HMB: Scheduling PREM-Like Real-Time Tasks at High Memory Bandwidth

Mohammadhassan Gholami Derouei
Università degli Studi di Modena e Reggio Emilia, Italy
Minerva Systems SRL, Modena, Italy

Paolo Valente
Università degli Studi di Modena e Reggio Emilia, Italy

Marco Solieri
Minerva Systems SRL, Modena, Italy

Andrea Marongiu
Università degli Studi di Modena e Reggio Emilia, Italy

Abstract

Current homogeneous and heterogeneous computing systems reach high performance through parallelization. Yet, parallel execution of tasks entails non-trivial latency-vs-throughput issues when it comes to concurrent accesses to shared memory. In this respect, effective bandwidth regulation solutions do exist, and provide a basic mechanism to control the latency of memory accesses. Such solutions, though, are often cumbersome to deploy and to configure to guarantee both bounded latency and high utilization of the memory bandwidth. The problem is that memory latency varies non-linearly with the number and type of concurrent accesses, and the latter may in turn vary with time, often unpredictably. For this reason, previous attempts at memory regulation in scheduling solutions resulted either in poor real-time execution guarantees, or in severe underutilization of the memory bandwidth. In this paper, we outline High Memory Bandwidth (HMB), a scheduling solution that guarantees bounded response times to real-time task sets through memory regulation, while also reaching a high utilization memory bandwidth. Since the complete solution is complex, just like the problem it addresses, this preliminary work defines in full detail only the core mechanism. This mechanism builds on the notion of memory access slowdown experienced by any processor performing back-to-back memory operations; this slowdown is due to the interference generated by other processors also accessing the memory at the same time. The core mechanism assumes that each processor can tolerate a certain amount of slowdown before the timing behavior of the task(s) it is running is compromised. Each processor has a priority assigned: the higher the priority, the more stringent the timing requirements. The slowdown can be controlled by regulating with precision the maximum amount of system bandwidth each processor is allowed to use, based on its priority. The proposed mechanism finds the maximum bandwidth each processor can use such that the highest number of processors simultaneously accessing memory is found (thus avoiding memory bandwidth underutilization) while guaranteeing that the slowdown of each processor is kept within the tolerated limits.

2012 ACM Subject Classification Computer systems organization → Real-time operating systems

Keywords and phrases Heterogenous systems, Parallel execution, Shared memory, Bandwidth regulation, Memory access, Real-time execution, Memory bandwidth utilization, High Memory Bandwidth (HMB), Memory access slowdown, Memory interference, Memory-centric scheduling

Digital Object Identifier 10.4230/OASIcs.NG-RES.2024.1

Category Invited Paper

Acknowledgements Authors are grateful to Tomasz Kloda at TUM, and Benjamin Rouxel at UNIMORE, with whom they enjoyed inspiring discussions in the conception of this work.

1 Introduction

Memory contention on parallel systems

Modern systems feature multiple execution units, including CPU cores and accelerator cores, running in parallel. These units may inherently perform memory accesses simultaneously. To support parallelism in memory accesses, these systems are equipped with cache hierarchies. Caches help to make memory accesses as local as possible, thereby reducing the likelihood of accessing shared memories and interconnects. However, in many cases, conflicting accesses to shared resources remain unavoidable. For instance, these conflicting accesses become unavoidable when there is insufficient space in local caches to accommodate the cumulative memory footprint of the processors (e.g. CPU cores, execution units) utilizing those caches. Additionally, such conflicts arise when distant processors need to communicate through shared memory. Contention arises across all shared resources along the path from the processors to the actual memory banks, including the interconnect, shared caches, memory bus, and others. This contention, in turn, results in a more or less significant and unpredictable inflation of memory latency, impacting the duration of the memory access performed by the competing processors. [10]. This is an evident problem in real-time applications. Increased memory latencies can slow down the execution of tasks that involve memory accesses. This may make it impossible for the tasks to meet their deadlines, thereby rendering the task set unfeasible. Several solutions have been proposed in the scientific literature to eliminate or control this slowdown, which can be broadly grouped into two classes: (i) exclusive memory accesses; and (ii) limited memory accesses.

Exclusive memory access: low bandwidth and high predictability

The first type of solution is based on allowing one processor (or, very few processors) at a time to access shared memory [20, 21]. This approach either eliminates or reduces interference to such a low level that no significant slowdown occurs. However, the available bandwidth from shared memories is typically sized to meet the cumulative average bandwidth demand of the set of processors connected to that memory. Consequently, the total bandwidth offered by the memory is often higher or much higher than the bandwidth that a single processor may request. In the end, if only one or relatively few processors access memory at the same time, the memory bandwidth may be underutilized, potentially to a severe extent. For example, the ratio between the memory bandwidth available to a single CPU core and the one available to the whole system (or the CPU complex, respectively) ranges between: 5.0% (or 11%) on an A57 core in the NXP i.MX 8QM platform, 13% (or 24%) on a Carmel core in the Nvidia Xavier AGX, 15% (or 35%) on a A53 core in the AMD Xilinx Zynq UltraScale+ [4].

Limited memory access: either high bandwidth, or high predictability

The other type of solutions follows a somewhat opposite approach, as it allows multiple processors to access memory in parallel [11, 18, 22]. In this case, slowdowns are controlled by imposing a *limit* on the maximum bandwidth at which each processor can access memory. This approach is effective as long as the sum of the per-processor bandwidths remains low compared to the total memory bandwidth. In this particular case, memory contention is negligible, and per-processor bandwidths add up linearly – the execution latencies of parallel tasks are nearly identical to when executed in isolation. Consequently, the only factor producing the slowdown is essentially the bandwidth limit itself. Therefore, in this case, the slowdown experienced by each processor can be conveniently controlled.

Conversely, operating at high utilization implies making memory work at or close to bandwidth saturation. However, in this regime, memory behavior becomes non-linear. The slowdown for a specific processor changes if other processor begin accessing memory and varies based on the types of memory accesses and bandwidth limits of the same processor and the competing ones. Several factors contribute to this behavior, such as varying conflicts on different memory banks and interconnect components, or contention affecting the cache hierarchy at the interconnect level. [4] Ultimately, slowdowns become complicated to predict and impossible to control through any static assignment of bandwidth limits. For these reasons, memory limitation mechanisms in real-time scenarios are typically employed conservatively, aiming to maintain memory bandwidth well below saturation. However, in such a configuration, memory bandwidth is once again underutilized, similar to the previous case.

