similar to the one shown in [10]. The difference here, which is brought on because we having different phases, is that the second memory phase has to wait for the completion of the preceding execution phase. The combined memory phase, therefore, is scheduled as if it had a memory phase time of $m_0 + R_e(e_0) + m_1$. By applying the memory phase response time analysis on this equivalent merged memory phase, the worst-case response time can be computed as : $R_m(m_0 + R_e(e_0) + m_1)$.

Finally, since the task response time cannot be larger than either of these two cases, we use the minimal of these two results as the upper bound of task response time. Putting these two together, task response time R_τ can be computed as

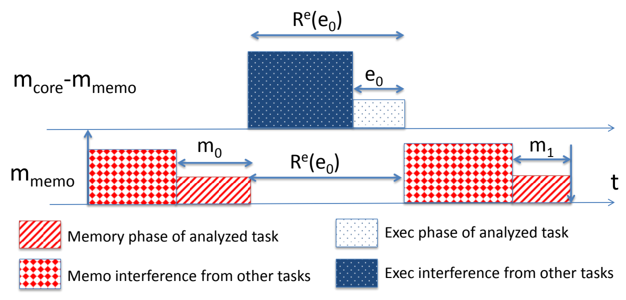
$$\begin{aligned} R_\tau = \min\{&R_m(m_0) + R_e(e_0) + R_m(m_1), \\ &R_m\big(m_0 + R_e(e_0) + m_1\big)\}. \end{aligned} \tag{11}$$

Intuitively, if the execution phase between these two memory phases is small, adding its response time into the memory phase length would compensate the pessimism brought on by computing the worst-case memory workload twice, and we expect the second term of the minimum to be smaller. On the other hand, if the execution phase is much larger than the memory phase, simply summing up the response times of all three phases may be a better option, and the first term of the minimum would probably be smaller.

D. Method summary and considerations

The entire procedure to perform schedulability analysis can be summarized as follows:

1) We perform the analysis in the global priority order. Hence, the slack information of higher priority tasks is available when we consider each task.
2) For each task under analysis, we first compute the response time of each individual phase, and then we compute the response time when the whole task is



R^e(e_0)
m_core-m_memo
e_0
m_0
R^e(e_0)
m_1
m_memo
t
Memory phase of analyzed task
Memo interference from other tasks
Exec phase of analyzed task
Exec interference from other tasks

Fig. 5. The computation of the task response time using the merged phase method. The task response time can be computed as the memory response time of a memory phase of length $m_0 + R_e(e_0) + m_1$.

considered as one merged memory phase. The task response time is obtained by taking the minimum of these two, as in Eqn.(11).

3) We can terminate the procedure once we find a task that is unschedulable. Otherwise, we update the task slack information and move on to the next task.

Our method considers the memory as multi-unit resource by using the variable m_{memo} to control the level of memory parallelism. It is also backward compatible to the previous research work which assumes the memory is an exclusive resource. This is achieved by setting m_{memo} =1. From the schedulability analysis point of view, as pointed out in [9], multicore response time analysis is also compatible with the single core case. Hence, our analysis method would still hold when setting m_{memo} =1.

Although we have presented the framework in terms of the three phases task model, it is worth examining what needs to be changed to allow for an arbitrary number of phases per job. With multiple phases, the key difference is that computing the workload from higher priority tasks needs to examine all possible alignments. The challenge with this is that there are many different possible phase combinations of higher-priority tasks, and a brute force algorithm may lead to exponential complexity. We are currently examining ways to address this problem by employing a dynamic programming approach that can re-use previously computed phase alignments, which has polynomial complexity. The main target of this work is to propose a new memory-centric scheduling policy and provide the corresponding analysis; the dynamic programming approach and its analysis on the multiple phases task model will be provided as part of our future work.

V. Evaluation

This section presents analytic evaluation results of the proposed memory-centric scheduling algorithm, denoted as *MemCentric*, using a set of generated synthetic workloads. For comparison, we also use a baseline scheduling algorithm, denoted as *Baseline*, that schedules tasks based on their static priorities without prioritizing memory phases over execution phases, as presented in [10]. *Baseline* differs from [10] in that tasks follow the PREM task model and only m_{memo} cores can access memory at any time instance. According to a more realistic model than one described in [10], the tasks suffer the effects of bus contention and in the worst case, when all m_{core} cores try to access memory at the same time, each memory access time can be increased by m_{core}/m_{memo} times. Therefore, under worst-case scenario, task execution time is increased to

$$\frac{m_{core}}{m_{memo}} * (m_0 + m_1) + e_0. \qquad (12)$$

We perform the analysis as in [10] with this increased task execution time and use this result as *Baseline* for the comparison.

