Dynamic Arbitration of Memory Requests with TDM-like Guarantees

Farouk Hebbache and Mathieu Jan
CEA, LIST, Embedded Real Time Systems Laboratory
Email: Firstname.Lastname@cea.fr

Florian Brandner and Laurent Pautet
LTCI, Télécom ParisTech, Université Paris-Saclay
Email: Firstname.Lastname@telecom-paristech.fr

***Abstract*—A major challenge with multi-cores in real-time systems is contention between concurrent accesses to shared memory. Dynamic arbitration schemes allow for an optimal utilization of the system's memory, while sacrificing time predictability. Time-Division multiplexing (TDM), on the other hand, sacrifices average-case performance in favor of predictability. In this work, we explore a dynamic arbitration scheme that is essentially based on TDM, and thus preserves many of its guarantees.**

I. Introduction

Multi-core architectures pose many challenges in real-time systems, which arise from the manifold interactions between concurrent tasks during their execution – most notably accesses to shared main memory. These interactions make it difficult to tightly bound the Worst-Case Execution Time (WCET) of real-time tasks. Engineers thus often resort to isolating critical tasks from other tasks. Time-Division Multiplexing (TDM), for instance, guarantees a fixed time window to a task to exclusively access memory. This provides predictable behavior and improves composability by bounding access latencies and guaranteeing bandwidth independently from other tasks.

TDM, however, is not *work-conserving* and often leads to low resource utilization. The problem arises when the owner of a TDM slot does not (yet) have a memory request ready to be served. Under a strict TDM scheme, this slot cannot be reclaimed by another task (as for instance under Round Robin).

This work explores a dynamic arbitration scheme that is based on TDM. The main idea is to associate deadlines with the memory accesses of tasks. The deadlines ensure that requests of critical tasks are in the worst-case completed at the same instant as an execution under a strict TDM scheme. At the same time, these deadlines allow the arbiter to estimate the *slack time* of each pending request in the system. If slack times permit, the arbiter can arbitrarily change the order in which requests are handled – and thus break with usual TDM conventions. Since slack times are tracked for each request individually and deadlines of critical requests are guaranteed to be respected at all times, TDM's guarantees are preserved.

We first provide a brief definition of a restricted system model that is assumed for the remainder of this paper. We then illustrate several variants of TDM-based schemes using motivating examples in Section III. Section IV then provides a more formal discussion. We present a preliminary evaluation using a simple use case in Section V. Section VI discusses related work. We finally conclude in Section VII.

II. System model

We assume a very simplified model. A system consists of a single memory and n cores, each executing a single task. Each task is a sequence of numbers representing the distance in clock cycles between the completion of a memory access and the arrival of the subsequent one (i.e., cache misses). The sequence $(2, 2, 0)$, for instance, represents a task performing three memory accesses. The first access occurs two clock cycles after the task's activation. The task then issues its second access two cycles after the completion of the first access, while the third memory access is issued immediately when the preceding access completes. We assume that the execution time between memory requests is constant and is independent from the execution of other tasks, ignoring the blocking time that tasks have to wait for memory requests to be handled by the central main memory. All task are immediately activated.

The concurrent execution of the system's tasks is then governed by the way memory accesses are arbitrated. For simplicity, we define two classes of tasks: (1) critical tasks, and (2) non-critical tasks. Critical tasks (denoted by capital letters) have to complete their execution in time, i.e., respect a deadline. We do not explicitly specify the deadline though, since we assume that critical tasks meet their deadlines when executed under a standard TDM arbitration scheme. The arbiter may, however, diverge from such a scheme, e.g., by keeping track of outstanding requests and selecting one of them in order to give the owning task exclusive access to main memory. We will cover three TDM-based arbitration schemes (`TDM`, `TDMdz`, and `TDMds`) in more detail in the next section.

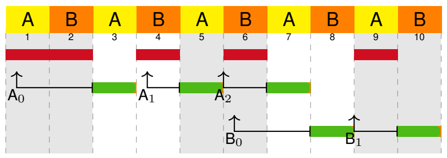
III. Motivating Examples

The following examples assume an architecture with 4 cores executing two critical tasks (A, B) and two non-critical tasks (c, d). Each critical task has a dedicated TDM slot, which alternate every 8 cycles. Non-critical tasks do not have TDM slots on their own and may only reclaim unused TDM slots.