Closing the gap with dynamic parallel access

How can we reconcile high predictability in system behavior with efficient memory bandwidth exploitation? In other words, how can we guarantee bounded slowdowns while fully utilizing memory bandwidth? In this paper, we propose a general solution to achieve such a goal. It builds on two main ingredients.

PREM task model. We adopt a PREM-like real-time task model, featuring a task set with per-task deadlines and a processor set where tasks are to be executed, and memory limits can be enforced. PREM-like tasks are identified by memory phases during which they execute contiguous memory accesses. In our model, we assume that memory phases can fall into two types: data prefetch, where only reads are executed, or data writeback, where only writes are performed. This is similar to other models such as the Logical Execution Time (LET) model of task execution, which distinguishes logical timing requirements from the actual physical platform execution. In the LET model, a task is sequential code with its own memory space and lacks internal synchronization points [8].

Dynamic memory policy. We assume to be given an *execution policy* that provides task allocation and scheduling – at any time, it assigns tasks to processors of interest. We do not introduce any further hypothesis on the execution policy, except for the fact it has to be memory-agnostic, i.e. allocation and scheduling choices must not depend on any memory concept like limits, contention or service times. We introduce a second policy, called the *memory policy*, which is responsible for the dynamic adjustment of the bandwidth limits of all active processors. This policy is a function of how many processors are accessing memory at the same time, and of which memory accesses they are performing between reads and writes. Limits are adjusted in such a way that the slowdowns experienced by each task are low enough to let the task still meet its deadlines, under the scheduling policy at hand.

The assumption regarding PREM-like tasks represents an important simplifying hypothesis as it enlarges the granularity of memory regulation decisions to a tractable size. Indeed, it costs time to detect a change in the memory access pattern of processors, compute new limits, and initiate the enforcement of these new limits. The highest time constant in play is the communication delay between the processors. For instance, in the case of a single CPU, which has lower delays compared to a heterogeneous system, we can estimate the delay to be in the order of 10 microseconds, assuming the communication is triggered by an inter-processor interrupt [15]. For this reason, our solution is feasible only for PREM-like tasks whose memory phases are longer than this base delay. For non-PREM tasks, it would be challenging or even impossible to determine the type of each generic memory access right before it happens and to dynamically adjust bandwidth limits for each instantaneous change in the type of memory accesses. Nevertheless, other dynamic approaches could be considered, such as measuring per task/processor bandwidths online and adjusting limits on the fly. However, such extensions are beyond the scope of this initial proposal.

Given the PREM task set and a system where delays make our solution feasible, the core challenge lies in computing the bandwidth limits. All potential combinations of parallel access patterns must be considered, and their number grows exponentially with the number of processors and available limit values. However, only a limited set of configurations that drive memory bandwidth close to saturation are of interest. The number of these configurations is significantly lower than the total number of possible combinations. Therefore, a critical feasibility aspect is defining an algorithm that discards all useless configurations, eliminating the need to store them all and potentially avoiding their evaluation altogether.

We remark that, by construction,we structured the problem to maintain orthogonality between the execution and memory policies. This provides complete freedom in defining the execution policy. Specifically, one can establish either a global or a partitioned task scheduling policy with regulated parallel accesses to memory. A partitioned scheme is likely the preferred option for an initial solution, as it is typically characterized by simpler analysis and implementation.

2 System Model

This work focuses on a multi-processor system featuring a shared last-level cache and shared memory. The proposed idea in this work does not hinge on any specific task model or scheduling policy. Nevertheless, for illustrative purposes, the following system model is employed to present the idea. Each processor is assigned a partitioned subset of tasks. Tasks within the system follow a sequence: they prefetch data in the Read-memory phase, perform computations in a single computation phase, and write back the results in the Write-memory phase. On each processor, tasks are scheduled based on a dynamic-priority, non-preemptive policy.

2.1 Processors

The main memory is shared among m identical processors, each assigned a distinct static priority. These priorities govern the allocated bandwidth for parallel access to the main memory. A detailed explanation of how these priorities can be utilized to govern the parallel memory accesses will be provided in Section 4. Processors are indexed following their priorities, with P_1 possessing the highest priority and P_m the lowest.

2.2 Tasks

This work considers a partitioned system where each task is statically assigned to a single processor. The assignment of the i-th task to the k-th processor is denoted by the notation $\tau_i \in P_k$. Each task, denoted as τ_i exhibits a dynamic behavior by releasing an infinite sequence of jobs sporadically. Each individual job within this sequence is subject to a specified minimum inter-arrival time denoted as T_i. Each job of τ_i must be executed and completed within a fixed time limit from its release, specified by D_i, the relative deadline. We employ a PREM-like task model to simplify governing the memory requests. This assumption is significant because it increases the granularity of memory policing decisions to a manageable level. Detecting changes in memory access patterns, calculating new limits, and enforcing these limits all take time. Due to this constraint, our solution is practical only for tasks resembling PREM, where the duration of memory phases exceeds the base delay. For non-PREM tasks, accurately predicting the type of each generic memory access just before it occurs and dynamically adjusting bandwidth limits for every instantaneous change in memory access type would be challenging, if not impossible. In this PREM-like model, each task consists of three phases: a read-memory phase, a computation phase, and a write-memory phase. During the read-memory phase, the task prefetches data from the main memory. In the computation phase, the task performs computations exclusively on the prefetched data without making any requests to access the main memory. The result is then written back to the main memory during the write-memory phase. In both memory phase, the tasks executes only memory accesses.

Tasks are scheduled on each processor according to a dynamic-priority non-preemptive scheduling policy and are indexed by priority, with τ_1 having the highest priority and τ_n the lowest, where n is the number of tasks allocated to the processor under study.

2.3 Memory

Memory functions as a globally shared resource, accessible to all processors with identical memory access latencies. In this model, the processors are symmetric, and they have the potential to saturate the bandwidth, meaning their cumulative demand may exceed the total memory bandwidth. Consequently, due to memory interferences, the duration extension of memory phases for each processor becomes unpredictable without regulation. Additionally, we assume that the order in which the memory controller serves memory requests simultaneously issued by different processors is unknown.

3 The Scheduling Policy

In this section, we present a partitioned memory-centric scheduling policy and illustrate it with an example. We will delve into the basic idea and the principal rules of the proposed policy. Our applied scheduling policy consists of two main components: the *execution policy* to distribute the task set among the processors and to schedule the execution order of each subset of the task set on each processor, and the *memory policy* for regulating the bandwidth to control access to the main memory. These two parts are explained as follows.