In generating task sets, we control two parameters: the average core utilization and the average memory utilization. The average core utilization is defined as the sum of each task's core utilization, which is computed as the task's computation time (including all memory and execution phases) divided by the task's period and the number of cores. Likewise, the average memory utilization is the sum of each task's memory utilization, which is the task's memory phases divided by the period and the memory capacity m_{memo}. Intuitively, these two utilization values measure overall computational and memory loads (the utilization value of one means fully loaded system).

We model a 8-core platform with the memory capacity of two: i.e., m_{core} =8 and m_{memo} =2. We vary the average core and memory utilization within the range of $[10\%, 60\%]$. The number of tasks is varied in the range of [8,24]; and for each task, the period is chosen in the range of [5000, 50000]. For a chosen overall computation and memory utilization pair $[U_{core}, U_{memo}]$, we randomly assign per-task computation and memory utilization values in the range of $[U_{core}/3, U_{core}]$ and $[U_{memo}/12, U_{memo}/4]$, respectively. For the chosen utilization and period values of each task, we calculate lengths of the task's execution phase e_0 and split the memory into m_0 and m_1 such that the ratio difference between these two is less than 1.5. Finally, each task is assigned a relative deadline equal to its period and a priority according to Rate Monotonic scheduling algorithm.

Figure 6 shows the schedulability level achieved by *MemCentric* (left) and *Baseline* (right). In this experiment, we randomly generate one hundred thousand task sets, evenly distributed in the two-dimensional space of core utilization (Y-axis) and memory utilization (X-axis). The Z-axis shows the percentage of schedulable task sets. As observed in the figure, when average core and memory utilization level are low, both *MemCentric* and *Baseline* can achieve a very high schedulability level; and as the core or memory utilization increases, the schedulability level gradually drops to zero. One clear advantage of *MemCentric* compared to *Baseline* is that *MemCentric* is much less sensitive with respect to memory utilization: the schedulability decreases much slower when memory utilization increases. This is because our proposed method promotes memory access and has them scheduled on the dedicated memory cores; therefore, their impact to the schedulability level is small as long as they are not saturated. Overall, out of the 100 thousands randomly generated tasksets in this simulation, 56.3% of tasksets can be scheduled in *MemCentric* while only 38.3% can be scheduled in *Baseline*.

To better understand the difference between the two methods, Figure 7 shows a contour line at the 50% schedulability level

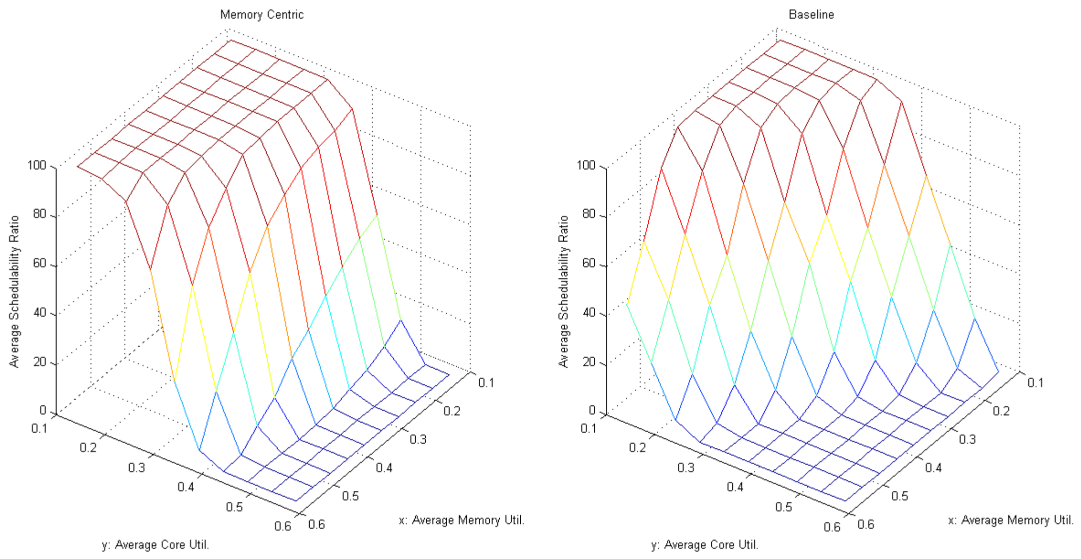


Fig. 6. In an 8-core system with memory capacity 2, memory-centric scheduling outperforms baseline scheduling.

