# The Multi-Resource Server for Predictable Execution on Multi-core Platforms

Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, Mikael Sjödin
Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
Email: rafia.inam@mdh.se

***Abstract*—In this paper we present an implementation and demonstration of the Multi-Resource Server (MRS) which enables predictable execution of real-time applications on multi-core platforms. The MRS provides temporal isolation both between tasks running on the same core, as well as, between tasks running on different cores. The latter could, without MRS, interfere with each other due to contention on a shared memory bus.**

**We demonstrate that MRS can be used to "encapsulate" legacy systems and to give them enough resources to fulfill their purpose. In our case study a legacy media-player is integrated with several resource-hungry tasks running at a different core. We show that without MRS the media-player starts to drop frames due to the interference from other tasks; while introduction of MRS alleviates this problem. Another part of our demonstration shows how traditional periodic real-time tasks can be kept schedulable even when tasks with high memory-demand are added to the system.**

***Index Terms*—hierarchical scheduling, CPU resource, memory resource, memory bandwidth, Linux, kernel, real-time systems.**

## I. Introduction

Using multi-cores for real-time applications presents many challenges. One such challenge is to achieve and maintain predictable execution of concurrent tasks that compete for both CPU- and memory-bandwidth resources. On uni-core platforms, the server-based scheduling approach successfully bounds the interference between the applications running [1], [2], [3]. However, this approach is limited to provisioning of the CPU resource only and it does not take the memory bandwidth problem into account, the latter problem often being inherited from when migrating software from a single-core to a multi-core architecture. In this paper we target statically partitioned multi-core real-time systems. For these systems we present the Multi-Resource Server (MRS) technology that schedules the two resources CPU- and memory-bandwidth, in order to achieve a predictable execution of embedded real-time systems.

In statically partitioned multi-core systems, concurrent tasks allocated to the same core interfere with each other by competing for CPU-bandwidth (we call this *local interference*), and concurrent tasks allocated to different cores interfere by competing for memory-bandwidth (we call this *global interference*). In addition to these sources of interference, tasks can also experience both local and global cache-pollution interference. If needed, e.g. for hard real-time systems, cache pollution can be relieved by cache-partitioning techniques like [4], [5], which are not within the scope of this paper. Thus, the implementation of the MRS presented here is suitable for soft real-time systems. However, if cache pollution can be avoided (e.g., by cache partitioning [5] or by disabling caches, or bounding caches by some static analysis technique [6], [7]), the schedulability analysis presented in [8] paves the way for using MRS also in hard real-time systems.

Additionally, we practically demonstrate the capability of MRS to maintain a predictable execution of a legacy soft real-time application. We show that MRS is not only useful when developing new systems; it is also useful to encapsulate and protect legacy applications, e.g., when performing a migration of applications from a single core to a multi-core platform. While our example uses a single task, the MRS allows for a complete subsystem with a set of tasks (and potentially its own scheduling algorithm) to be encapsulated in a *server* and then share an allocation of CPU- and memory-bandwidth resources. This is later demonstrated in a case-study using a synthetic setup.

We have presented the basic idea of the MRS approach in [8] where a theoretical analysis framework is provided to assess the composability of applications/subsystems. In this paper, we focus on the implementation of the server, and its evaluation using (1) a case study and (2) a synthetic experimental setup. Partitioned scheduling is considered in which servers and tasks are statically allocated to a specific core. The rationale for looking at statically partitioned systems is due to our industrial partners' preference.
The main contributions of this paper are:

- We present the first implementation of the MRS[1]. This implementation is made as a user-space library for Linux running on COTS hardware.
- We demonstrate how the MRS can be used to preserve the functionality of a legacy application when it is executed on a single core while another core executes tasks with adverse memory behavior.
- We demonstrate for a synthetic task-set how the MRS can be used to isolate tasks from each other to prevent adverse behavior of some tasks to negatively impact other tasks.
- We measure the overhead of memory related parts of the

This work is supported by the Knowledge Foundation (KK-Stiftelsen), via the research programme PPMSched.

[1]The MRS implementation is available as an open source project at http://www.idt.mdh.se/~MemSched/

MRS and we conclude that it is low.

**Paper Outline:** Section II explains system model, followed by Section III that describes the MRS concept in details. A brief overview of the software framework used to implement the MRS is presented in Section IV. Implementation details are covered in Section V. Section VI presents the evaluation setup and Section VII describes a case study using a soft real-time application. Synthetic evaluations are performed and results are analyzed in Section VIII. Section IX presents the related work, and finally, Section X concludes the paper.

## II. System model

In this section we present our target hardware platform, the system model that we use, and the assumptions that we follow.

### A. Architecture

In our work we assume the architecture to consist of a processor with a set of identical cores that all have uniform access to the main memory. Each core has a set of local resources; primarily a set of caches for instructions and data. The system has a set of resources that are shared amongst all cores; this will typically be the last-level cache, main-memory and the shared memory bus. Our architecture is industrially relevant and is the first step towards more advanced architectures.

We assume that a local cache miss is stalling, which means that whenever there is a miss in a local cache the core is stalling until the cache-line is fetched from memory. We focus on the shared memory bus and we assume that all accesses to the shared memory and the last-level cache go through the same bus, and that the bus serves one request at a time. It is worth noticing that any single-core could easily generate enough memory traffic to saturate the memory bus by executing memory intensive tasks.

### B. Server model

Our scheduling model for the multi-core platform can be viewed as a set of trees, with one parent node and many leaf nodes per core, as illustrated in Figure 1. The parent node is a *node scheduler* and leaf nodes are the servers. Each server has its own set of tasks that are scheduled by a *local scheduler*. The node scheduler is responsible for dispatching the servers according to their bandwidth reservations (which include both CPU- and memory-bandwidth). The local scheduler then schedules its task set according to a server-internal scheduling policy.

Each server $S_s$ is allocated a budget for CPU- and memory-bandwidth according to $\langle P_s, Q_s, M_s \rangle$, where $P_s$ is the period of the server, $Q_s$ is the amount of CPU-time allocated to the server each period, and $M_s$ is the number of allowed memory requests in each period. The CPU-bandwidth of a server is thus $Q_s/P_s$ and we assume that the total CPU-bandwidth for *each core* is not more than $100\%$. $M_s/P_s$ is the memory-bandwidth for $S_s$, and we assume that the total memory-bandwidth allocated to servers *on all cores* is not more than what can be served on the shared memory bus.

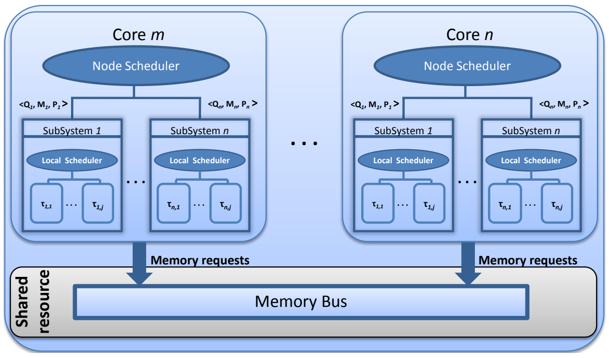

Fig. 1: The multi-resource server model

Server parameters can be obtained using analysis [8] or from domain expertise.