Execution Policy. Each task in the task set is statically assigned to a single processor, and no migration is allowed. The task set is first sorted based on their relative deadlines. Subsequently, tasks are assigned one by one to the processors according to their priority. Assuming there are n_T tasks in the task set and m processors in the system, the task with the closest relative deadline is assigned to the highest priority processor, denoted as P_1. The task with the second closest relative deadline is assigned to P_2, and this assignment continues until each processor is assigned a task. Once every processor has a task, the $m+1^{th}$ closest relative deadline task is assigned to P_1 again. This procedure repeats until all tasks in the task set are assigned to processors. On each processor, tasks run based on a non-preemptive Earliest Deadline First (EDF) algorithm. Considering that all the required data for the execution of each task is prefetched during the read-memory phase, using a preemptive scheduling policy may increase the response time of each task by prefetching the same data several times. Therefore, choosing a non-preemptive policy can be more efficient. Moreover, by applying the EDF, the system can enjoy the benefits of dynamic task priorities.

Memory Policy. All the memory requests are governed globally through bandwidth regulation. When a new memory request arrives, or an existing memory access finishes, depending on the number of parallel memory accesses, and the workload on the targeted processor and the interfering ones, the supervisor should check the corresponding cell in the table of regulation factors($\mathcal{RF}$) to identify the amount of bandwidth allocated to each active processor.

4 Implementing The Policy

4.1 Memory bandwidth regulation

4.1.1 Regulation mechanisms

Memory bandwidth limits are realized by *memory regulation mechanisms* that can be implemented in different ways – they can be either hardware assisted, or provided by software components, either external from, or internal to, the workload to be limited.

HW regulation. Hardware regulation is commercially available with Memory Bandwidth Allocation, part of the Resource Director Technology by Intel [9], that is mainly featured on higher end Xeon family processors. A similar technology is provided also by Arm with the Memory System Resource Partitioning and Monitoring (MPAM) set of IPs [2], but we are not aware of any commercially available chip including it. The common principle is that CPU cores can be assigned with a given quality of service for memory transactions, and that such limit is enforced by regulating the traffic originated from the last-level cache or the system-level cache.

External SW regulation. Software regulation can be conveniently offered by a component of the platform software. It can be the case of the operating system like in MemGuard [21], the hypervisor like in MinervaSys Jailhouse [6] or some system-resident firmware like in MemPol [22]. The underlying principle is the same – a processor is allotted a predefined maximum number of memory accesses (budget) to execute within a fixed period. Should the processor exhaust its budget before the period concludes, the regulation mechanism halts the processor until the period concludes, at which point the processor receives a new budget for the subsequent period.

Internal SW regulation. This method, also called Voluntary Throttling (VoLT), entails augmenting the code of the memory phases of the PREM tasks so to introduce a number of NOPs (No Operation instructions) periodically during its memory phase(s) [4]. The number of NOPs is externally configurable and provides the mean to regulate the throttling length (or frequency) of throttling. The code augmentation can be inserted at compile time, which is especially convenient when code PREMization is already automated [7].

4.1.2 Regulation factors

In order of us to abstract from the implementation details, we define the notion of regulation factor, that acts as a knob to adjust the bandwidth allocation for a process. A regulation factor of 0% indicates that no bandwidth is allocated for the service, while a factor of 100% implies that the entire available bandwidth is dedicated to the service.

▶ **Definition 1** (Regulation Factor (RF)). *For each processor configured with a memory limit, we define its regulation factor (RF) as the ratio between the* limited memory bandwidth *measured in isolation by performing back-to-back Reads (or Writes) operations, and the* (unlimited) memory bandwidth *measured in the same conditions while removing the memory limitation on the processor. For the sake of simplicity, we assume RFs to range among percentage integers between* 0 *(no bandwidth) to* 100 *(full bandwidth).*

We remark that the target measurement is performed in isolation because the attainable bandwidth is, in general, influenced by the number of processors operating in parallel, and the workload on both the affected processor and any interfering processors. Consequently, let us also stress that the actual bandwidth experienced by a processor is influenced, not entirely determined, by its regulation factor.

Observe that the regulation primitives greatly differ among the various memory regulation mechanisms. A concrete manner is thus needed to compute the function that maps any configuration of the mechanisms-specific knobs to its corresponding regulation factors. In most cases, only an experimental method enables to obtain a precise definition, as the one provided in Subsubsection 4.1.3 for VoLT.

4.1.3 Example: VOLT regulation factors

In a VoLT system the only available knob is the number of NOPs injected in the code. To compute the RFs, we need to experimentally find, for each workload, for each kind of processor, and for each regulation factor, the number of NOPs that produce the limited bandwidth of our interest. For instance, when the regulation factor is 10%, it means that the number of NOPs is such that if the processor executes this specific workload (Read or Write) in isolation, it receives 10% of the unlimited bandwidth. This table can be constructed experimentally following the algorithm outlined in Algorithm 1.

For each memory access type, the algorithm initializes two counters: N_{old} and N_{new}, representing the initial and current number of NOPs, respectively. Then, it sweeps through a range of RF values from 100 to 0 with a step of -10, representing increasing regulation levels. At each RF iteration, a $Match$ flag is set to false, indicating that a precise match

Algorithm 1 Construct the translating table between regulation factors and limited bandwidth values.

```
Input : Unlimited bandwidth values $UnB_R, UnB_W$ for Read and Write memory
        accesses
Output: Translating table $\mathcal{RF2BW}$ between regulation factors and actual
        bandwidth values:
1  foreach $TMR \in \{R, W\}$ do
2      $N_{old} = 1$
3      $N_{new} = 2$
4      for $RF = 100$ to $0$ with step $-10$ do
5          Match = False
6          while $!Match$ do
7              $BW_{old}$ = measure the bandwidth with $N_{old}$ NOPs
8              $BW_{new}$ = measure the bandwidth with $N_{new}$ NOPs
9              if $|BW_{new} - \frac{RF}{100} \times UnB_{TMR}| < |BW_{old} - \frac{RF}{100} \times UnB_{TMR}|$ then
10                 $N_{old}{+}{+}$
11                 $N_{new}{+}{+}$
12             else
13                 $N(TMR, RF) = N_{old}$
14                 Match = True
15 return $Result$
```

between RF and bandwidth has not yet been established. The algorithm then enters a loop that repeatedly measures bandwidth using two different NOP configurations. The first measurement (BW_{old}) reflects the current NOP count (N_{old}), while the second measurement (BW_{new}) employs N_{old} incremented by one ($N_{new} = N_{old} + 1$).