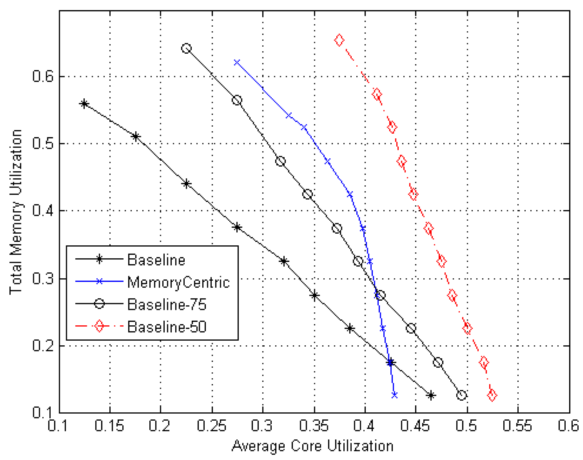


Fig. 7. The 50% schedulable utilization boundaries such that taskset generated with two utilizations on that curve has half chance to be schedulable. The curves Baseline and MemoryCentric is derived from Fig 6, the Baseline-50 and Baseline-75 are for different slowdown factors

of Figure 6. This can be better understood by imagining an intersection between a horizontal plane at 50% schedulability ratio level and the plotted three dimensional shape. In the intersection line, each point corresponds to the pair of memory and core utilizations, under which the ratio of schedulable tasksets is 50%; and the taskset generated with utilization pair in the area below the intersection line will have schedulability level larger than 50%. We also include two other contour lines, Baseline-75 and Baseline-50, which will be explained in the next paragraph. As shown in the figure, *MemCentric* covers a larger area compared to *Baseline*. At 50% memory utilization, for example, *MemCentric* can schedule twice more task sets, extending the core utilization from 18% to 35%. This means that when the memory load is high, *MemCentric* performs better by promoting the priority of memory phases so that it would not be interfered by the execution phases. Note, however, that two curves intersect around the memory utilization level of 20%; the *Baseline* performs better below this memory utilization. This is because of the pessimism in our *MemCentric* analysis: it assumes the execution phases are only executed on the m_{core}-m_{memo} virtual cores, even though the memory load is low. Since we are more interested in the cases when memory load is high, we believe this is not a serious problem.

As shown in Section II-B, the slowdown of memory access due to interference may vary depending on many factors, including the underlying hardware architecture. Considering this fact, simply increasing the memory phase time by a factor of $\frac{m_{core}}{m_{memo}}$ can become inaccurate in calculating the Baseline. To cope with this, we reproduce the figure with same configurations but with different slowdown factors: Baseline-75 represents the situation when the memory phases are slowed down by $0.75 * \frac{m_{core}}{m_{memo}}$ and Baseline-50 as 0.5. This work is by no means to derive a precise slowdown factor, rather, we are motivated by the fact that task may suffer different level of slowdown and we are interested in gauging the schedulability impact on *Baseline*. We examine the slowdown factor in the range of [0.5, 1] and we believe this will provide some insights to estimate the schedulability impact. It is also worth mentioning that our proposed *MemCentric* is not affected the memory slowdown factor as it schedules memory phases according to their relative priorities on the dedicated memory cores.

As expected, when the memory phase is slowed down by a smaller factor, the schedulability bound provided by *Baseline* improves as it covers a larger area. The slope of Baseline curve also changes: when the memory slowdown is less severe (with smaller slowdown factor), *Baseline* can achieve the same schedulabilty level at a higher memory load. Comparing the set of Baseline curve with *MemCentric*, Baseline-75 and *MemCentric* still intersect, and this intersection point is 10%

higher memory utilization level and slightly smaller core utilization level, compared to the intersection point of Baseline and *MemCentric*. Also notice Baseline-75 and *MemCentric* cover similar size of area, and *MemCentric* has some marginal benefits only when at high memory load. When the scaling factor is 0.5, Baseline-50 always outperforms *MemCentric*, and this suggests that when the memory slow down is not severe, using a traditional algorithm may work better.