During run-time each server is associated with two dynamic attributes $q_s$ and $m_s$ which represent the amount of available CPU- and memory-budgets respectively. The implementation in this paper uses *Fixed Priority Pre-emptive Scheduling (FPPS)* policy for both node scheduling and server scheduling.

We assume that each server is assigned to one core and that its associated tasks will always execute only on that core i.e., we use the partitioned multiprocessor scheduling technique. The terms memory-bandwidth reservation and memory reservation are used interchangeably in the rest of the paper.

### C. Task model

We are considering a simple sporadic task model in which each task $\tau_i$ is represented as $\tau_i(T_i, C_i, D_i)$ where $T_i$ denotes the minimum inter-arrival time of task $\tau_i$ with Worst-Case Execution Time $WCET_i$ and deadline $D_i$, where $D_i \leq T_i$. Each task $\tau_i$ has a fixed priority $\rho_i$. During the execution of tasks, memory requests can be made arbitrary at any time which can cause cache misses, i.e., the model of memory requests of each task is not known in advance.

## III. The multi-resource server

The goal of the MRS is to provide temporal isolations through resource reservation approaches in the context of CPU bandwidth reservation [9] and memory bandwidth reservation [10]. The following subsections explain the MRS server and mechanisms used to manage memory budget of the server.

### A. The MRS mechanism

We explain the MRS using the following rules:

**Rule 1:** The MRS server is of periodic type, i.e., it replenishes both CPU- and memory-budgets to the maximum values periodically. At the beginning of each server period its dynamic attributes are set as $q_s = Q_s, m_s = M_s$.

**Rule 2:** In each core, a node scheduler is responsible to schedule all ready servers. A server is in the *ready state* if its remaining budgets are greater than zero, i.e. $q_s > 0$ and $m_s > 0$. The scheduled server applies its associated local scheduler to schedule its ready tasks.

**Rule 3:** The CPU resource that is used by a task will be decremented from its associated server's CPU dynamic attribute $q_s$, i.e., if a task $\tau_i$ executes $x$ time units then $q_s = q_s - x$.
**Rule 4:** The number of memory requests issued by each task will be decremented from its associated server memory dynamic attribute $m_s$, i.e., if a task $\tau_i$ issues $y$ requests then $m_s = m_s - y$.
**Rule 5:** A server is in a *suspended state* if any of its CPU- or memory-budget is depleted, i.e., if $m_s = 0$ or $q_s = 0$ then $m_s = q_s = 0$ and the server is suspended until the next server period. Thus, if any of the budgets is depleted then the other remaining budget will be discarded.
**Rule 6:** We use the idling periodic server strategy [11] for CPU reservation, i.e., if the scheduled server has remaining budget but there is no task ready then it simply idles away its budget until a task becomes ready or the budget depletes.

Note that the MRS follows exactly the rules of the idling server type except for the additional rules related to the memory requests. The memory part of the server behaves like a deferrable server [12] where the capacity of the server is consumed only whenever a memory request is made. The presented rules guarantee that a server only consumes its given CPU budget which limits its effect on the other servers that share the same core. The rules also guarantee that a server only consumes its give memory budget, which limits its effect on servers that are located in different cores. In addition and due to the interaction between the two different types of the budgets of the MRS server, the server may overrun its CPU budget. This may happen when a task in a server issues a memory request just before the CPU budget depletion. Since the core is stalling until the memory request is served then the server suspension will be delayed and an overrun can occur. The amount of overrun can be at most equal to the time to serve one memory request. In this paper we ignore such overruns since they are negligible compare to our clock resolution, but for hard real-time analysis the overruns are considered in the analysis of the MRS server [8].

We explain the execution of a MRS using a simple example of a subsystem that consists of five tasks $\tau_1, ..., \tau_5$ where tasks are ordered by their priorities in a descendent order i.e. $\tau_1$ has the highest priority and $\tau_5$ has the lowest priority. Figure 2 shows a possible execution scenario of tasks inside a server $S_s(P_s, Q_s, M_s)$. At the first budget period, we assume that $\tau_5$ is the only ready task and it issues a memory request and then waits until the request is served, and after a very small time $\tau_4$ is released and it preempts the execution of $\tau_5$ when the request is served (during core stalling, no task is allowed to execute) and it issues a memory request. Assume that $M_s = 2$, the memory budget depletes when the request is served and the remaining CPU budget is dropped. As a result the server execution will be suspended until the next budget period. The tasks $\tau_1$ and $\tau_2$ are activated within the first budget period but cannot execute due to the budget expiration of the server. These tasks will get a chance to execute in the next period when the server will be replenished to its full resources. In second budget period, the highest priority ready task $\tau_1$ executes.

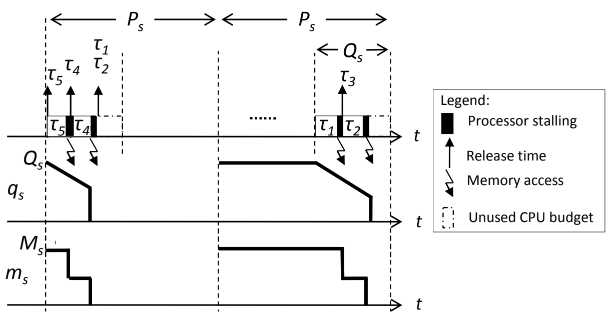

Fig. 2: An example illustrating the execution of the MRS

The implementation of the periodic server types considering the CPU-budget has been studied extensively (e.g. [13], [14], [15], [16], [17]). However, the main challenge is to add the memory budget and to consider the interactions between the two different types of budgets, more specifically Rule 4 and Rule 5. To implement these additional rules related to the memory part we need to track the memory requests issued by tasks within each server. We explain this in the following sections:

## B. Determining the consumed memory budget ($m_s$)

In many cases, a continuous determination and tracking of the consumed memory-bandwidth is very difficult without using a dedicated external hardware that monitors the memory bus. Since we target the use of standard hardware, we use a software-based technique similar to what has been used in [10], [18].

Most modern processors host a range of Performance Monitor Counters (PMCs) which can be used to infer the amount of resources consumed and are used to implement software-based memory throttling. These counters are hardware registers attached with the processor and they contain measures of various programmable events occurring in the processor. Different processor architectures provide different sets of performance counters which makes determination of the consumed memory budget $m_s$ more or less easy and accurate.

## C. Online monitoring and policing of $m_s$

To provide continuous online monitoring of the consumed memory-bandwidth, we need to continuously monitor counters and store their values. Performance counters are usually programmable and they can be configured to generate an interrupt at overflow, and hence they can be configured to count different types of events. For any given CPU architecture, its usage depends on issues like available events to count, number of available counters (often a small set of counters are available to be programmed to count various events), and the characteristics of the memory-bus.