The algorithm compares the absolute differences between the measured bandwidth values and the expected bandwidth value for each RF, calculated by multiplying RF by the unlimited bandwidth for the corresponding memory access type (UnB_{TMR}). If the difference for the newer NOP setting ($|BW_{new} - \frac{RF}{100} \times UnB_{TMR}|$) is smaller than that for the older setting ($|BW_{old} - \frac{RF}{100} \times UnB_{TMR}|$), it suggests that the newer NOP configuration provides a more accurate bandwidth estimation for that particular RF. In such a case, the algorithm increments both counters ($N_{old}++$, $N_{new}++$), effectively refining the bandwidth estimation resolution. The algorithm continues iterating within this loop until the Match flag is set to true, indicating that the optimal match between RF and bandwidth has been identified. Upon reaching this point, the algorithm stores the corresponding NOP count (N_{old}) in the translating table $\mathcal{RF2BW}$ for the specified memory access type (TMR) and RF value. This process is repeated for both R and W memory access types to construct the complete translating table.

It's crucial to recognize that since the number of NOPs is limited to integer values, the bandwidth values derived from regulation factors are not entirely precise – they represent the closest approximation of the true bandwidth achievable with a specific regulation level.

4.2 Construct the table of slowdown measurements

The table of regulation factors ($\mathcal{RF}$) comprises factors for regulating memory accesses corresponding to each workload pattern. To fill in the cells of this table, a series of slowdown measurements must be conducted for every conceivable set of regulation factors on active processors. This enables us to deduce the factors that more effectively align with our timing constraints.

As these slowdown measurements merely pertain to the specifications of the hardware in use, rather than the actual task set, we optimize system performance by employing a separate table, designated as the *table of slowdown measurements ($\mathcal{SM}$)*, to store the recorded slowdown values for each scenario of memory accesses and a specific set of regulation factors on the active processors. The construction of the slowdown measurements table is a one-time necessity. Subsequently, based on the task set specifications, we can then select the set of suitable factors.

In the remainder of this section, we systematically introduce the algorithms for constructing the aforementioned tables, step by step. But first, we shall define some preliminary concepts.

4.2.1 Formal definitions

The set of notations used is briefly explained in the Table 1.

To determine the appropriate set of throttling factors for effective bandwidth regulation, it is essential to undertake a series of slowdown measurements encompassing all possible combinations of throttling factors for active processors. In this section, we aim to outline a systematic approach for conducting these measurements while avoiding redundant cases. But before presenting the algorithm a few concepts should be clarified.

Table 1 The table of notations.

Notation	Definition
$\mathbf{C}(j, \mathbf{S}_i, \mathbf{RF}(j))$	a generic configuration consists of a set of regulation factors,$\mathbf{RF}(j)$, its corresponding slowdown measurement$\mathbf{SD}(j, \mathbf{S}_i, \mathbf{RF}(j))$, and the actual bandwidth of processors, or $\mathbf{UB}$
j	number of processors accessing the memory in parallel
m	total number of processors in the system
$N(TMR, RF)$	number of NOPs to achieve a specific regulation factor
RF_k	the regulation factor assigned to the $k-th$ highest priority processor among those accessing the memory at the same time
RF	a generic regulation factor to regulate the bandwidth
$\mathbf{RF}(j)$	a generic vector of regulation factors used to measure the slowdown values in case of j parallel memory accesses following $\mathbf{S}_i$ scenario corresponding to s measured slowdown value
$\mathcal{RF2BW}$	the translating table between regulation factors and actual bandwidth values
$\mathbf{S}_i$	each scenario of parallel memory requests
$\mathcal{S}(j)$	set of all possible scenarios of parallel memory accesses for each value of j
$\mathbf{SD}(j, \mathbf{S}_i, \mathbf{RF}(j))$	the vector of measured slowdown values corresponding to j parallel memory accesses following $\mathbf{S}_i$ scenario using $\mathbf{RF}(j)$
$\mathcal{SM}$	table of slowdown measurements
$\mathbf{SM}[\mathbf{S}_i(j), \mathbf{C}(j, \mathbf{S}_i), \mathbf{RF}(j)]$	sub-table of slowdown measurements corresponding to j parallel memory accesses following $\mathbf{S}_i$ scenario
$\mathbf{UB}$	the vector of actual bandwidth of the processors
ε	a very small positive value

▶ **Definition 2** (Slowdown Measurement). *Processor slowdown measurement involves quantifying the reduction in the speed of a processor as it performs tasks. This measurement is typically expressed as a percentage decrease in processing speed compared to the processor's original performance. In the context of this discussion, slowdown measurements refer to the decrease in speed of processors when accessing the memory as a consequence of throttling the bandwidth.*

In practical scenarios, the memory phases of different processors may partially overlap. However, to consider the worst-case scenario, these measurements involve executing all conceivable combinations of memory phases in parallel continuously. In this scenario, the system encounters the most severe slowdown due to memory interference.

▶ **Definition 3** (Set of workload combinations($\mathcal{S}(j)$)). *For any given number j of parallel memory accesses, the set of all the scenarios of memory accesses, or in other words, the different patterns of Reads and Writes coming from different processors in parallel, is called the set of workload combinations and is denoted by $\mathcal{S}(j)$. Each scenario within this set is denoted by $\mathbf{S}_i$.*

For example, in the case of two parallel memory requests, there will be four distinct scenarios of memory accesses: R_hR_l, R_hW_l, W_hR_l, W_hW_l, where 'h' refers to the memory access of the higher-priority processor and 'l' to the lower-priority one. Therefore, $\mathcal{S}(2)$ will have, 2^2 elements. Following the same reasoning, in general $\mathcal{S}(j)$ includes 2^j elements.

▶ **Definition 4** (Configuration of parallel memory accesses($\mathbf{C}$)). *For any given number j of parallel memory accesses, and any scenario of memory accesses $\mathbf{S}_i$, a configuration of parallel memory accesses is a vector that concatenates a vector of regulation factors for the corresponding active processors $\mathbf{RF}$, its corresponding measured slowdown $\mathbf{SD}(j, \mathbf{S}_i, \mathbf{RF})$, and the vector of actual bandwidth of the active processors $\mathbf{UB}(j, \mathbf{S}_i, \mathbf{RF})$. Or*

$$\mathbf{C}(j, \mathbf{S}_i, \mathbf{RF}(j)) = [\mathbf{RF}(j); \mathbf{SD}(j, \mathbf{S}_i, \mathbf{RF}(j)); \mathbf{UB}(j, \mathbf{S}_i, \mathbf{RF}(j))]$$

▶ **Definition 5** (Slowdown measurements table($\mathcal{SM}$)). *Slowdown measurements table is a table of subtables, each denoted by $\mathbf{SM}[\mathbf{S}_i(j), \mathbf{C}(j, \mathbf{S}_i), \mathbf{RF}(j)]$, corresponding to each number of parallel memory accesses (j) and each scenario of memory access ($\mathbf{S}_i$). Each subtable is represented as a two-dimensional array, where the rows represent different scenarios of memory accesses($\mathbf{S}_i$), and the columns represent different valid configurations of parallel memory accesses.*

4.2.2 Slowdown interplay

When employing each set of regulation factors, the amount of slowdown experienced by each processor may vary depending on the specific combination of processors concurrently accessing memory, the overall number of active processors, and the type of memory access (read or write) on both the targeted processor and its competitors. Since this study operates under the assumption of a homogeneous system, the identity of the processors simultaneously requesting memory is irrelevant; the only relevant factors are their relative priorities, and their total number, represented by j.

