Multi-core Composability in the Face of Memory-bus Contention*

Moris Behnam, Rafia Inam, Thomas Nolte, Mikael Sjödin
Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
Email: {moris.behnam, rafia.inam, thomas.nolte, mikael.sjodin}@mdh.se

ABSTRACT
In this paper we describe the problem of achieving composability of independently developed real-time subsystems to be executed on a multi-core platform, and we provide a solution to tackle it. We evaluate existing work for achieving real-time predictability on multi-cores and illustrate their lack with respect to composability.

To address composability we present a multi-resource server-based scheduling technique to provide predictable performance when composing multiple subsystems on a shared multi-core platform. To achieve composability on multi-core platforms, we propose to add memory bandwidth as an additional server resource. Tasks within our multi-resource servers are guaranteed both CPU- and memory bandwidth; thus the performance of a server will become independent of resource usage by tasks in other servers. We are currently implementing multi-resource servers for the Enea OSE operating system for a Freescale P4080 8-core processor, to be tested with software for a 3G-basestation.

Keywords
server-based scheduling, memory throttling

1. INTRODUCTION
Contemporary scheduling of real-time tasks on multi-core architectures is inherently unpredictable, and activities in one core can have negative impact on performance in unrelated parts of the system (on other cores). A major source of such unpredictable negative impact is contention for shared physical memory. In commercially existing hardware, there are currently no mechanisms that allow a subsystem to protect itself from negative impact if other subsystems start stealing its memory bandwidth. For performance-critical real-time systems, overcoming this problem is paramount.

In this paper we review existing work in memory-aware real-time scheduling and analysis and illustrate their shortcomings with respect to providing composability. Our goal is to develop a multi-resource server technology that allows a subsystem (i.e. a set of tasks allocated to a server) to be developed, tested and verified independently of other subsystems that will share the same hardware in the final system. Further, we want to achieve this goal using Commercial Off-The-Shelf (COTS) hardware in order to allow legacy (single-core) systems to be migrated to multi-cores without negative or unpredictable performance penalties.

We present multi-resource servers that provide composable hierarchical scheduling on multi-core platforms. We extend traditional server-resources by associating a memory-bandwidth to each server; thus a multi-resource server has both CPU-bandwidth and memory bandwidth allocated to it. The multi-resource servers are useful to provide partitioning, composability and predictability in both hard and soft real-time systems. In this paper we outline a theoretical framework to provide hard real-time guarantees for tasks executing inside a multi-resource server. A system consisting of a set of components (subsystems) is said to be composable if properties that have been established during the development of each component in isolation do not change when the components are integrated [1]. In this paper we focus on the schedulability of tasks, i.e. meeting their deadlines, as the main timing property, thus a system is composable if the schedulability of tasks that have been validated during the development of each component is guaranteed when the components are integrated.

The main contribution of this paper are:

- Proposing a multi-resource server that provide both CPU- and memory bandwidth isolation among servers running on a multi-core platform.
- Presenting a compositional schedulability analysis technique to assess the composability of subsystems containing hard real-time tasks. Using the compositional analysis technique, the system schedulability is checked by composing the subsystems interfaces which abstract the resource demand of subsystems [2].
- Presenting some technical details to implement the multi-resource server targeting the Freescale P4080 processor running the Enea OSE operating system.

Paper Outline: Section 2 presents the related work on server-based and memory-bandwidth access for multi-core systems. Section 3 explains our system model. Section 4 gives an overview of our multi-resource server, and few details for its implementation. The local schedulability analysis is described in Section 5. In Section 6 we provide a brief discussion on our assumptions, and finally Section 7 concludes the paper.

2. RELATED WORK
Within the real-time community a number of server-based techniques have been developed to provide real-time guarantees [3, 4, 5]. These server-based scheduling techniques were initially developed to introduce predictable interference from unknown execution requests by aperiodic task releases [6].