VI. Related Work

Schedulability analysis for multicore systems has been studied by many researchers. Bertogna et al. extended single-core response time analysis [6] to multicore, using the idea of carry-in and carry-out to upper-bound interference [10]. This approach has been further extended in several directions: for example, the non-preemptive case [14], [17] and the arbitrary deadline case [13]. Recently, Lee and Shin extended it to limited preemptive scheduling on multi-core systems [16]. Our analysis uses a similar idea to calculate the task response time, but is different in that it considers not only CPU but also memory in the analysis.

There are several existing timing analysis methods targeting contention on shared memory or bus under various access models [5], [23], [26]–[28]. Rosen et al. proposed a method to obtain efficient TDMA arbitration policies [24]. Our previous work [30] is also based on TDMA; tasks are statically partitioned to cores and each core is allowed to access memory only during its granted TDMA slot. In contrast, our current work is based on a global preemptive scheduler and tasks are modeled as if they were running on virtual cores. Since partitioned and global scheduling are generally incomparable, we do not directly compare against [30] in this work.

Task preemption and migration are challenging in real-time scheduling because of cache effects. Anderson et al. proposed a cache-cognizant hierarchical scheduling method, based on Pfair scheduling discipline, that minimizes L2 cache-misses in multicore platforms [4] . Calandrino et al. proposed several heuristics, based on G-EDF scheduling, that improves cache performance of real-time applications [11]. Recently, Sarkar et al. investigated migration of locked cache lines for task migration under a global preemptive scheduling method [25]. Mancuso et al. proposed a cache management framework that combines page coloring and cache-lockdown to minimize cache interference in multicore platforms [19]. Because our work focuses on main memory, these techniques complement our work.

Interference in main memory is an important problem in modern multicore platforms and various solutions were investigated to address the problem. One direction is to minimize memory interference by proposing new hardware mechanisms. Akesson et al. proposed the Predator memory controller design that combines regulators and a credit based scheduler (arbiter) to provide latency and bandwidth guarantees [1]. Paolieri et al. also proposed a memory controller design, called AMC, to provide performance guarantees but this work differs in that it uses a round-robin based arbiter [21]. While valuable, these works require modification in hardware, and are not available in COTS components. Another direction is software-based approaches that use OS schedulers to coordinate memory accesses in such a way to minimize interference. The work presented in this paper falls into this category. One closely related work is MemGuard [31] which provides per-core memory bandwidth reservations for memory-performance isolation. The main difference is that current work benefits from the PREM task model and is based on a global scheduling algorithm that allows task migration.

Apart from [30], several scheduling works based on PREM or similar models have been published in recent years. The most closely related work is [2], which similarly considers global scheduling of PREM tasks. However, the model in [2] is incomparable with the one in our current work: in [2], the authors assume that the execution phase of each task must complete before the task's deadline, while the subsequent write back phase is executed together with the memory phase of some other task, possibly after the deadline. In contrast, in this paper we argue that the write back memory phase of the task should also complete by the deadline, since output data requires timing guarantees. Furthermore, in [2] tasks are scheduled non-preemptively to reduce required cache size. For both reasons, we cannot provide a direct comparison with [2]. [3] discusses global scheduling of PREM-like tasks on scratchpad-based multicore systems; the work considers co-scheduling a separate DMA component to perform transfers from main memory to scratchpad and vice versa. In contrast, as discussed in Section II-B, our work targets commercially-available systems using cache memory; hence, the core itself must be used to prefetch data into local memory.

Finally, since we consider tasks divided in phases sequentially executed on virtually separate resources, our model has similarities to the ones used in distributed real-time computing approaches such as delay composition [15] and holistic analysis [20], [29]. However, to the best of our knowledge, there is no extension to delay composition for globally scheduled resources; such extension is not trivial, since the analysis framework requires a fixed order of job executions on all traversed resources. Our approach is similar to the general holistic analysis framework in that we obtain the task's end-to-end response time by considering the response times of individual phases; in fact, the technique used in Section IV-C to reduce the interference pessimism is akin to adding a separation (offset/jitter) of $R_e(e_0)$ between phases executing on the same resource (m_{core} memory cores). We do not directly employ holistic analysis because again, to the best of our knowledge, there is no published response time analysis for global scheduling with offsets and jitters.