4.2.3 Algorithm core ideas

The algorithm to construct the slowdown measurement table receives the number of processors in the system as the input and outputs the table of slowdown measurements. For each number of parallel memory requests and each scenario, $\mathbf{S}_i \in \mathcal{S}(j)$, the algorithm commences by measuring the slowdown values and the actual bandwidth of all the active processors for the full-throttling case. Subsequently, starting from the processor with the lowest priority and progressing to the higher-priority ones, the throttling factor of each processor is reduced by 10 percent, and the measurements are then repeated to track every possible configuration of parallel memory accesses.

The main objective is to maximize bandwidth utilization, therefore we need to maintain bandwidth as close to saturation as possible avoiding over-allocation.

Within the saturation zone, the relationship between utilized bandwidth and throttling factors is non-linear. Counter-intuitively, adjusting the throttling factor of one processor can lead to fluctuations in the actual bandwidth values of other processors. For instance, decreasing the throttling factor of one processor might increase the actual bandwidth of other processors. On the opposite end, in the underutilization zone, the bandwidth is out of saturation. Therefore, the behavior of the system becomes essentially linear. In this zone, decreasing the regulation factors of one processor will either decrease or maintain the bandwidth of that processor without affecting the bandwidth of other processors. Considering this behavior, we can develop a discarding technique to improve the efficiency of the algorithm for storing slowdown measurements. This technique involves retaining only configurations close to saturation.

4.2.4 Discarding configurations

There are two primary reasons for discarding a configuration: either when we have already identified a better configuration in terms of bandwidth utilization in the saturation zone, or when the current configuration results in under-utilization of the available bandwidth.

If reducing the regulation factor results in negligible changes to the measured slowdown, it implies that the previous configuration was saturating the bandwidth. Consequently, we can discard the previous configuration and retain the current one. Given the uniform decrease of the regulation factors, this comparison can be conducted for every two consecutive measurements to eliminate configurations that result in over-allocation of bandwidth. Conversely, if a regulation requires less bandwidth than an already existing configuration for the same workload combination, it indicates that this configuration does not provide us with a better solution and there is no need to retain it. Considering the gradual reduction of regulation factors, this can be examined by comparing the actual bandwidth of all the processors in two consecutive measurements. If none of the processors gain a better bandwidth, it means this configuration under-utilizes the bandwidth. Therefore, we should discard it.

4.2.5 Termination

By the following termination rule, we can determine whether continuing the measurement will yield beneficial results or if we can terminate it. If modifying a single regulation factor solely affects the bandwidth of the corresponding processor, while the bandwidth of all other processors remains unchanged, it indicates that there is no interference between memory accesses, and the system is operating in the under-utilization zone. Therefore, we can terminate the current loop and proceed to the next outer loop. This nested table structure provides a concise and organized representation of the system's slowdown values, enabling efficient retrieval and analysis of the impact of regulation factors.

4.3 Construct the table of regulation factors

Given that slowdown values are influenced by the selected regulation factors, we can control slowdown values by adjusting these factors. With a predefined maximum tolerable slowdown value, aligned with the timing constraints of the task set, we can choose regulation factors to facilitate parallel memory accesses. This concept can be implemented through a table, which provides the appropriate set of Regulation factors for each scenario of parallel memory accesses under predefined slowdown constraints.

▶ **Definition 6** (Table of Regulation factors ($\mathcal{RF}$)). *The table $\mathcal{RF}$ is structured as a collection of sub-tables. For each number of parallel memory accesses, denoted as j, there exists a sub-table. Within each sub-table, for every possible scenario of memory accesses, it encapsulates the set of Regulation factors for active processors, tailored to meet the slowdown constraints imposed by the task set.*

To present the algorithm to construct this table, we should clarify a few notations.

▶ **Definition 7** (Possible Configurations ($\mathcal{PC}$)). *The set of candidate configurations corresponding to j parallel memory accesses with workload combination $\mathbf{S}_i$ is shown by $\mathcal{PC}(j, \mathbf{S}_i)$.*

▶ **Definition 8** (Maximum Bandwidth Utilization ($\mathcal{MBU}$)). *The set of regulation factors' vectors with the same maximum bandwidth utilization by $\mathcal{MBU}(j, \mathbf{S}_i)$.*

This algorithm receives as inputs: the table of slowdown measurements($\mathcal{SM}$), and the set of maximum tolerable values for slowdowns in line with the timing constraints of the task set, denoted by $\mathcal{MTS}$. It provides as output the regulation factors table $\mathcal{RF}$. Alongside this algorithm, for each number of parallel memory accesses, first, the set of all possible scenarios of parallel memory requests, or $\mathcal{S}(j)$, is generated. Then for each scenario, all the configurations stored at sub-table $\mathbf{SM}[\mathbf{S}_i(j), \mathbf{C}(j, \mathbf{S}_i)]$ are compared to their corresponding maximum tolerable value in $\mathcal{MTS}$. If the measured slowdown value is smaller or slightly larger than the maximum tolerable value, the algorithm appends this configuration to $\mathcal{PC}(j, \mathbf{S}_i)$.