In our work we implement servers that *enforce/police the consumed bandwidth.* To perform enforcement, an accurate and non-intrusive estimate of the bandwidth consumptions is very important. For policing purposes, using alarms could potentially provide the most accurate approach for accounting of the consumed bandwidth. One method could be to poll the performance monitor counters at each memory-request generation to evaluate $m_s$. Another method could be to allow an application to generate at most $y$ events before being policed by initializing the event accordingly (Rule 4), in this case an event would be generated after $y$ number of requests. This could obviously result in a situation where $m_s < 0$, which at a first glance would seem inappropriate (or, for hard real-time systems, even dangerous).

## IV. Software framework

The Linux operating system has been selected to be used for the implementation of the multi-resource server approach. The ExSched [15] framework has been used to support hard real-time behavior in Linux. Using this framework, real-time schedulers are developed without the need to patch or modify the main kernel itself. ExSched has been shown to suffer from some overheads. However, for our implementation of a research prototype for realizing the MRS these overheads are acceptable and in our implementation we only focus on the overheads generated by our new MRS functions, not the overhead incurred by the ExSched framework itself. Our selection of the framework is based on our study presented in [19].

The ExSched framework supports a user-space library and a loadable kernel module to control the CPU scheduler without modifying the underlying scheduler. The kernel module uses native Linux-kernel primitives and exports them as a simplified interface. Different plug-ins are provided that schedule tasks (user-space applications) using these interfaces. The flow of function calls of user-space applications to the Linux kernel through the core module are described in detail in [15]. New plug-ins can be developed by extending the scheduling policies, for example two hierarchical scheduler plug-ins (for fixed-priority scheduling and for EDF scheduling) and three multi-core scheduler plug-ins are presented in [15]. Since we use hierarchial scheduling for the MRS, we explain below how a two-level fixed-priority hierarchial scheduler is implemented in ExSched.

The uni-core hierarchical scheduler plug-in supports a two-level hierarchical scheduler and it schedules tasks within their servers [15]. All tasks are initially migrated to core 0 and they are assigned to their specific servers at system start using `job_init_plugin`. Tasks are executed periodically using the `rt_wait_for_period` API that internally calls the `job_complete_plugin` and `job_release_plugin` interfaces. Two main interrupt-handler functions to handle the server's activation and depletion activities are `server_release_handler` and `server_complete_handler`, respectively. They are triggered by server release and deplete events through timer activations. These functions release a server (along with its tasks) with its full CPU-budget to execute and suspend the server at its CPU-budget depletion (along with its tasks) respectively. A server ready queue and a server release queue are implemented using bitmaps (as Linux 2.6 native task ready-queue) to store the ready and depleted servers respectively.

## V. MRS implementation

The implementation details of the MRS in the context of partitioned multi-core scheduling are presented here. The hierarchical scheduler implemented in the ExSched framework manages the CPU-budget using FPPS at both levels of scheduling [15]. We extend the hierarchical scheduler to support the memory-budget and we extend it for multi-core platforms by implementing partitioned scheduling.

### A. MRS design for partitioned HSF

**Global structures:** To implement a partitioned multi-core hierarchical scheduler, an `SERVERS[]` array of `server_struct` type, and an `per_CPU[NR_RT_CPUS]` array of `perCPU_struct` type are used in the system globally as depicted in Figure 3. All other structures including timers and queues for servers are maintained per core. The `SERVERS[]` array holds all servers in the system. The only reason why the global variable `SERVERS[]` is maintained as an implementation choice is that a user API to create servers will be much easier and also to preserve the API scheme used by ExSched.

**`perCPU_struct` structure:** This structure contains core id, a timer and two queues, a `SERVER_READY_QUEUE` and a `SERVER_RELEASE_QUEUE`, to schedule servers on that core. Both server-queues are of bitmap extension type arrays of pointers. The timer is used to activate the server events on that core[2]. A server can be either in `SERVER_READY_QUEUE` or `SERVER_RELEASE_QUEUE` at any time, and is implied as ready (Rule 2) or inactive (Rule 5) respectively. Only one highest priority servers from the `SERVER_READY_QUEUE` executes at a time on each core. The `perCPU_struct` also contains an `severs[NR_OF_SERVERS]` array that is used for mapping servers to a specific core, and it stores all servers' IDs that are allocated to that core. This local `severs` array's index is mapped to the global `SERVERS[]` array's index for a faster access of server parameters during decisions making like comparing priority, updating remaining budget, referring to period etc.

**Server control block:** This contains all information needed by an MRS server in a `server_struct`, i.e. the `period` ($P_s$), `priority`, `budget` ($Q_s$), `remaining_budget` ($q_s$), `budget_expiration_time` and a `task_list` that points to tasks belonging to the server as presented in Figure 3. To execute the server on a particular core, the `CPU_id` variable is added to the `server_struct`. Further, the `mem_budget` ($M_s$) and `remain_mem_budget` ($m_s$) variables are added to monitor the memory-bandwidth consumption of MRS.

**Server release and complete handlers:** The two timer event handlers used to control activation and deactivation of servers in the multi-core HSF are `server_release_handler()` and `server_complete_handler()`, respectively. These handlers are triggered when previously setup timer events expire due to periodic activation, budget depletion or pre-emption by a higher

[2]More details on queues and timers can be found in code or in [15].

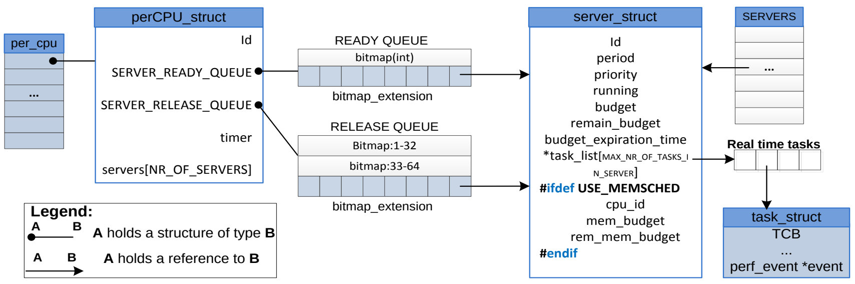

Fig. 3: MRS design for partitioned HSF

priority server. Multiple activities are performed in these handlers such as budget updating, task enqueuing/dequeuing, new timer setup etc. More details can be found in code.

### B. Implementing memory throttling by configuring and accessing counters

On multi-core processors each core usually contains its own set of hardware performance counters making it possible to account for memory events happening on a given processor. We implement the performance counters by using the `perf_event` interface of the Linux kernel. We use this interface within our implementation, and create and install performance counter events in the PMU[3]. When implementing the memory throttling, architecture specific performance-counters need to be used. Our current implementation is adapted for the Intel Core 2 architecture which has on-chip L1 and L2 caches. Thus, for this architecture it makes most sense to measure and throttle on the L2 cache-misses.

To account for the L2 misses incurred by a specific server, several reservation schemes in `perf_event` can be used, namely per-task or per-core assignment. Since multiple servers execute on each core, the per-core assignment scheme does not suit our problem. Therefore, we configure the counter for the per-task scheme, thereby accounting for events for only those tasks that belong to the server. In order to save and restore the counter registers upon a task context-switch, a `struct perf_event * event` structure is added to the task control block as shown in Figure 3.