---

*This work is supported by the Knowledge Foundation (KK-Stiftelsen), via the research programme PPMSched.
Later on the good property of predictable interference provided by server-based scheduling techniques was given attention also in the context of software integration. In fact, server-based scheduling techniques do not only provide a predictable interference seen from outside of the server, but they also provide a predictable resource share within the server [2]. Hence, executing tasks within a server also guarantees this software a predefined share of the resource that the server is scheduling. Moreover, putting servers in servers allow for a hierarchical approach to software integration, where software parts in the ideal case could be developed independently of each other, and then later could be easily integrated. A large number of hierarchical scheduling techniques have been developed using different fundamental scheduling techniques, like [2, 7, 8, 9]. Also solutions for multi-core architectures have been developed [10], however, based on strong (often unrealistic) assumptions on no interference originating from the shared hardware.

All the aforementioned related work rely on the one central assumption that the hardware will not cause any side-effects in terms of software parts interfering with each other in any unpredictable manner. This assumption worked fairly well in single-core architectures, however, when dealing with multi-core architectures, the assumption is no longer valid.

Works that actually study the problem of memory contention in the context of multi-core embedded systems are somewhat scarce. Specifically very little work has been performed related to compositional analysis and on server-based methods.

Some approaches to Worst-Case Execution-Time (WCET) analysis are emerging which analyze memory-bus contention, e.g. [11]. However, WCET-approaches do not tackle system wide issues and do not give any direct support to provide isolation between subsystems.

Schliecker et al. [12, 13] 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. They have also measured the effect of different cache sizes. They used compositional analysis to measure the response times of tasks. A tighter upper bound on the number of requests, than the work of [12, 13], is presented by Dasari et al. [14] where they provide a response-time analysis for COTS based multi-core systems and they solve the problem of interleaving cache effects by using non-preemptive task scheduling. They have used hardware performance counters (PMC) 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 independently developed subsystems and the use of memory servers to limit the access to memory bandwidth.

Some highly predictable Time Division Multiple Access (TDMA) based techniques are used to access the shared resources (memory bus arbitration) and the worst case execution time WCET analysis have been provided using a multiprocessor system on chip architecture. Rosen et al. [15] 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. Static scheduling is used to restrain the number of memory accesses by each processor. Later Schranzhofer et al. [16] relaxed the assumption of fixed positions for the bus access using the same TDMA technique to arbitrate the shared bus. They divide tasks into sets of superblocks that are specified with a maximum execution time and a maximum number of memory accesses, and these superblocks are executed either sequentially or time-triggered. 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. Another work that guarantee a minimum bandwidth and a maximum latency for each memory access is proposed by Akesson et al. [17] by designing a two-step approach to share a predictable SDRAM memory controller for real-time systems. This is a key component in CoMPSoC [18]. Our approach, however, uses COTS hardware and it is software based using performance counters which are available in almost all processors.

Pelizzoni et al. [19] initially proposed the division of tasks into superblock sets by managing most of the memory request either at the start or at the end of the execution blocks. This idea of superblocks was later used in TDMA arbitration [16].

Bak et al. presents a memory aware scheduling for multi-core systems in [20]. They use PRedictable Execution Model (PREM) [21] compliant task sets for their simulation-based evaluations. However, they do not consider a server-based method. Further the usage of PREM requires modifications in the existing code (tasks), hence this approach is not compliant with our goal to execute legacy systems on the multi-core platform.

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 [22] for an Intel architecture running Linux. In their model, the system consists of one critical core executing a critical application and a set of non-critical cores executing non-critical applications. Memory servers are used to limit memory requests generated by tasks that are located on non-critical cores, one server per core. A software-based memory throttling approach is used to limit the memory-accesses of the interfering cores so that the interference from other non-critical cores on the critical core is bounded. To implement the software-based memory throttling, both measurement of the memory access and enforcement (enforcement means take an action when reaching a certain limit) mechanisms are required. Hardware performance counters are used to measure last level cache misses to measure memory accesses and the scheduler is updated to enforce memory throttling. A response time analysis has been proposed for tasks that are located on critical cores, including the interference that can be generated from non-critical cores (memory servers).