Algorithm 2 Construct the slowdown measurements table.

```
Input  : number $m$ of processors
Output : table $\mathcal{SM}$ of slowdown measurements
1  for $j = 2 : m$ do
2      Generate: $\mathcal{S}(j)$
3      foreach $\mathbf{S}_i \in \mathcal{S}(j)$ do
4          $\mathbf{RF}_{\text{old}} = [100, 100, \ldots, 100]$
5          Perform slowdown and used bandwidth measurements for unlimited
             bandwidth case $\mathbf{SD}(j, \mathbf{S}_i, \mathbf{RF}_{\text{old}})$, $\mathbf{UB}$
6          $\mathbf{SD}_{\text{old}} = \mathbf{SD}(j, \mathbf{S}_i, \mathbf{RF}_{\text{old}})$
7          $\mathbf{UB}_{\text{old}} = \mathbf{UB}$
8          for $RF_1 = 100$ to $0$ with step $-10$ do
9              for $RF_2 = RF_1$ to $0$ with step $-10$ do
10                 ...
11                 for $RF_{j-1} = RF_{j-2}$ to $0$ with step $-10$ do
12                     $UnderUtilization = \text{False}$
13                     $RF_j = RF_{j-1}$
14                     while $!UnderUtilization$ do
15                         $\mathbf{RF}_{\text{new}} = [RF_1, RF_2, \ldots, RF_j]$
16                         Perform measurements $\mathbf{SD}(j, \mathbf{S}_i, \mathbf{RF}_{\text{new}}), \mathbf{UB}$
17                         $\mathbf{SD}_{\text{new}} = \mathbf{SD}(j, \mathbf{S}_i, \mathbf{RF}_{\text{new}})$
18                         $\mathbf{UB}_{\text{new}} = \mathbf{UB}$
                           // over-utilization check:
19                         if $\mathbf{SD}_{new} \approx \mathbf{SD}_{old}$ then
20                             $\mathbf{C}(j, \mathbf{S}_i, \mathbf{RF}_{\text{old}}) = \mathbf{C}(j, \mathbf{S}_i, \mathbf{RF}_{\text{new}})$
21                             $\mathbf{RF}_{\text{old}} = \mathbf{RF}_{\text{new}}$
22                             $\mathbf{SD}_{\text{old}} = \mathbf{SD}_{\text{new}}$
23                             $\mathbf{UB}_{\text{old}} = \mathbf{UB}_{\text{new}}$
                           // non-optimal solution check:
24                         if $\mathbf{UB}_{new} \leq \mathbf{UB}_{old}$ then
25                             Discard the new configuration
                           // termination condition:
26                         if $(\mathbf{UB}_{new}[1 : j-1] \approx \mathbf{UB}_{old}[1 : j-1]$ && $\mathbf{UB}_{new}[j] \leq$
                             $\mathbf{UB}_{old}[j]) \parallel RF_j == 0$ then
27                             $UnderUtilization = \text{True}$
28                         else
29                             $RF_j -= 10$
30 return $Result$
```

Whenever $\mathcal{PC}(j, \mathbf{S}_i)$ is empty, the algorithm reports: **"Not enough bandwidth"**, which means we should suspend the memory access coming from the lowest priority processor to make sure this case will not happen. However, if $\mathcal{PC}(j, \mathbf{S}_i)$ is not empty, the algorithm looks for the configuration that yields the maximum used bandwidth(or **UB**). If this solution is not unique, among them, this algorithm picks the one with the maximum regulation factors.

Algorithm 3 Construct the table of regulation factors.

```
Input  : Number $m$ of processors, set $\mathcal{MTS}$ of maximum tolerable values for
         slowdowns in line with the timing constraints of the task set, table $\mathcal{SM}$ of
         slowdown measurements
Output : Table $\mathcal{RF}$ of regulation factors
1  for $j = 2 : m$ do
2      Generate: $\mathcal{S}(j)$
3      for $\forall\, \mathbf{S}_i \in \mathcal{S}(j)$ do
4          Move to sub-table $\mathbf{SM}[\mathbf{S}_i(j), \mathbf{C}(j, \mathbf{S}_i)]$ in $\mathcal{SM}$
5          for $\forall\, \mathbf{C} \in \mathbf{SM}[\mathbf{S}_i(j), \mathbf{C}(j, \mathbf{S}_i)]$ do
6              if $\mathbf{SD} \leq (\mathcal{MTS}(j, \mathbf{S}_i) + \varepsilon)$ then
7                  Append $\mathbf{C}$ to $\mathcal{PC}(j, \mathbf{S}_i)$
8          if $\mathcal{PC}(j, \mathbf{S}_i) \neq \emptyset$ then
9              $MBU(j, \mathbf{S}_i) = \max_{\mathbf{C} \in \mathcal{PC}(j, \mathbf{S}_i)} \mathbf{UB}$
10             $\mathcal{MBU}(j, \mathbf{S}_i) = \{\mathbf{C} \in \mathcal{PC}(j, \mathbf{S}_i) | \mathbf{UB} \approx MBU(j, \mathbf{S}_i)\}$
11             if $\mathcal{MBU}(j, \mathbf{S}_i) \neq \emptyset$ then
12                 $\mathcal{RF}(j, \mathbf{S}_i) = \max_{\mathbf{C} \in \mathcal{MBU}(j, \mathbf{S}_i)} \mathbf{RF}$
13         if $\mathcal{PC}(j, \mathbf{S}_i) = \emptyset$ then
14             Report: "Not enough bandwidth"
15 return Result
```

4.4 An illustrative example

To illustrate the policy, the following example can provide clarity. Let's consider a task set as described in the table below. All tasks in this set are activated synchronously. Our objective is to efficiently schedule this task set on a system with three processors. These tasks are prioritized based on their periods and scheduled accordingly. We also assume that no preemption is allowed. In table Table 2 the duration of the read memory phase, write memory phase, computation time, period(or minimum inter-arrival time), and the relative deadline of task τ_i are denoted by, rm_i, wm_i, c_i, T_i, and D_i, respectively. All the values are measured in microseconds:

Table 2 Task Set Characteristics.

τ_i	$rm_i(\times 10\mu s)$	$c_i(\times 10\mu s)$	$wm_i(\times 10\mu s)$	$T_i(\times 10\mu s)$	$D_i(\times 10\mu s)$
τ_1	3	4	2	20	10
τ_2	2	3	1	25	15
τ_3	2	2	1	30	20
τ_4	1	1	1	35	25
τ_5	1	4	1	40	30

Following the partitioning policy, τ_1, and τ_4 are assigned to P_1, τ_2, and τ_5 to P_2, and τ_3 is assigned to P_3. In Gantt charts, red, grey, and blue blocks represent the read-memory phase, the computation phase, and the write-memory phase, respectively. Initially, we assume no parallel memory access is allowed, limiting each processor to accessing memory one at a time according to their priorities. Tasks are partitioned using the same policy, and on each processor, tasks are executed using a non-preemptive Earliest Deadline First (EDF) scheduling algorithm.