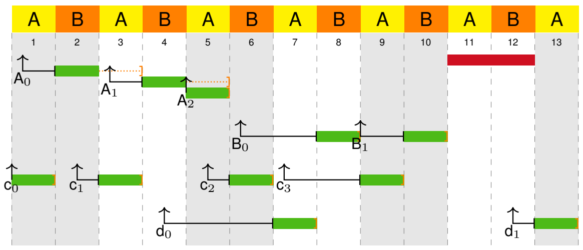
Considering the two critical tasks only, Figure 1a shows the arrival dates (↳) of memory requests (A_i, B_i) and the TDM slot during which requests completed (🟩) under strict `TDM`. Request A_1, for instance, arrives 2 cycles after completing A_0 in slot 4, and completes at cycle 39, the end of slot 5.

As indicated by the unused slots (🟥) at the top, `TDM` is not work-conserving (e.g., request A_0 at slot 2). These unused slots can be reclaimed by non-critical tasks. Under such a scheme, critical tasks would still be constrained to *only* use their dedicated TDM slots, giving no freedom to the arbiter.

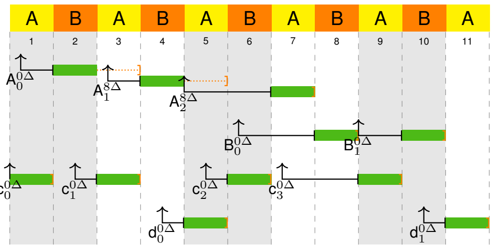
One way of interpreting TDM is to associate deadlines with requests. For strict `TDM` this deadline corresponds to the end of the next TDM slot of a critical task. For simplicity, we assign the end of the upcoming next TDM slot as the deadline for requests from non-critical tasks. If a critical and a non-critical task have the same deadline, the critical task wins. If a non-critical request misses its deadline, we simply



(a) (Strict) TDM considering only the two critical tasks A and B.



(b) Dynamic TDMdz without slack counters



(c) Dynamic TDMds with slack counters (Δ)

Fig. 1: Three TDM-based variants arbitration schemes.

push the deadline a slot length into the future. Deadlines are unique among critical tasks, the TDM arbiter thus can ensure that requests complete exactly at their deadlines.

Now, every request carries a deadline and a more dynamic TDM-based arbiter could very well chose to execute requests before their actual deadlines. For this it suffices to order the requests in a priority queue by their deadlines. On a tie the priority queue prioritizes the critical over non-critical tasks.

Subfigure 1b shows all 4 tasks executing under such an arbitration scheme, which we call TDMdz. Critical request may now complete before their deadline, which is indicated by a doted line (). In this example requests A_0 and A_1 each complete 1 TDM slot earlier than their deadlines. Non-critical tasks may reclaim unused TDM slots (c_1 in slot 3), but may also delay critical requests (d_0 wins over B_0 in slot 7).

A closer comparison between Subfigure 1a and Subfigure 1b reveals that request A_2 (and thus task A) completes two TDM slots ahead of the original execution under TDM (slot 5 vs. slot 7). This observation gives rise to an extension of TDMdz that tracks and accumulates slack time with regard to an execution under TDM. This can be done by subtracting the completion date of a request from its deadline. The slack (Δ) is used to compute the deadline of the task's next request.

Subfigure 1c illustrates the resulting arbitration scheme, TDMds. Tasks may now accumulate slack, which gives more freedom to the arbiter to choose which request to handle next. Requests c_2 and d_0 can both delay request A_2, which accumulated a slack of 2 TDM slot lengths at its arrival (c.f. the additional parameter 8Δ in Subfigure 1c). Completing request d_0 in slot 5 enables the arrival of request d_1 earlier. This allows to reduce the total number of cycles required to process all requests, from 13 TDM slots to 11 TDM slots, while keeping the same guarantees for critical requests as strict TDM.