We propose a more general approach to support both composability and independent development of subsystems by using servers on all cores. The analysis in [22] only considers the memory server on non-critical cores while we present analysis for both time and memory aspects of the servers executing on all cores. They have only one server per core while we consider multiple servers per core. Further they have presented the analysis of tasks executing on the critical core that do not use servers. We use software-based memory throttling however, our memory throttling mechanism is proposed per server level instead of per core level using different hardware and software platforms than the previous work.

3. SYSTEM MODEL

Here we present our target hardware platform, the system model, and the assumptions we follow.
3.1 Architecture

In our work we assume that we use a processor with a set of identical cores that have uniform access to the main memory. Each core has a set of local resources; primarily a set of caches for instructions and/or data and busses to access these from the cores. The system has a set of resources that are shared amongst all cores; this will typically be the Last-Level Cache (LLC), main-memory and the shared memory bus. Our target hardware, the Freescale P4080 8-core chip [23], complies to these assumptions; its architecture is presented in Figure 1.

We assume that a local cache miss is stalling, which means whenever there is a miss in a local cache the core is stalling until the cache-line is fetched from memory. In this initial work 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. We use a simplified notion that memory accesses are uniform in time; for this purpose we use only the worst case memory latency which typically occurs when a miss to the last-level cache requires memory to be fetched from a non-opened row in a DRAM-bank. It is worth noticing that any single-core could easily generate enough memory traffic to saturate the memory bus.

3.2 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 2. The parent node is a node scheduler and leaf nodes are the subsystems (servers). Each subsystem contains its own internal set of tasks that are scheduled by a local scheduler. The node scheduler is responsible for dispatching the servers according to their bandwidth reservations (both CPU- and memory-bandwidth). The local scheduler then schedules its task set according to a server-internal scheduling policy.

4. THE MULTI-RESOURCE SERVER

In this section we present the multi-resource server. To bound the memory accesses for a server, we need some mechanism of memory throttling to control the activity of memory requests of the server such that each server can have access to the memory according to its pre-reserved memory bandwidth limit. Later in this section, we first describe the means to implement memory throttling by measuring the memory bandwidth, and then we explain how we can monitor and enforce the consumed bandwidth $m_s$.

4.1 The concept of multi-resource server

Each multi-resource server maintains two different budgets; one is the CPU-budget and the other one reflects the number of memory requests. The server is of periodic type, meaning that it renews both budgets to the maximum values periodically. At the beginning of each server period this is done as $q_s := Q_s$, $m_s := M_s$.

Each core has its own node scheduler which schedules the servers on that core. The node scheduler maintains a state of each server on the core; the state is either runnable or suspended. A server is in the runnable state if it still has allocated resources to use. Formally: $\text{runnable}_s = q_s > 0 \land m_s > 0$. That is, a server needs to possess both remaining CPU- and memory-budgets to be runnable.

A server which depletes any of its resources is placed in the suspended state, waiting for replenishment which will happen at the beginning of the next period.

The node scheduler selects a runnable server for execution. When a server $S_s$ is executing, $q_s$ decreases with the progression of time, while $m_s$ decreases when a task in the server issues a memory request using the shared memory-bus.

A scheduled server uses its local scheduler to select a task to be executed according to the local scheduling mechanism. When a task
Running inside a server $S_i$, issues a memory request, the associated core is stalling until the cache-line is fetched from memory. A higher priority task can pre-empt the execution of lower priority tasks but not during the core stalling. A memory request is managed even if the budget of $S_i$ has depleted. When the memory budget is depleted, the remaining CPU budget will be dropped and the state of the server is changed to suspended.