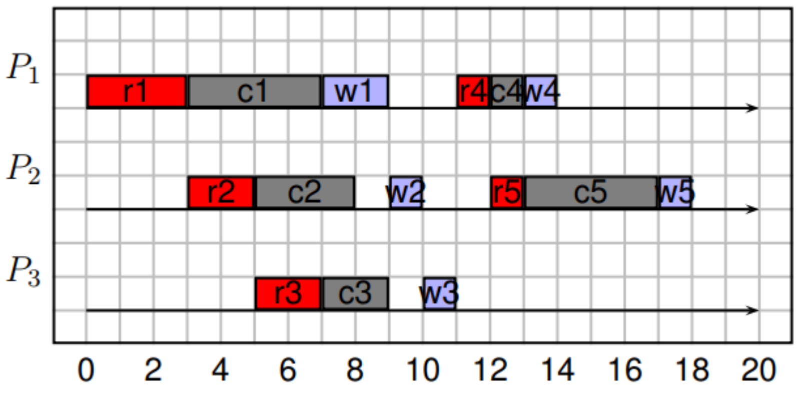

Figure 1 Partitioned scheduling algorithm, no parallel memory access.

Next, we aim to schedule the identical task set employing our policy. Broadly speaking, the asynchronous execution capability of the write memory phase renders it faster than the read memory phase. Therefore, we consider distinct values for unlimited bandwidth in read and write operations. As an illustrative numerical example, let's assume the system has an unlimited bandwidth of 2.5 GB/s for read-memory phases and 8 GB/s for write-memory phases. The regulation factors for two and three parallel memory accesses are outlined in Table 3 and Table 4, respectively. In practice, these tables must be filled following the algorithms.

Table 3 Table of regulation factors for two parallel memory accesses.

Workload pattern	Regulation factors for P_h	Regulation factors for P_l
RR	100	100
RW	90	70
WR	80	60
WW	60	40

Table 4 Table of regulation factors for three parallel memory accesses.

Workload pattern	Regulation factors for P_1	Regulation factors for P_2	Regulation factors for P_3
RRR	100	100	100
RRW	90	70	50
RWR	80	60	40
WRR	70	50	40
RWW	80	40	30
WRW	60	30	30
WWR	60	30	30
WWW	60	30	10

According to the tables of regulation factors, the Gann chart will be:

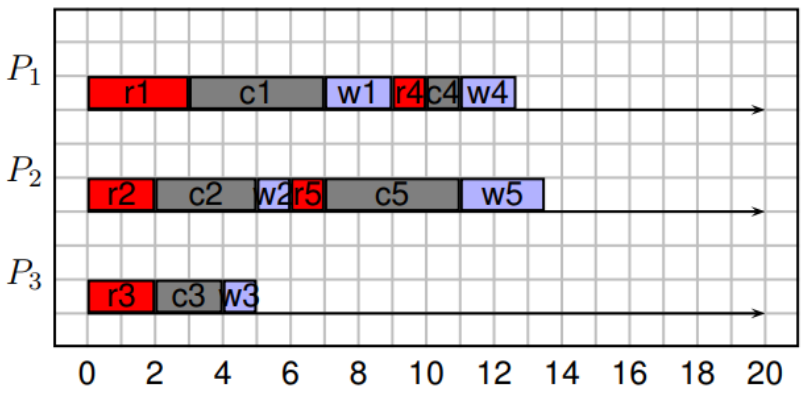

Figure 2 Partitioned scheduling algorithm, using HMB.

As illustrated in the example, three read-memory phases can be executed in parallel with negligible extension of the memory phase duration. However, when two write-memory phases access the memory in parallel (as is the case with w_4 and w_5), according to the table, 60% of the unlimited bandwidth of the processor will be allocated to the higher priority processor, and 40% to the lower priority one. Consequently, the duration of these memory phases will be extended accordingly. Nevertheless, upon comparing the Gantt charts, there is a remarkable improvement in the overall time-span of the task set following our proposed policy.

5 Related Works

The impact of memory contention in contemporary systems has been extensively explored in prior scientific literature. [10] Previous studies have focused on investigating the decline in Worst-Case Execution Time (WCET) for applications contending for memory, particularly in multi-core embedded systems [13]. Proposals for memory-bandwidth partitioning schemes aimed at ensuring temporal isolation have been introduced [12]. In [21], the authors introduced a memory bandwidth reservation system named MemGuard. This system was proposed, designed, and implemented with the primary aim of providing bandwidth reservation to ensure temporal isolation, and maximizing the utilization of the reserved bandwidth. In [14], the memory utilization is periodically sampled, While using standard MemGuard's interrupts – and associated overheads – to regulate cores and to trigger the sampling. While partitioning represents a straightforward and robust solution, it encounters challenges related to underutilizing the bandwidth. Moreover, it offers less refined control over task execution compared to the PREM approach. Similar challenges are observed in alternative hardware-level partitioning solutions and mechanisms for enforcing bandwidth allocation documented in existing literature [5, 18]. As an example in [22], known as MemPol, in introduced that operates a regulation mechanism from outside the cores, monitoring performance counters for the application core's activity in main memory at a microsecond scale. In contrast, our work adopts an internal mechanism to regulate the bandwidth offering a more flexible scheme to maximize bandwidth utilization. A substantial body of literature addresses the application of PREM model to multi-core systems [1, 3, 17, 19]. However, a primary limitation of these studies is their restriction to permitting only one memory access at a time, leading to bandwidth underutilization. In [20], the authors extended PREM by accommodating more than one task to access memory in parallel. Through experimentation, they demonstrated that the latency of main-memory accesses increases at a rate less than linear when multiple cores simultaneously access memory. Their model supports k parallel memory accesses, where k is a statically configurable number determined based on hardware specifications. The primary drawback of this model lies in its rigidity. As shown in [4], allowing k cores to utilize bandwidth without constraint, whether needed or not, may lead to bandwidth overutilization while selecting $k-1$ could result in bandwidth underutilization. Addressing this issue, the main advantage of our model lies in the dynamic allocation of bandwidth to processors based on workload patterns. This allows for the adaptive selection of the number of parallel memory accesses, optimizing resource utilization. Despite differences in scope, a comparable work to ours is [16], which introduces the Envelope-aWare Predictive model, abbreviated as E-WarP. It aims to provide both the technological foundations and theoretical bases for a workload-aware analysis of real-time systems.

6 Conclusion

6.1 Discussion

Contemporary homogeneous and heterogeneous computing systems attain enhanced performance levels through parallelization. However, the parallel execution of tasks introduces complex trade-offs between latency and throughput, particularly in the context of simultaneous accesses to shared memory. Numerous approaches aim to mitigate memory interference issues, with bandwidth regulation being popular. In the literature, various viable solutions for bandwidth provide basic mechanisms to address the latency of memory accesses. However, their primary drawbacks include the complexity of deployment and rigidity. It has been observed that existing solutions may lead to underutilization of the available bandwidth.