IV. Formal Description

Based on the system model from Section II, we now provide a formal definition of our arbitration schemes.

The arbiter represents requests as pairs (a, d), consisting of the arrival date and deadline. Requests are immediately issued to the arbiter's priority queue upon arrival and ordered by their deadline (prioritizing critical tasks over non-critical). At the beginning of each TDM slot the arbiter selects a request r with the highest priority (i.e., lowest deadline) and assigns a completion date $C(r)$ to it. The deadline of all non-critical requests that missed their deadline is incremented by the slot length. If the queue is empty the TDM slot is unused, and no request is completed. An execution is valid if for every request $\mathsf{X}_i = (a, d)$ of a critical task $C(\mathsf{X}_i) < d$ holds.

The arbiter drives the system's execution based on the arrival times and deadlines of requests. As a first step, the arbiter needs to compute the arrival times of a memory request by accessing individual elements of the specification of its owning task, i.e., the sequence of numbers introduced in Section II. The ith entry in the sequence of task X is denoted by $\mathsf{X}(i)$. Considering, for instance, task A from before with its sequence $(2, 2, 0)$, the first and last elements of the sequence are given by $\mathsf{A}(0) = 2$ and $\mathsf{A}(2) = 0$ respectively. The arrival date of a request $\mathsf{X}_i = (a, d)$ is then simply $a = C(\mathsf{X}_{i-1}) + \mathsf{X}(i) + 1$.

Algorithm 1 shows how to compute the deadline d of a critical request under the TDMdz scheme. The algorithm takes three arguments, the arrival date, the start of the current TDM period, and the offset of the task's own TDM slot with regard to the beginning of a TDM period. In addition, the constant length of the TDM period is needed (TDMPERIOD). For non-critical tasks the deadline is the end date of the current TDM slot plus the TDM slot length. The arrival date and deadline of request A_2 in Figure 1b is computed from the completion date of the previous request $C(\mathsf{A}_1) = 31$ as follows: $\mathsf{A}_2 = (31 + 1 + 0, 31 + 1 + 7) = (32, 39)$. The arrival date and the start of the current TDM period, are at cycle 32, and the end of A's TDM slot w.r.t. the beginning of a TDM period is 7.

For the TDMds approach, an additional slack counter (X_Δ) needs to be stored for each critical task X. The request's actual arrival date does not change. Instead of computing the deadline from this arrival date as in TDMdz, Algorithm 1 is invoked with the arrival date of the request as it would have occurred under strict TDM: $C(\mathsf{X}_{i-1}) + \mathsf{X}(i) + 1 + \mathsf{X}_\Delta$. Considering again request A_2 from the motivating example. The original arrival date under strict TDM is computed from $C(\mathsf{A}_1) = 31$ and the value of A's slack counter $\mathsf{X}_\Delta = 8$, which gives $31 + 1 + 8 = 40$ (as in Subfigure 1a). The request thus falls into the TDM period starting at cycle 32, from which the deadline is computed as follows: $32 + 7 + 16 = 55$. Since the request's arrival offset $(40 - 32 = 8)$ is too large (Line 4), the request misses A's slot at offset 7 and is delayed by a TDM period (+16).

It remains to define how slack counters are updated. It is not possible to simply accumulate slack from one request to another by summation. The slack counter of a critical task X_Δ has to be recomputed after completion of its most recent

Algorithm 1 Deadline computation for critical tasks.

```
1: function DEADLINE(arrival, curPeriod, slotOffs)
2:     deadline = curPeriod + slotOffs
3:     if arrival − curPeriod > slotOffset then
4:         deadline = deadline + TDMPERIOD
5:     return deadline
```

request $r = (a, d)$ as follows: $\mathsf{X}_\Delta = d - C(r)$. This allows to accumulate slack over sequences of requests, as the deadline is pushed further into the future (e.g., for requests A_1 and A_2).