A memory-requests counting event is created using `struct perf_event* init_counter (task_struct* task, int cpu)` API. Since tasks are statically allocated to a specific core, the event is also bound with the task and the core. It is configured to monitor hardware events (`PERF_TYPE_HARDWARE`) and to measure the L2 misses (`PERF_COUNT_HW_CACHE_MISSES`) in the kernel space. The counting event is created when a task calls its `task_run` function at its initialization and connects to its server.

### C. Online monitoring and policing of the memory budget

For online monitoring of the memory-budget, the performance counter is configured to cause an interrupt at an overflow. The `sample_period` is set to 1 to call the overflow handler `memory_overflow_handler()` at each memory-request. The memory budget of the event-generating tasks' server is decremented in the handler, i.e. $m_s--$; `remain_mem_budget` variable of each servers `server_struct` is used to monitor the consumption.

At memory-budget exhaustion (Rule 5), the `memory_server_complete_handler()` is called to enforce the server depletion. This handler works in the same manner as of `server_complete_handler()`, except that it is not an interrupt handler itself, rather it is called from the interrupt handler. It sets up the next activation time of the depleted server, it deactivates/dequeues the server along with its task set, and it activates/enqueues the next highest priority server with its tasks. If no server is ready at that time, then the idle tasks or other low priority tasks of Linux will execute.

## VI. Evaluation Setup

### A. Hardware and software platforms

All experiments are performed on an Intel core 2 CPU 6700 with two cores running at a frequency of 2.66 GHz having 32+32KB of local L1 instruction- and data-cache, and sharing an L2 cache of size 4MB. The frequency scaling is disabled to prevent the system from going into power-save modes and reducing its clock-frequency. With this setup the architecture is able to support a memory load of 133K memory-requests (counted as L2 cache-misses) per second.

We use Ubuntu 10.04.4 LTS with Linux kernel version 3.6.0-rt. The scheduler resolution (system tick) is set to $1ms$. The standard C library is used for programming and all programs are compiled using the gcc compiler.

### B. The behavior of synthetic tasks

Two different synthetic task-types are used in the case study and in the synthetic evaluations, namely *normal task*, and *memory intensive task*, their behavior is described here:

*The normal task* generates a relatively low number of requests per server period as compared to the memory intensive task. The task's code iterates a dynamic linked list consisting of a total of 140000 nodes (each node is of 8 bytes, and the size of the list is approximately 1MB) and it assigns an integer value to the single data item of the list. The WCET of the task is dependent on the selected number of iterations. In our investigation, the use of this dynamic linked list generates a good amount of reads and writes to

[3]Performance Monitoring Unit (PMU) in the Intel architecture where performance counters are implemented.

the memory. Some accesses goes to the cache (due to good locality of consecutively allocated memory blocks) but we also get a quite large amount of cache misses, resulting in memory requests on the shared bus.

*The memory intensive task* generates a very high number of requests per server period. The task iterates through the same kind of linked list as the normal task except that the number of nodes in the list is increased by 4 times. Consequently, the list size (list size is slightly greater than L2 cache size i.e. 4MB) becomes much bigger than the list size of a normal task. Further, the task is executed continuously (i.e. it never goes idle waiting for a new period) within a server, thus the task is only bounded by its server's reservation and it will execute as much as the server allows it to. Hence, this task will heavily affect other tasks' execution in the system due to its unbounded execution time and a very high memory-bandwidth usage.

## VII. Case study: Executing a legacy application

The purpose of this experiment is (1) to show that a legacy application that works well when executed on one core may fail to deliver its service if applications on other cores consume too much resources, and (2) to show that if applications resource utilization are bounded with the MRS, then we can protect the legacy application and allow it to deliver its service.

We use a soft real-time legacy application: *mplayer*[4], that decodes and plays an audio/video file and it requires continuous access to memory to fetch and process video frames.

| Server | Core | Priority | Period | CPU-budget |
|---|---|---|---|---|
| `mplayer` | 0 | `High` | 15 | 10 |
| `Server0` | 1 | `High` | 80 | 12 |
| `Server1` | 1 | `Low` | 80 | 12 |

TABLE I: The servers' specification for the case study.

mplayer demands a high amount of memory-bandwidth to display the video at an acceptable rate. Further, the timing is important for mplayer, otherwise it starts dropping video frames affecting the quality of service. We execute mplayer as a task within a server on core 0 that is bounded only by its server's CPU reservation. On the other core, we execute synthetic tasks within servers. Note that mplayer server is not throttled for memory-bandwidth in all experiments of the case study. We throttle servers executing on core 1 to observe the effect of bounding memory-bandwidth usage of core 1 on mplayer which is executing on core 0.

For the case study we have executed a high-definition HD video, i.e. a trailer of Avatar ([H264] 1920x800 24bpp) of a total of 260 seconds duration. To assess the performance of the quality of service delivered by mplayer, we use the *number of dropped frames* as our benchmark. The servers used for the case study are presented in Table I. The case study is performed in three steps, presented below. In these steps, the servers' priority, period, and CPU-budget remain the same as given in Table I, while the tasks' behaviour and the memory-budget vary in different experiments. Server and task period, CPU-budget, and Worst Case Execution Time (WCET) values are presented in $ms$, while the memory-budget is provided as a number of memory-requests in Tables I and II. The details for these steps are presented here:

[4]http://www.mplayerhq.hu

*1) :* we executed mplayer with our example video-file as a stand-alone application on core 0 to find its normal execution behavior having all resources available. We found that it drops $0\%$ of the frames while playing the video at a rate of $25fps$. These measures are later compared when mplayer is executed along with other MRSs in the system and the resources are shared among all applications/subsystems.

*2) :* we inserted two MRSs on core 1, each executing two tasks as given in Table II. Note that a higher number means a higher priority for tasks. Without memory reservation on MRSs, the mplayer dropped $1\%$ of the frames due to the global interference. However, mplayer executed with $0\%$ dropped frames when the MRSs on core 1 are throttled with a memory-budget of 1100. Hence, using MRSs, mplayer can be executed with desired results, which was not possible without MRS.

| Task | Server | Priority | Period | WCET |
|---|---|---|---|---|
| `Task1` | `Server0` | 98 | 160 | 10 |
| `Task2` | `Server0` | 97 | 160 | 14 |
| `Task3` | `Server1` | 98 | 200 | 8 |
| `Task4` | `Server1` | 97 | 200 | 8 |

TABLE II: Tasks properties and their assignment to servers.

*3) :* we introduced heavier memory-traffic by executing two memory intensive tasks, where each server on core 1 is executing one task. As mentioned previously, both tasks execute continuously, bounded by their server's CPU-budgets respectively, and produce a heavy memory traffic.