VII. Conclusions

Main memory is an increasingly important shared resource in multicore based real-time systems. Traditional analysis focusing solely on the CPU may work poorly, especially when memory load is high, due to an increased worst-case stall time when accessing main memory. In this work, we have proposed a new memory-centric scheduling method based on the global scheduling algorithm. The novelty of our approach is that it prioritizes accessing memory over local CPU execution at runtime. We also have proposed a parametric schedulability analysis method, based on the idea of response time analysis on the virtual cores. Our analysis allows multiple cores to access memory concurrently, hence reduces analysis pessimism. The evaluation shows that the proposed memory-centric scheduling method can improve real-time task schedulability by almost a factor of two, compared to a non memory-centric scheduling approaches.

As future work, we plan to remove the assumption on the number of phases of each task, and incorporate cache effects in the schedulability analysis.

Acknowledgements

This research is supported in part by ONR N00014-12-1-0046, Lockheed Martin 2009-00524, Rockwell Collins RPS#6 45038, NSERC DG 402369-2011, NSF CNS-1302563, and NSF CNS-1219064. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the ONR, NSERC, NSF or the supporting companies.

References

[1] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable sdram memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '07, pages 251–256, 2007.

[2] A. Alhammad and R. Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, New Delhi, India, Oct 2014.

[3] A. Alhammad, S. Wasly, and R. Pellizzoni. Memory efficient global scheduling of real-time tasks. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Seattle, WA, Apr 2015.

[4] J.H. Anderson, J.M. Calandrino, and U.M.C. Devi. Real-Time Scheduling on Multicore Platforms. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 179–190, 2006.

[5] B. Andersson, A. Easwaran, and J. Lee. Finding an upper bound on the increase in execution time due to contention on the memory bus in cots-based multicore systems. *ACM Sigbed Review*, 7(1), 2010.

[6] N.C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[7] S. Bak, E. Betti, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time control of I/O COTS peripherals for embedded systems. In *Proc. of the 30th IEEE Real-Time Systems Symposium*, Washington DC, Dec 2009.

[8] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. Memory-aware scheduling of multicore task sets for real-time systems. In *RTCSA*, pages 300–309, 2012.

[9] S. K. Baruah. Techniques for multiprocessor global schedulability analysis. In *Real-Time Systems Symposium, 2007.*, pages 119–128, 2007.

[10] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Real-Time Systems Symposium, 2007.*, pages 149–160, 2007.

[11] J. M. Calandrino and J. H. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *ECRTS*, pages 299–308, 2008.

[12] Freescale. *P4080 website*. http://www.freescale.com.

[13] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *Real-Time Systems Symposium, 2009.*, pages 387–397, 2009.

[14] N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In *RTSS*, pages 137–146, 2008.

[15] P. Jayachandran and T. Abdelzaher. Delay composition in preemptive and non-preemptive real-time pipelines. *Real-Time Systems*, 40(3):290–320, 2008.

[16] J. Lee and K. G. Shin. Controlling preemption for better schedulability in multi-core systems. In *RTSS*, pages 29–38, 2012.

[17] H. Leontyev and J. H. Anderson. A unified hard/soft real-time schedulability test for global edf multiprocessor scheduling. In *RTSS*, pages 375–384, 2008.

[18] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[19] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[20] J.C. Palencia and M. Gonzàlez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[21] M Paolieri, E Quinones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *IEEE Embedded System Letter*, 1(4), Dec 2009.

[22] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded system. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, Chicago, IL, USA, April 2011.

[23] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of Design, Automation and Test in Europe (DATE)*, Dresden, Germany, Mar 2010.

[24] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Real-Time Systems Symposium, 28th IEEE International*, pages 49–60, 2007.

[25] A. Sarkar, F. Mueller, and H. Ramaprasad. Predictable task migration for locked caches in multi-core systems. In *LCTES*, pages 131–140, 2011.

[26] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 759–764, 2010.

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

[28] A. Schranzhofer, J.J. Chen, and L. Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 215–224, 2010.

[29] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 50:117–134, April 1994.

[30] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, 2012.

[31] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, Philladelpia, USA, April 2013.