V. Preliminary Evaluation

We evaluated the `TDMds` approach using a simple use case consisting of 4 benchmarks from the MiBench [2] benchmark suite that were executed on the Patmos [7] architecture. For this, we extended the Patmos simulator to generate traces matching our system model. The traces were collected in single-issue mode (Patmos' VLIW features were disabled), using the following hardware configuration: a 32 KB *method cache* using the *LRU* replacement policy on 32 entries and 32 byte blocks, a 256 byte *stack cache* with block size 4, a 32 KB write-through *data cache* with *LRU* replacement on 4 sets and 32 byte blocks. We used Patmos's default configuration for main memory, which assumes an access latency of 21 cycles. The duration of TDM slots thus was set to 21 cycles.

We selected the 4 longest running programs from the benchmark suite, namely `rijndael` (task A, AES encryption), `blowfish` (task B, encryption), `djikstra` (task c, shortest path search), and `adpcm` (task d, audio encoding), and simulated a concurrent execution. Two of the benchmarks (`rijndael`, `blowfish`) where considered critical, which results on a TDM period of 42 cycles ($42 = 21 \cdot 2$). Table I summarizes the benchmark characteristics, including the number of accesses to main memory, the execution time as a single task in isolation (using TDM), and the execution time when running all four benchmarks concurrently.

During the simulation the evolution of various metrics was recorded, such as the number of granted memory accesses (Subfigure 2a), average memory wait time (Subfigure 2c), and slack counter values (Subfigure 2b). Due to the large number of simulated TDM slots (about 20 million) a data point in the figures represents approximately 6400 TDM slots. The figures show average values during this sampling period.

A close lock at the memory wait times (Subfigure 2c) reveals 5 different phases during the simulation, where the first phase ends after about 7 million TDM slots, which appears to be triggered by changes in the memory access patterns of a benchmark. The other phases are marked by the completion of one of the benchmark programs, after about 11.3, 16.7, 17.3, and 19.3 million TDM slots respectively.

Overall we can observe that the critical tasks exhibit a very constant behavior in terms of wait time and access numbers, while non-critical tasks appear to be subject to large variations. It also appears that the critical tasks largely saturate the main memory bandwidth, which results in large waiting times, in particular for `adpcm` (d). This also explains the low average slack counter values during the first execution phase. Note, however, that the average numbers are misleading here. Most of the time both critical tasks accumulate and immediately *spend* up to 200 cycles of slack during each sampling period.

TABLE I: Benchmark characteristics.

Benchmark	Memory accesses	Execution Time (cycles) isolated	Execution Time (cycles) concurrent
`rijndael`	7165378	236881722	349949607
`blowfish`	3527173	219815862	237530328
`djikstra`	4856155	209856906	362402145
`adpcm`	1716629	141243669	405932856
Total	17265335	–	–

The second phase appears to be marked by a reduction in the number of memory requests of `adpcm` (d) (see Subfigure 2a). While we were unable to determine the cause, we can observe that the average wait times for `adpcm` increases while that of `djikstra` (c) decreases. The reduced memory demand leads to a visible increase in the accumulation of slack for the critical tasks. Also the maximum slack values rise (up to 1000 cycles), which are as before quickly consumed.

The end of the critical task `blowfish` (b) marks the beginning of the third phase. Both non-critical task profit from the additional memory bandwidth. This is particularly true for `djikstra` (c), whose waiting times and access numbers are frequently better than those of the remaining critical task `rijndael` (`A`). However, there are marked spikes in `djikstra`'s (c) waiting times and access numbers that often coincide with drops in the slack counter values of the critical task. We also observe a huge increase in the slack counter values, which rise up to a maximum of 17136 cycles. As before slack is quickly consumed by the non-critical tasks.

The two final phases show considerable improvements in the waiting time and access number for `adpcm` (d) which terminates last in our simulation run.

From this simple use case we conclude that `TDMds` allows to accumulate considerable amounts of slack even when the memory bandwidth is highly saturated. The high variability in the slack counter values also indicates that our approach offers a very fine granularity of memory arbitration, while providing TDM-like guarantees for critical tasks.