Executing the system without memory reservation on MRSs produce a bad effect due to a global interference on mplayer by dropping $5\%$ of the frames. This effect is significantly reduced by throttling two MRSs on core 1: with a memory-budget of 750 requests, the dropped frames decreased to $3\%$; and with a memory-budget of 200, the dropped frames decreased to $0.3\%$ (only 17 frames dropped from a total of 5013 frames). Hence, using MRSs, mplayer can be executed with limited and acceptable effect on its performance, which was not possible without MRS. This case study shows that a predictable execution of a legacy uni-core application can be achieved on a multi-core platform by providing both temporal- and memory-bandwidth isolations through the usage of MRSs.

When running the memory intensive task-behavior on core 1 we see a slight decrease in the performance of mplayer compared to the normal task-behavior. While we have not investigated the reason for this decrease in detail, we hypothesize that the reason is related to increased cache-pollution in the shared L2 cache – making the mplayer experience more cache-misses and thus performing slightly worse. In the next section we show that cache-pollution *is* an issue that matters and that it can cause temporal interference among tasks.

## VIII. Synthetic evaluation – Results and analysis

Here, we measure performance overheads of the implementation and we evaluate the timing isolation of the MRS using a set of synthetic tasks.

### A. *Performance assessments*

We present the overheads for memory related functionality of the MRS. The first measured overhead is of executing the Performance Monitor Counters (PMC) and it is negligible as it only writes to a register of a core. The overhead of the interrupt function to handle overflow `memory_interrupt()` is $56ns$ (nano seconds) on average. This interrupt is called at each memory-request. Since the architecture can support a maximum of 133K memory-requests per second, this means in worst case $0.7\%$ overhead for our interrupts.

Other overhead measures are the time required to execute (1) the `server_release_handler()` function that activates servers and its tasks at the server's activation time, (2) the `server_complete_handler()` that suspends servers and its tasks at server's CPU-budget depletion, and finally (3) the `memory_server_complete_handler()` that suspends servers and its tasks at server's memory-budget depletion. Two scenarios are accounted for each of these functions: first, the function is called when *no other active server* was on the core (the idle task was executing) and a server context-switch will occur to execute the newly released server; and second, another *active server* was executing on the core, in this case a server context-switch may occur to execute the newly released server depending upon the server's priority. The overhead of a server context-switch is included within the measures.

The system is executed for 5 minutes and overhead measures are extracted for each scenario as presented in Table III. The *Count* column in the table represents the total number of times that a particular scenario executed and then average, minimum, maximum, and standard deviation on these values are calculated and presented. All values are given in microseconds ($\mu$s). It is obvious from the table that overheads are very low, i.e no more than $0.68\%$ for all functions for our experiments. The total overheads of the system are high for the underlying ExSched framework due to having a kernel modification-free solution and these overheads are presented in [15].

### B. *Synthetic experiments*

Synthetic experiments are performed by executing a schedulable example consisting of five servers along with their task sets on both cores for 10 seconds. Two servers, i.e. `Server0` and `Server1` are executed on core 0, while all other servers are executed on core 1. The servers' timing properties and their assignment to the CPU-core is given in Table IV. Tasks' properties and their assignment to their corresponding server is given in Table V. To execute our tasks before the Linux tasks and to avoid task pre-emptions due to other Linux tasks, we have assigned the highest priority values to tasks.

All synthetic experiments are performed in two steps: first executing all servers and tasks without memory reservation using a simple idling periodic server; and then executing using memory reservation as the MRS. The number of missed deadlines for all tasks are measured for both steps to examine the effect of global interference and to reveal the memory reservation and performance isolation properties of the MRS.

| Server | Core | Priority | Period | CPU-budget | Memory-budget |
|---|---|---|---|---|---|
| `Server0` | 0 | `High` | 24 | 8 | 650 |
| `Server1` | 0 | `Low` | 40 | 16 | 750 |
| `Server2` | 1 | `Medium` | 40 | 8 | 900 |
| `Server3` | 1 | `High` | 80 | 12 | 1100 |
| `Server4` | 1 | `Low` | 80 | 12 | 1100 |

TABLE IV: The servers' specification to test the behaviors.

| Task | Server | Priority | Period | WCET |
|---|---|---|---|---|
| `Task1` | `Server0` | 98 | 40 | 2 |
| `Task2` | `Server0` | 97 | 48 | 4 |
| `Task3` | `Server1` | 98 | 60 | 8 |
| `Task4` | `Server2` | 98 | 60 | 4 |
| `Task5` | `Server2` | 97 | 160 | 10 |
| `Task6` | `Server3` | 97 | 160 | 14 |
| `Task7` | `Server4` | 98 | 200 | 8 |
| `Task8` | `Server4` | 97 | 200 | 8 |

TABLE V: Tasks properties and their assignment to servers.

*1) Experiment 1: Memory-bandwidth reservation:* This experiment is performed to illustrate the memory-bandwidth reservation of MRS in the context of a schedulable system by means of a trace of execution and by calculating the number of missed deadlines for all tasks. To fully utilize the CPU- and memory-resources, we execute the server and task sets described in Section VIII-B on both cores using only normal tasks in the servers. Each experiment is executed in two steps and the total sum of missed deadlines for all tasks is measured. Since normal tasks are low memory-intensive and they require a low number of resources, they get a good chance to execute and thereby never miss their deadlines.

The visualization of the execution for the MRS on core 0 is presented in Figure 4. The execution trace of core 1 shows the same behavior; we omit it due to limitation of space in the paper. In the diagram, the horizontal axis represents the time in $ms$ starting from 0. In the task's visualization, the arrow represents task arrival, a gray rectangle means task execution, a white rectangle represents either a local pre-emption by another task in the same server or a global pre-emption due to its server's budget depletion or its server's pre-emption by a higher priority server. In the server's visualization, the numbers along the vertical axis are the server's CPU-capacity and the number along the diagonal line represents the memory-capacity (or the number of requests made by the server) during the period. The diagonal line represents the server execution, the vertical line shows the server depletion due to memory-budget, while the horizontal line represents either the waiting time for the next activation (when the budget has depleted) or the waiting for its turn to execute (when some other higher priority server is executing). There is one idle task per core that executes only when no task is ready on the core.

Note that it is clear from the diagram that all servers and

| Scenarios | server_release_handler() | | | | | server_complete_handler() | | | | | memory_server_complete_handler() | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Count | Avg. | Min. | Max. | St. Dev. | Count | Avg. | Min. | Max. | St. Dev. | Count | Avg. | Min. | Max. | St. Dev. |
| No other active server | 7443 | 5.9758 | 2 | 17 | 1.9047 | 6025 | 6.0211 | 2 | 21 | 2.8607 | 1239 | 5.5343 | 4 | 7 | 0.5180 |
| Another active server | 2478 | 7.3010 | 5 | 15 | 0.7192 | 11125 | 6.895 | 3 | 17 | 1.3890 | 1250 | 5.529 | 4 | 12 | 0.7557 |

TABLE III: Overhead measures for the memory related functionality of the MRS.