4.1.1 The CPU part of the server
For the CPU part of the server, any type of periodic server (i.e. idling periodic [24] or deferrable servers [25]) could be used to determine the consumption of allocated CPU budget of each server. In idling periodic servers, tasks execute and use the server’s capacity until it is depleted. If a task arrives before the budget depletion, it will be served. In case of the deferrable server, tasks execute and use the server’s capacity until it is depleted. If the deferrable server has capacity left but there is no task ready then it suspends its execution and preserves its remaining budget until its period ends. If a task arrives later but before the end of the server’s period, it will be served and it will consume the server’s capacity until the capacity depletes or the server’s period ends. If the capacity is not used before the period end, it is lost. There have been many previous implementations of both types of servers in single-core environments e.g. [26, 27].

4.1.2 The memory part of the server
The memory part of the server is modelled as deferrable server since the memory-budget is consumed only when a memory request is made. The implementation of such a server has been investigated in [22]. We plan to implement the same kind of implementation, however, using a different hardware and operating system. Some parts of our server resemble their implementation and some will be different because of the underlying platform.

4.1.3 The execution of the server
We explain the execution of a multi-resource server using a simple example of a subsystem that consists of five tasks $\tau_1, \ldots, \tau_5$ where $\tau_1$ has the highest priority and $\tau_5$ has the lowest priority. Figure 3 shows a possible execution scenario of tasks inside a server $S_i$ having $(P_s, Q_s, M_s)$. 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 will be served, and after a very small time $\tau_4$ is released and it pre-empts 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 will be 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 the second budget period, the highest priority ready task $\tau_1$ is executed. It issues a memory request, and as a result it waits until the memory request is completed.

4.2 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 memory throttling technique similar to what has been used in [22], where they limit the memory throttling per core, however we plan to limit it per subsystem level. As mentioned earlier, each core can contain more than one subsystem.

Most modern processors host a range of Performance Monitor Counters (PMCs) which can be used to infer the amount of resources consumed. 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 by monitoring the LLC misses. These could be used to interrupt a server once its allocated memory budget $M_s$ has been consumed.

We target the Freescale P4080 processor that hosts eight e500mc cores [28] and the associated memory-controller to determine the events that are needed to be counted in order to accurately determine the amount of consumed memory budget [29]. The PMCs are described in general in [23] and with more details in Table 9-47 in the e500mc Core Reference Manual [28]. The e500mc has 128 countable events which we can use to infer how much memory-bandwidth that has been consumed. In our ongoing work we are implementing a solution using the performance counters for an industrial real-time multi-core operating-system, OSE [30], developed by ENEA. Our initial aim is to provide a predictable environment which will allow migration of single-core legacy systems (software of a 3G-base station of our industrial partners) to multi-core, and that will allow performance tuning by predictable allocation of execution resources.

4.3 Online monitoring and policing of $m_s$
To provide continuous online monitoring of the consumed bandwidth, we need to continuously monitor counters and store their values. For any given CPU architecture, it 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 will implement servers that enforce/policing the consumed bandwidth and thus accurate and non-intrusive estimates of the bandwidth consumptions become even more important. For policing purposes using alarms could give the most accurate approach for accounting of the consumed bandwidth. In the e500mc, we can set wrap-around alarms on counters. Thus, if we would like to allow an application to generate at most $x$ events before being policed we could initialize the 64-bit event counter to $2^{64} - x$ before running the application. For L3-cache events and main-memory accesses, it is yet unclear to us whether we can set the alarms or not, and we are currently investigating this.

![Figure 3: An example illustrating the multi-resource server](image-url)
When counter-alarms are not possible to use, we propose the use of periodic polling of performance monitor counters to evaluate $m_s$. This could obviously result in a situation where $m_s < 0$, which at a first glance would seem inappropriate (or, for hard real-time, even dangerous). In server-based scheduling there exist techniques to handle such overruns if they are bounded [31]. Periodic polling cannot enforce accurate memory budget utilization by servers; however periodic polling along with overrun methods can be used in case of soft real-time systems.