The challenge lies in the non-linear behavior of memory latency based on the number and type of concurrent accesses, which can fluctuate over time in an unpredictable manner. Past attempts to integrate memory regulation into scheduling solutions have, as a consequence, either failed to provide guarantees for real-time execution or led to significant underutilization of memory bandwidth.

In this paper, we introduce High Memory Bandwidth (HMB), a scheduling solution designed to guarantee bounded response times of real-time task sets through memory regulation while ensuring a high utilization of memory bandwidth. The intricate nature of both the problem and its potential solution necessitates a comprehensive approach, one that this preliminary work only begins to unfold. In this first step, we focus on the core mechanism of HMB, providing a detailed explanation of its inner workings and how it addresses the challenges posed by real-time task sets. Our goal is to lay a solid foundation for future research and development, paving the way toward a scheduling solution that seamlessly integrates the demands of real-time systems with the efficient utilization of memory resources.

The core concept of this work lies on the notion memory slowdown, a phenomenon that occurs when a processor's memory access performance is hindered by the concurrent memory access patterns of other processors. This slowdown, particularly during bursts of back-to-back memory accesses, can significantly impact the execution time of real-time tasks, potentially violating their timing constraints. To address this challenge, HMB employs a novel mechanism that dynamically allocates memory bandwidth among processors based on their priority levels. By carefully balancing the needs of high-priority processors, which typically have stricter timing requirements, HMB ensures that their memory accesses are prioritized, minimizing slowdown and maintaining their responsiveness.

HMB's efficiency stems from its ability to optimize memory bandwidth utilization while adhering to the priority-based allocation scheme. It continuously evaluates the system's memory access patterns and dynamically adjusts bandwidth caps to accommodate the demands of high-priority processors without compromising overall efficiency. This intricate balance enables HMB to achieve both bounded response times for real-time tasks and high memory bandwidth utilization, a remarkable feat in the context of resource-constrained real-time systems.

6.2 Further works

In this short work-in-progress paper, our primary focus has been on expounding the core concept of HMB and its underlying mechanism for bandwidth regulation. The next crucial step involves conducting a comprehensive series of experiments to rigorously validate the feasibility and evaluate the efficiency of this proposed mechanism in real-world scenarios.

During the implementation phase, potential overheads can arise at multiple levels, demanding careful consideration and optimization. At the hardware level, we must carefully evaluate the size of tables required to effectively implement HMB and ensure that data transfer speeds are sufficient to support the proposed bandwidth allocation scheme. Additionally, we need to analyze the overhead introduced by the algorithm and the dispatcher at the execution level. To ensure feasibility, it is imperative that the overall overhead remains lower than the shortest memory phase.

To objectively assess the effectiveness of HMB, we can compare its performance to similar works in this domain, such as [20] and [16]. By comparing against these established solutions, we can gain valuable insights into the relative strengths and weaknesses of HMB, paving the way for further refinements and improvements.

References

1 Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 285–296, 2015. `doi:10.1109/RTAS.2015.7108452`.

2 Arm. *MPAM Architecture Reference Supplement*, 2022. URL: `https://developer.arm.com/documentation/ddi0598/latest`.

3 Stanley Bak, Gang Yao, Rodolfo Pellizzoni, and Marco Caccamo. Memory-aware scheduling of multicore task sets for real-time systems. In *Proceedings of the 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '12, pages 300–309, USA, 2012. IEEE Computer Society. `doi:10.1109/RTCSA.2012.48`.

4 Gianluca Brilli, Roberto Cavicchioli, Marco Solieri, Paolo Valente, and Andrea Marongiu. Evaluating controlled memory request injection for efficient bandwidth utilization and predictable execution in heterogeneous socs. *ACM Trans. Embed. Comput. Syst.*, 22(1), December 2022. `doi:10.1145/3548773`.

5 Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core devices for safety-critical systems: A survey. *ACM Comput. Surv.*, 53(4), August 2020. `doi:10.1145/3398665`.

6 Jailhouse commmunity, Technical University of Munich, Università di Modena e Reggio Emilia, Boston University, and Minerva Systems SRL. MinervaSys Jailhouse, 2019–2023. URL: `https://gitlab.com/minervasys/public/jailhouse`.

7 Björn Forsberg, Marco Solieri, Marko Bertogna, Luca Benini, and Andrea Marongiu. The predictable execution model in practice: Compiling real applications for cots hardware. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5):1–25, 2021.

8 Arkadeb Ghosal. *A Hierarchical Coordination Language for Reliable Real-Time Tasks*. PhD thesis, EECS Department, University of California, Berkeley, January 2008. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-10.html`.

9 Intel. *Resource Director Technology Refrence Manual*, 2019.

10 Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesus Carretero. A survey of techniques for reducing interference in real-time applications on multicore platforms. *IEEE Access*, 10:21853–21882, 2022. `doi:10.1109/ACCESS.2022.3151891`.

11 Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118, 2014. URL: `https://api.semanticscholar.org/CorpusID:12573936`.

12 Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118, 2014. `doi:10.1109/ECRTS.2014.20`.

13 Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746, 2010. `doi:10.1109/DATE.2010.5456952`.

14 Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael Pressler, Arne Hamann, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Memory utilization-based dynamic bandwidth regulation for temporal isolation in multi-cores. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 133–145, 2022. `doi:10.1109/RTAS54340.2022.00019`.

15 Gero Schwäricke, Tomasz Kloda, Giovani Gracioli, Marko Bertogna, and Marco Caccamo. Fixed-priority memory-centric scheduler for cots-based multiprocessors. In *Euromicro Conference on Real-Time Systems*, 2020. URL: `https://api.semanticscholar.org/CorpusID:220275158`.

16 Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. E-warp: A system-wide framework for memory bandwidth profiling and management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 345–357. IEEE, 2020.

17 Muhammad R. Soliman and Rodolfo Pellizzoni. PREM-Based Optimal Task Segmentation Under Fixed Priority Scheduling. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:23, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECRTS.2019.4`.

18 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016. `doi:10.1109/RTAS.2016.7461361`.

19 Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48, November 2012. `doi:10.1007/s11241-012-9158-9`.

20 Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 65(9):2739–2751, 2016. `doi:10.1109/TC.2015.2500572`.

21 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2015.

22 Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. Mempol: Policing core memory bandwidth from outside of the cores. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–248. IEEE, 2023.