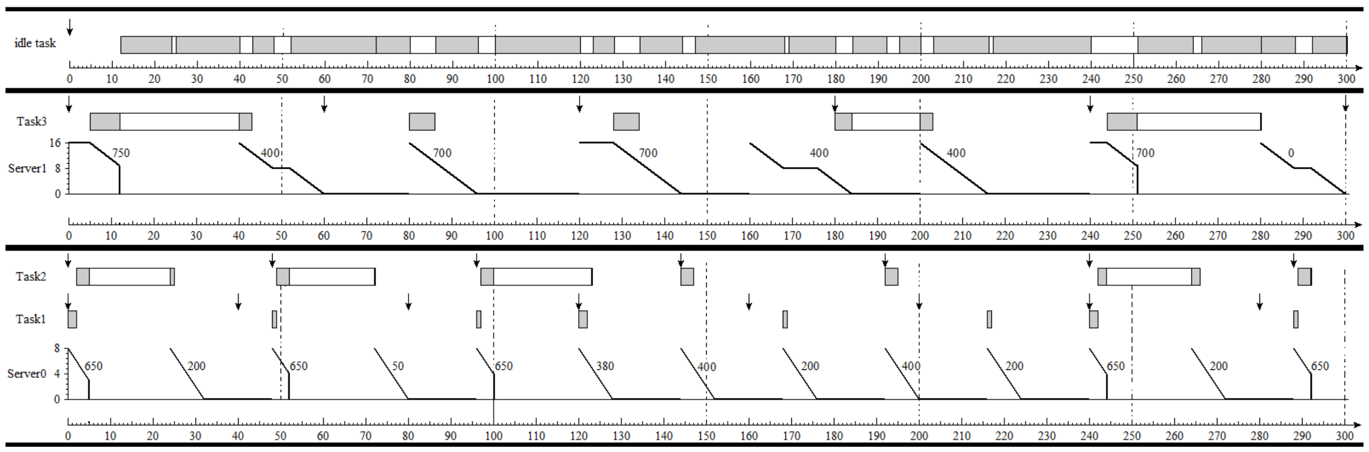

Fig. 4: Memory-bandwidth reservation of MRS: an execution trace of core 0 - using the normal tasks

tasks execute smoothly because of enough available resources. At the start of the system, both servers deplete due to their memory-budget exhaustion since the caches are empty and the system fetches a lot of data during this time to properly start the execution. This effect has been observed in all experiments.

*2) Experiment 2: Performance isolation effect of memory reservation:* This experiment is performed to illustrate the memory-isolation effect among the MRSs due to memory bandwidth reservation, even during the overload situation. For example, if one MRS is overloaded and its tasks miss-behave, produce a large number of memory requests, and fill-up the memory-bandwidth, it should not affect the execution of other MRSs in the system.

For this purpose, all servers execute the normal tasks except `Server1` that executes the memory intensive task as `Task3` for longer duration and produces an increased number of memory-requests. The experiment is executed without- and with memory reservation steps and traces of execution for both steps for core 0 are presented in Figures 5 and 6 respectively. The total sum of deadline misses (`No.of DMisses`) during the total number of task's activation (`Tot. Activations`) by all tasks is also measured and is presented in Table VI.

`Server1` is over-flooding the system with its memory-bandwidth usage thereby highly affecting the execution of all other tasks in the system when executed without memory-throttling as obvious from looking at Figure 5 and from the number of deadline-misses outlined in Table VI. Tasks of other servers miss their deadlines due to the reduced availability of memory-bandwidth. Both the local and the global interference has been observed here. However, when the same setup is executed by enabling memory reservation in the MRS, `Server1` gets bounded by its memory-budget and its overloaded memory-bandwidth usage keeps on reducing its affect on the execution behavior of other servers and tasks in the system, and finally at the memory-budget of 100 requests per period, the number of deadline-misses become 0 for all normal tasks as obvious from the forth column of Table VI.

Note that when executed without memory reservation, the execution of tasks of `Server0` becomes unpredictable. Not only the tasks' executions times are increased a lot but they are also generating a varying number of memory-requests, as obvious from Figure 5. Due to space reasons, the idle task execution is removed from the figure. Since the memory-bandwidth requirement from normal tasks is not as high as from memory intensive task, we consider that it is the bad effect of not only the high bandwidth usage but also of the cache pollution from `Task3`. However, we observe from the detailed execution trace of Figure 6 with memory reservation that the execution times of the normal tasks are slightly increased in very few periods. Mostly the trace of `Server0` and its tasks resemble the trace of experiment 1 with normal tasks in Figure 4. This could be due to the cache pollution effect.The use of the MRS highly reduces the cache pollution by limiting the high-demanding bandwidth server, but it could not delete it completely.

To further investigate the effect of `Server1` on other servers in the system and to confirm that the bad-effect on tasks' executions is only due to this server, we perform a third step by reserving all servers for memory except `Server1` and we measured the deadline miss count. As it is clear from the sixth column of Table VI, tasks suffer from the global interference and cache pollution and they miss their deadlines due to the un-bounded amount of memory requests from `Server1`.

*3) Experiment 3: Reducing cache pollution by memory reservation:* To further investigate the cache pollution effect and its reduction due to the memory reservation, we execute a *cache polluting task* in `Server1`. All other servers and tasks

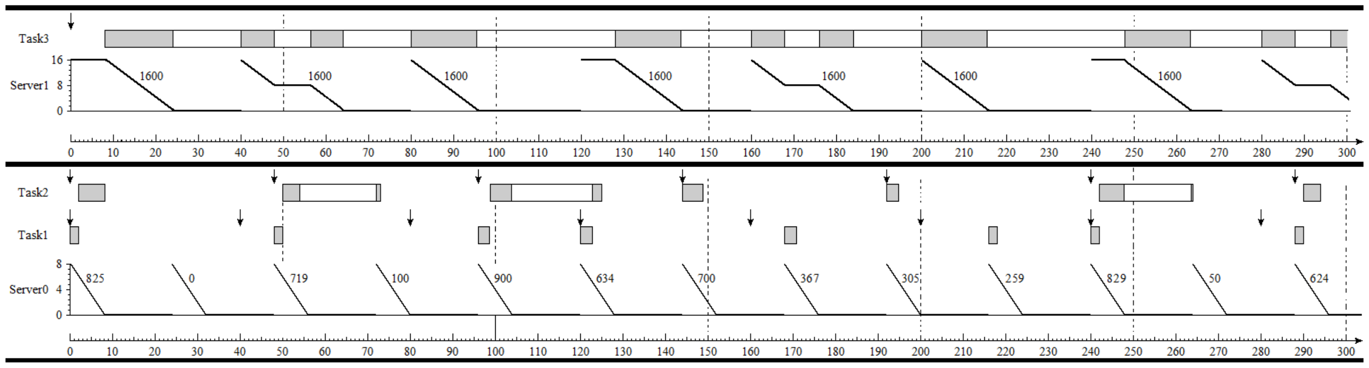

Fig. 5: Execution trace without memory reservation on Core 0 - using the memory intensive tasks