5. COMPOSITIONAL ANALYSIS

In this section, we present the schedulability analysis for the multi-resource server proposed in this paper. To check the schedulability of the system we use the compositional schedulability analysis techniques in which the system schedulability is checked by composing the subsystems interfaces which abstract the resource demand of subsystems [2]. The analysis is performed in two levels; the first level is called the local schedulability analysis level where for each subsystem its interface parameters are validated locally based on resource demand of its local tasks. In the second level, which is called the integration or the global schedulability level, the subsystems interfaces are used to validate the compensability of the subsystems. In the following subsection we present the required analysis that will be used to verify the parameters of the subsystems interfaces.

5.1 Local schedulability analysis

First, we present the local schedulability analysis without considering the effect of the memory bandwidth part of the multi-resource server, i.e., assuming a simple periodic server, and then we extend the analysis to include the effect of the memory bandwidth requests. Note that, at this level, the analysis is independent of the type of the server as long as the servers follow the periodic model, i.e., a budget is guaranteed every server period. We assume that the server’s period, CPU-budget, and memory bandwidth request-budget are all given for each server.

5.1.1 Considering only CPU-budget

The local schedulability analysis under FPPS is given by [2]:

$$\forall \tau_i, \exists t_0 < t \leq D_i, \quad \text{rbf}_i^*(t, i) \leq \text{sbf}_i(t),$$

(1)

where $\text{sbf}_i(t)$ is the supply bound function that computes the minimum possible CPU supply to $S_i$ for every time interval length $t$, and $\text{rbf}_i(t, i)$ denotes the request bound function of a task $\tau_i$ which computes the maximum cumulative execution requests that could be generated from the time that $\tau_i$ is released up to time $t$. $\text{sbf}_i(t)$ is based on the periodic resource model presented in [2] and is calculated as follows:

$$\text{sbf}_i(t) = \begin{cases} t - (k - 1)(P_i - Q_s) - BD_s & \text{if } t \in W^{(k)} \\ (k - 1)Q_s & \text{otherwise} \end{cases},$$

(2)

where $k = \max \left\{ \left\lfloor \frac{t + (P_i - Q_s) - BD_s}{P_i} \right\rfloor, 1 \right\}$ and $W^{(k)}$ denotes an interval $[(k - 1)P_i + BD_s, (k - 1)P_i + BD_s + Q_s]$. Blackout Duration $BD$ is the longest time interval that the server cannot provide any CPU resources to its internal tasks and it is computed as $BD_s = 2(P_i - Q_s)$. The computation of $BD$ guarantees a minimum CPU supply, in which the worst-case budget provision is considered assuming that tasks are released at the same time when the subsystem budget has depleted, the budget has been served at the beginning of the server period, and the following budget is supplied at the end of the server period due to interference from other higher priority servers.

For the request bound function $\text{rbf}_i^*(t, i)$ of a task $\tau_i$, to compute the maximum execution requests up to time $t$, we assume that $\tau_i$ and all its higher priority tasks are simultaneously released. $\text{rbf}_i^*(t, i)$ is calculated as follows:

$$\text{rbf}_i^*(t, i) = C_i + \sum_{\tau_k \in HP(i)} \left[ \frac{t}{T_k} \right] \times C_k,$$

(3)

where $HP(i)$ is a set of tasks with priority higher than that of $\tau_i$. Looking at (3), it is clear that $\text{rbf}_i^*(t, i)$ is a discrete step function that changes its value at certain time points $(t = a \times T_k$ where $a$ is an integer number). Then for (1), $t$ can be selected from a finite set of scheduling points $\{SP_i\}$.

5.1.2 Considering CPU- and memory-budget

In [13, 14, 22], the effect of the memory bandwidth access has been included in the calculations of the response times of tasks. The basic idea in all these works is the computation of the maximum interference $MI(t)$ caused by the memory-bandwidth contention on tasks during time interval $t$. The new request bound function including the memory-bandwidth contention is provided in (4).

$$\text{rbf}_i^*(t, i) = C_i + \sum_{\tau_k \in HP(i)} \left[ \frac{t}{T_k} \right] \times C_k + MI(t).$$

(4)

In their approaches, $MI(t)$ is computed by multiplying the time needed for a request to be completed by the upper bound of memory-bandwidth requests in $t$ issued by all other tasks (issued by all other servers in [22]) located on cores other than the one hosting the analyzed task. However, this method cannot be used in our case since we assume that subsystems are developed independently hence the tasks’ parameters that belong to the other cores are not known in advance. In addition, the effect of having two different budgets (CPU- and memory-) should be accounted for in the multi-resource server, which has not been considered in the previous works.

5.1.3 Computing maximum memory interference $MI(t)$

To solve this problem, we now focus on the memory-bandwidth requests that can be generated by the tasks running inside a multi-resource server. Considering the behaviour of the multi-resource server presented in the previous section, we can distinguish two cases that can affect its tasks’ execution.

1. When a task $\tau_i$, executing in $S_s$, issues a memory-bandwidth request that causes a miss in a local cache, the associated core is stalling until the cache-line is fetched from memory. The maximum time that the task can be delayed due to the core in stalling state is denoted as $D_\ell_s$, and this delay should be considered in the analysis. Let $D_\ell_s$ define the maximum delay value among all tasks that execute in that server, i.e., $D_\ell_s = \max(D_\ell_i, s) \mid \forall \tau_i \in S_s$.

2. CPU-budget depletion due to memory-budget depletion. When tasks belonging to the same server issue $M_s$ memory requests, the memory-budget will deplete which will force the CPU-budget to be depleted as well. In the worst case, $M_s$ memory requests can be issued from tasks of the same server $S_s$ sequentially, i.e., tasks send a new request directly after
serving the current one. If this happens at the beginning of the server execution, a complete CPU-budget will be dropped and the server’s internal tasks will not be able to execute during this server period (this case is shown in the first server period in Figure 4).

For the first case, the value of $D_{\ell_{1}}$ depends on the maximum number of requests that can be generated in all other servers along with the policy used to serve parallel requests. However, since we assume that the information from other servers is not available because of independent development of subsystems then $D_{\ell_{1}}$ will not be known in advance. To solve this problem, we use schedulability analysis to find the maximum value of $D_{\ell}$ in each server $S_i$ that guarantees the schedulability of all tasks of $S_i$ and this value is denoted as $D_{\ell_{1}}^{est}$. Then at the system integration level, where the parameters of all servers are given, the actual value of $D_{\ell_{1}}$ can be evaluated and then compared with $D_{\ell_{1}}^{est}$. If $D_{\ell_{1}}^{est} < D_{\ell_{1}}$, then we can guarantee that the memory-bandwidth access will not cause any deadline miss.

In the schedulability analysis of $\tau_i$, we model the task delay due to core stalling as an interference from a fictive higher priority task $\tau^*_{f|e}$, with an execution-time equal to $D_{\ell_{1}}^{est}$. The number of times that $\tau^*_{f|e}$ interferes with the execution of $\tau_i$ is defined as $NR(i,t)$ which equals to the maximum number of memory requests that can be generated at a time interval $t$. Note that during the execution of $\tau_i$, it can be delayed by at most one memory request sent from a lower priority task and by the number of requests that the task itself sends and finally by the higher priority tasks that can preempt its execution. Thus (3) can be used to find $NR(i,t)$ by substituting $C_i$ and $C_h$ by the number of memory request of the corresponding tasks, i.e., $CM_i + 1$ and $CM_h$ respectively. Note that we include the delay due to a memory request sent from a lower priority task in the equation by adding one in $CM_i + 1$. Finally, $MI(t)$ in (4) is then calculated by $MI(t) = NR(i,t) \times D_{\ell_{1}}^{est}$.