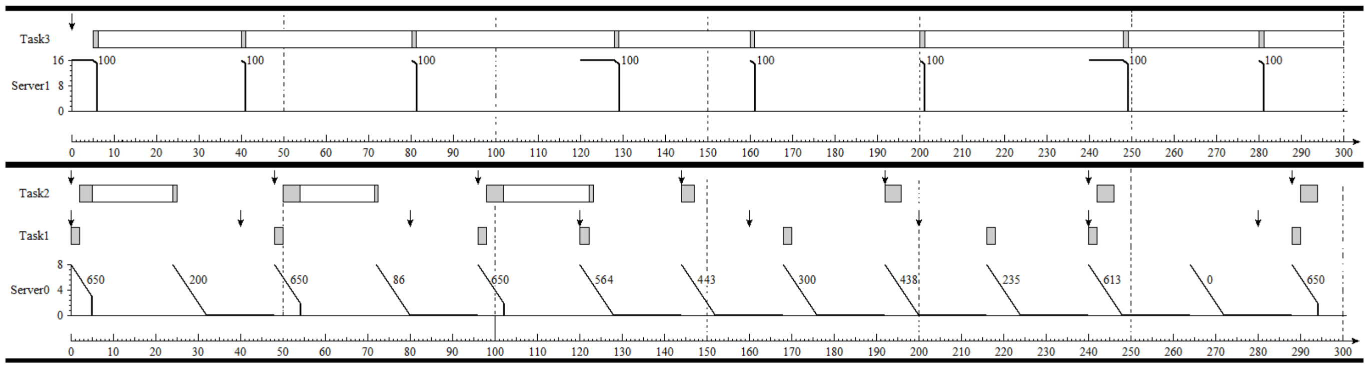

Fig. 6: Trace showing temporal- and memory-isolation among MRSs - using the memory intensive tasks

| Tasks | Without memory reservation | | With memory reservation | | throttling all servers except `server1` | |
|---|---|---|---|---|---|---|
| | `No.of DMisses` | `Tot. Activations` | `No.of DMisses` | `Tot. Activations` | `No.of DMisses` | `Tot. Activations` |
| `Task1` | 0 | 249 | 0 | 236 | 0 | 238 |
| `Task2` | 1 | 206 | 0 | 197 | 0 | 198 |
| `Task4` | 0 | 166 | 0 | 157 | 3 | 155 |
| `Task5` | 22 | 35 | 0 | 59 | 24 | 29 |
| `Task6` | 2 | 60 | 0 | 59 | 24 | 34 |
| `Task7` | 0 | 50 | 0 | 47 | 0 | 48 |
| `Task8` | 1 | 48 | 0 | 47 | 0 | 47 |

TABLE VI: Comparison of deadline misses by tasks to evaluate the behavior of memory-throttling.

have the same specifications and they execute the same code as presented in Experiment 1.

The *cache polluting task* is designed to examine the cache pollution effects and it is executed continuously in a server like the memory intensive task. However, its code is modified in two ways: first the size of the linked list is now multiplied by 10 to make the cache polluted; secondly, it dynamically creates nodes for two linked lists, reads the data from the first list and writes to the second list, and then deletes the nodes. Note that the caches have limited effect in this case of extreme cache pollution since the data size is much bigger than the cache size and additionally it is constantly changing over a chunk of memory due to allocations and de-allocations of memory in the same iteration. Therefore, this task represents the worst case of memory access pattern.

We observed that without memory-throttling, all tasks miss their deadlines many times as expected. However when we executed the experiment using MRSs with memory-throttling, there were always either two or three different tasks missing their deadlines once per execution, hence either two or three deadlines were always missed (due to space reasons we are not presenting all the data here). This experiment shows that our solution has the potential to reduce the cache pollution problem, however not solving it completely.

## IX. Related work

The problem of contention of shared resources has gained a significant importance in the context of multi-core embedded systems. CPU time partitioning is one technique to provide predictable execution on unicore platforms. In avionics, ARINC-653 is used as a platform to implement partitioned software with emphasis on predictability and safety-critical issues [20]. It provides fully deterministic top-level Time Division Multiple Access (TDMA) based schedule. A kernel-level implementation to support partitioning and hierarchical scheduling in ARINC-653 for Linux is provided in [21].

Some highly predictable TDMA based techniques are used to access the shared resources (memory bus arbitration) using a multiprocessor systems-on-chip (SoC) architecture. Rosen et al. [22] measured the effects of cache misses on the shared bus traffic where the memory accesses are confined at the beginning and at the end of the tasks. Later Schranzhofer et al. [23] relaxed the assumption of fixed positions for the bus access by arbitrating the shared bus. TDMA arbitration techniques eliminate the interference of other tasks due to accessing shared resources through isolation; however, they are limited in the usage of only a specified hardware. Akesson et al. [24] proposed a two-step approach to share a predictable SDRAM memory controller for real-time systems. This is a key component in CoMPSoC [25]. Stuijk et al. [26] used Synchronous Dataflow Graphs (SDFG) for allocating resources on a heterogeneous multi-processor system and provided throughput guarantees on multiprocessor systems-on-chip (MP-SoC). Zimmer et al. [27] used constraint programming to optimally maps tasks on network-on-chip (NoC), developed a TDMA-like approach to ensure separation of analysis for communication messages on NoC and then designed/implemented a heuristic-based solver to solve message-based NoC contention.

All these approaches are based on special hardware architectures. Our approach, however, uses COTS hardware and it is software based using performance counters which are available in almost all processors.

Pellizzoni et al. [28] initially proposed the division of tasks into superblock sets by managing most of the memory requests either at the start or at the end of the execution blocks. This idea of superblocks was later used in TDMA arbitration [23]. Bak et al. presented a memory aware scheduling for multi-core systems in [29]. They use PRedictable Execution Model (PREM) [30] compliant task sets for their simulation-based evaluations. However, PREM requires modifications in the existing code, hence this approach is not compliant with our goal to execute legacy systems on the multi-core platform.

Some approaches to WCET analysis are emerging which analyze memory-bus contention, e.g. [31]. However, WCET-approaches do not tackle system wide issues and do not give any direct support to provide isolation between subsystems.

Schliecker et al. [6] have presented a method to bound the shared resource load by computing the maximum number of consecutive cache misses generated during a specified time interval. The joint bound is presented for a set of tasks executing on the same core covering the effects of both intrinsic and pre-emption related cache misses. A tighter upper bound on the number of requests is presented by Dasari et al. [7] where they solve the problem of interleaving cache effects by using non-preemptive task scheduling. They have used PMCs in the Intel platform running the VxWorks operating system to measure the number of requests that can be issued by a task. However, these works lack the consideration of shared memory-bandwidth and the use of memory servers to limit the access to memory-bandwidth.

Recently a server-based approach to bound the memory load of low priority non-critical tasks executing on non-critical cores was presented by Yun et al. in [10] for an Intel architecture running Linux. In their model, one memory server is implemented on each non-critical core to limit memory requests generated by tasks that are located on that core. Hence the interference from other non-critical cores on the critical core is bounded. The servers are implemented on Linux using cgroups in [10]. This approach might not be suitable for those real systems that may contain more than one critical application. In addition, using one memory server in each non-critical core will degrade the performance of all applications in that core even if the core contains only one memory intensive task. This work has been extended in [18] by using a memory reclaiming technique when a core is not fully utilizing its allocated memory budget, and is implemented as a dynamic loadable Linux kernel module with some small modifications in the main kernel.

We propose a more general approach by implementing the MRS that handles both time and memory aspects reserved resources. Multiple subsystem/applications can share one core through multiple MRS's. Our memory throttling mechanism is proposed per server level instead of per core level (as in [10], [18]) and thereby the time and memory reservation aspects are applied per server.

## X. Conclusions

We have presented the first implementation of the Multi-Resource Server (MRS) for reserving both CPU- and memory-bandwidth on multi-core systems. We have evaluated the implementation of MRS and the results show that overhead of our implemented functionality is low. Furthermore, we have demonstrated the MRS suitability to execute legacy uni-core applications in a predictable manner on a multi-core platform by limiting interference between applications running on different cores.

Our demonstration shows that scheduling alone (i.e. controlling the allocation of resources over time) is not enough to achieve complete timing isolation. We observe that cache-pollution can have a tangible effect on timing properties of tasks executing in different serves. However, we also show that MRS, itself, can be used to mitigate cache-pollution since it bounds the effect on the shared cache for each server. Nevertheless, we conclude that our MRS should be complemented with some technique to remove/bound cache-pollution amongst servers, e.g., accounting for it in the analysis [6], [7] or implementing a cache partitioning solution [4], [5].

In spite of the advantage of providing a kernel modification-free solution by ExSched framework, it includes some overheads by itself. One research direction is to look at other implementation alternatives to achieve better performance. Another future direction is to find an algorithm to calculate the optimum budgets for both resources of the MRS. Some smart online algorithms can be developed to assign the unused capacity of one resource to another server to improve overall average response times.

## References

[1] J. P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. 8th IEEE Real-Time Systems Symposium (RTSS' 87)*, pages 261–270, December 1987.

[2] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems*, 1(1):27–60, June 1989.

[3] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS' 03)*, pages 2–13, December 2003.

[4] B.C. Ward, J.L. Herman, C.J. Kenna, and J.H. Anderson. Making shared caches more predictable on multicore platforms. In *Proc. 25th Euromicro Conf. on Real-Time Systems (ECRTS' 13)*, 2013.

[5] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proc. 19th IEEE Real-Time Technology and Applications Symposium (RTAS' 13)*, 2013.

[6] S. Schliecker and R. Ernst. Real-time Performance Analysis of Multiprocessor Systems with Shared Memory. *ACM Transactions in Embedded Computing Systems*, 10(2):22:1–22:27, January 2011.

[7] D. Dasari and B. Anderssom and V. Nelis and S.M. Petters and A. Easwaran and L. Jinkyu . Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *Proc. of the IEEE International Conference on TrustCom '11*, Nov 2011.

[8] M. Behnam, R. Inam, T. Nolte, and M. Sjödin. Multi-core composability in the face of memory bus contention. In *5th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'12)*. ACM, December 2012.

[9] I. Shin and I. Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS' 03)*, pages 2–13, December 2003.

[10] H. Yun and G. Yao and R. Pellizzoni and M. Caccamo and L. Sha. Memory- access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *Proc. 24th Euromicro Conf. on Real-Time Systems (ECRTS' 12)*, July 2012.

[11] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. 7th IEEE Real-Time Systems Symposium (RTSS' 86)*, pages 181–191, December 1986.

[12] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-time Environments. *IEEE Transactions on Computers*, 44(1), 1995.

[13] Daeyoung Kim, Yann-Hang Lee, and M. Younis. Spirit-ukernel for strongly partitione real-time systems. In *Proc. of the 7th International conference on Real-Time Computing Systems and Applications (RTCSA'00)*, 2000.

[14] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. H. Ashjaei, and S. Afshar. Support for Hierarchical Scheduling in FreeRTOS. In *16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*, France, September 2011.

[15] M.Åsberg, T. Nolte, S. Kato, and R. Rajkumar. Exsched: An external cpu scheduler framework for real-time systems. In *18th IEEE International Conference (RTCSA' 12)*, 2012.

[16] M.M.H.P. van den Heuvel, M. Holenderski, R. J. Bril, and J. J. Lukkien. Constant-bandwidth supply for priority processing. *IEEE Transactions on Consumer Electronics (TCE)*, 57(2), 2011.

[17] R. Inam, J. Mäki-Turja, M. Sjödin, and M. Behnam. Hard Real-time Support for Hierarchical Scheduling in FreeRTOS. In *7th Annual Workshop (OSPERT' 11)*, pages 51–60, Porto, Portugal, July 2011.

[18] 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 *Proc. 19th IEEE Real-Time Technology and Applications Symposium (RTAS' 13)*, 2013.

[19] R. Inam, J. Slatman, M. Behnam, M. Sjödin, and T. Nolte. Towards implementing multi-resource server on multi-core Linux platform. In *18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 13), WiP*, September 2013.

[20] S. Han and H.W. Jin. Full virtualization based ARINC - 653 partitioning. In *30th IEEE/AIAA Digital Avionics Systems Conference*, 2011.

[21] Sanghyun Han and Hyun-Wook Jin. Kernel-level ARINC-653 partitioning for Linux. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12)*, 2012.

[22] 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 *Proc. 28th IEEE Real-Time Systems Symposium (RTSS' 07)*, pages 49–60, December 2007.

[23] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Worst-case Response Time Analysis of Resource Access Models in Multi-core Systems. In *Design Automation Conference (DAC '10)*, 2010.

[24] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A Predictable SDRAM Memory Controller. In *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256, September 2007.

[25] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):2:1–2:24, 2009.

[26] S. Stuijk, T. Basten, M. C W Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Design Automation Conference, (DAC '07)*, 2007.

[27] C. Zimmer and F. Mueller. Low contention mapping of real-time tasks onto tilepro 64 core processors. In *In (RTAS' 12)*, 2012.

[28] R. Pellizzoni and A. Schranzhofer and J.-J.Chen and M. Caccamo and L. Thiele. Worst Case Delay Analysis for Memory Interference in Multicore Systems. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE' 10)*, pages 759–764, 2010.

[29] S. Bak and G. Yao and R. Pellizzoni and M. Caccamo. Memory-Aware Scheduling of Multicore Task Sets for Real-Time Systems. In *Proc. of the (RTCSA '12)*, 2012.

[30] R. Pellizzoni and E. Betti and S. Bak and G. Yao and J. Criswell and M. Caccamo and R. Kegley. A PRedictable Execution Model for Cots-based Embedded Systems. 2011.

[31] T. Kelter and H. Falk and P. Marwedel and S. Chattopadhyay and A. Roychoudhury. Bus-Aware Multicore WCET Analysis Through TDMA Offset Bounds. In *Proc. 23th Euromicro Conf. on Real-Time Systems (ECRTS' 11)*, June 2011.