$$M I(t) = \begin{cases} NR(i,t) \times D_{\ell_{1}}^{est} + \max \left( \left\{0, \frac{NR(i,t)}{M_i}\right\} \times \left(Q_s - M_i \times D_{\ell_{1}}^{est}\right) \right) & \text{if } M_i \times D_{\ell_{1}}^{est} \geq Q_s, \\ NR(i,t) \times D_{\ell_{1}}^{est} & \text{otherwise.} \end{cases}$$

Now, we explain the evaluation of $D_{\ell_{1}}^{est}$. Let us define the CPU-resource slack of a task $\tau_i$ as $sl(i,t) = spf_{\ell}(i,t) - q_{\ell}(i,t)$, and the tasks is schedulable if $\exists t \in \{SP\} | sl(i,t) \geq 0 \ \forall \ \text{see } (1)$. The task will be schedulable after adding $MI(t)$ if and only if $\exists t \in \{SP\} | MI(t) \leq sl(i,t)$, i.e., at certain $t$ the interference on $\tau_i$ due to memory bandwidth accesses does not exceed the CPU-slack of the task at the same time. From the previous condition, it is possible to find the maximum time that $\tau_i$ can be delayed due to memory bandwidth accesses without missing its deadline $D_{\ell_{1}}^{est}(i,s)$. Then to make sure that all tasks in $S_i$ will not miss their deadlines we select $D_{\ell_{1}}^{est} = \min(D_{\ell_{1}}^{est}(i,s))$ for all $\tau_i$ running within $S_i$.

5.2 System composability

To check the composability of the system consisting of a set of subsystems executing on a multi-core architecture, two different tests should be applied. The first test is on the CPU part to make sure that the required CPU budget will be provided. The second test is on the memory bandwidth to compute the actual maximum delay $D_{\ell}$ and compare it with $D_{\ell_{1}}^{est}$. Both tests can be performed independently and should succeed to guarantee that all tasks meet their deadlines. Therefore the parameters that should be provided in the interface of each subsystem $S_i$ to apply both tests are $P_i, Q_i, M_i, D_{\ell_{1}}^{est}$.

For the memory bandwidth test, first the maximum memory bandwidth delay $D_{\ell}$ is computed for each subsystem and then the following test is applied $D_{\ell} \leq D_{\ell_{1}}^{est}$ to check the schedulability of the subsystem after integration. The value of $D_{\ell}$ for each server depends on the interface parameters of all other servers and also on the policy used to schedule multiple parallel memory requests (for example, Round Robin RR, First In First Out FIFO, or TDMA). This keeps our subsystem independent from the underlying platform architecture since in our local analysis we have not made any assumption on how multiple parallel memory-bandwidth requests are scheduled. As a simple example and assuming the FIFO policy and knowing that only one request can be sent from each core at
a time (a core is stalling when a request is sent), then $D_{\ell_s}$ equals to the number of cores multiplied by the time taken to serve each request. The reason is that for each core when it tries to send a memory bandwidth request, as a worst case all other cores send one request just before the core under analysis, which bounds the number of requests. Unfortunately, the policy used in the platform considered in this paper is unknown for us and we are currently working on this part to identify it.

Since the CPU part of the server is of periodic type, each subsystem can be modelled as a simple periodic task where the subsystem period is equivalent to the task period and the subsystem budget is equivalent to the worst case execution time of a task. Then the schedulability analysis used for simple periodic tasks can be applied on all servers that share the same core for this test [2]. Note that since for each memory bandwidth request, the associated core is stalling then a higher priority server may be blocked by a lower priority server at most once with maximum blocking time equal to $D_{\ell_s}$. This blocking time should be considered in the analysis.

6. DISCUSSION

In this paper, we have made some simplifying assumptions to allow us to start tackling the very complex problem of multi-core composability. Some of these assumptions will incur considerable pessimism in the analytical results for schedulability. However, the assumptions have been selected to not impair implementability or efficiency of implementations.

In this section we highlight some hardware complexities that we have so far avoided. The purpose is to define the problem space that needs to be tackled in order to provide tight analytical results and to avoid over-estimations in run-time calculations of consumed memory bandwidth.

The assumption that the accesses to shared memory are uniform can be fulfilled by accounting for the worst case memory access delay in all memory accesses. However, this assumption incurs pessimism both in the analytical results and in the on-line accounting of how much memory bandwidth has been consumed. Whenever a memory access can be served from the LLC, the amount of time the shared bus is occupied is more than an order of magnitude shorter compared to when the memory access has to be served from a non-open row of DRAM.

Our assumption that each task has its own cache partition is often impractical to implement, and significantly increases the cache miss-ratio compared to sharing the cache with all local tasks. However, from a predictability and composability point-of-view, having shared local caches has negative impact; since the very nature of sharing resources between tasks and servers would incur unpredictable behavior. Our assumption limits the amount of resource sharing, however, sharing of data-elements between tasks on different cores must still be allowed. When updating shared cached data the cache-coherency fabric may incur traffic on the shared bus. In this paper we have not accounted for such memory traffic that do not stem directly from a load/store instruction.

Our assumption that the CPU stalls during a memory access may be true for simpler architectures. However, many high-performance architectures allow speculative and out-of-order execution; this could allow some instructions to complete while a memory access instruction is being stalled. This effect, however, is typically bounded by the number of instructions that can be simultaneously executed in the CPU pipeline. More advanced forms of parallelism, such as CPU multi-threading, allow a complete task/thread switch to occur upon a memory stall. This way a non memory-bound task does not have to be penalized by memory stalls caused by other tasks. These types of in-core parallelism could have beneficial impact on the local schedulability if we find ways to model and analyze it (however, it will not affect the on-line accounting of consumed memory bandwidth).

Yet a source of (unaccounted for) parallelism is in the shared memory subsystem. For instance it is quite common to have multiple DRAM controllers that could serve memory requests in parallel; however the exact degree of parallelism obtained would depend on how the external cache and DRAM-banks are organized. Far bigger impact on our assumption of one access being served at a time is the anticipated architectures with interleaved access to the shared memory bus. Such architectures will accept multiple memory requests in an asynchronous fashion and return the results upon completion; thus serving multiple requests at once, and returning the results in an unpredictable order.

For multi-cores with more than four cores we start to see clustered on-chip memory architectures where, e.g., only L1-cache is local and L2-caches are shared amongst a subset of cores. Such architectures will increase the complexity of our analytical solutions and implementations; however they will not violate any fundamental assumptions of our approach.

7. CONCLUSIONS

In this paper, we have proposed a multi-resource server approach to address composability of independently developed real-time subsystems executed on a multi-core platform. The memory-bandwidth is added as an additional server-resource to provide predictable performance of multiple subsystems. We have indicated a software-based memory throttling mechanism to bound the memory accesses for each server, hence tasks within a multi-resource server execute guaranteed with both CPU- and memory-budgets. We also provide a compositional analysis framework for multi-resource servers.

Ongoing work is to implement (1) memory throttling using hardware performance counters for our target hardware platform Freescale P4080 processor and (2) the multi-resource server-based hierarchical scheduling framework on an industrial real-time multi-core operating-system, OSE. Once the implementation efforts are complete, we will evaluate the implementation using legacy code from a telecom system (the software for a 3G radio base station and/or a 3G radio network controller) which will be migrated from a single-core system and executed within our multi-resource servers.

Another very interesting research direction is to investigate the relationship between the server parameters including the server period, memory-budget and CPU-budget and provide methods to evaluate the optimal parameters.

8. REFERENCES