VI. Related work

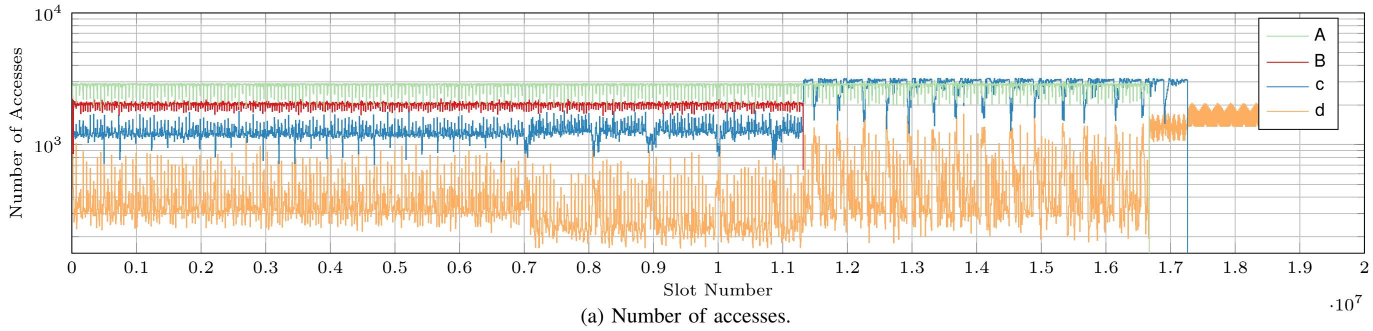
A common approach is to improve the resource utilization of TDM by increasing the number of TDM slots according to task weights [8]. Others apply strict TDM arbitration to critical tasks [1], while allowing other schemes for non-critical tasks. In both cases the TDM strategy itself is not modified, which remains non-work-conserving.

Kostrzewa et al. propose a technique to dynamically adapt to a varying number of *active* tasks [4], which execute under non-work-conserving TDM. Yonghui et al. truly *skip* unused entries in a TDM schedule in order to allow for variable-sized TDM slots [6]. Our approach likewise is work-conserving, but additionally allows to track and accumulate slack.

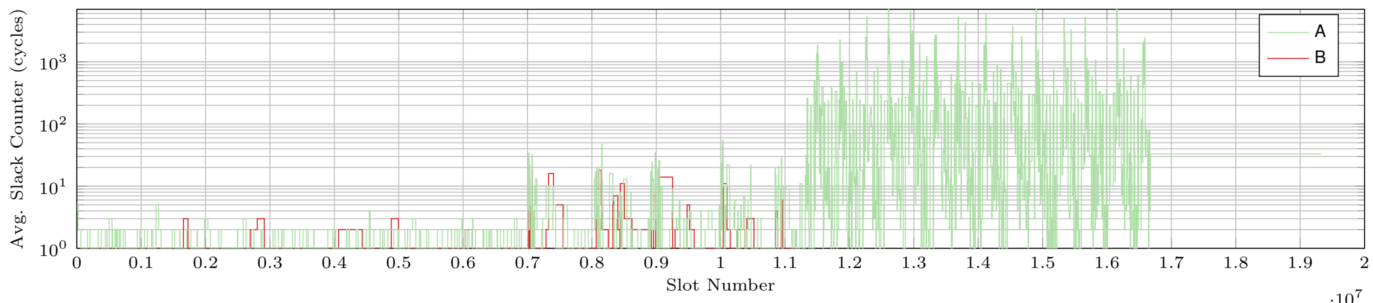
Similar to our approach, Kritikakou et al. [5] track the slack time of critical tasks in software. Non-critical tasks can access memory as long as all critical tasks still have slack left, otherwise *all* non-critical tasks are stopped. In our case, non-critical tasks may always continue execution and arbitration is performed at the granularity of individual memory accesses.

MemGuard [9] ensures isolation between cores by implementing a credit-based approach for tracking memory requests in software. Tasks executed by a core are suspended when the budget of memory requests, periodically assigned to the core, is depleted. A reclaim manager can donate *predicated* non-used budget of memory requests to other cores, making the approach suitable for soft real-time systems only. Our slack counters correspond to a memory budget that is not periodically replenished but accumulated, based on execution history. This allows for more flexibility while providing strict timing guarantees for critical tasks.

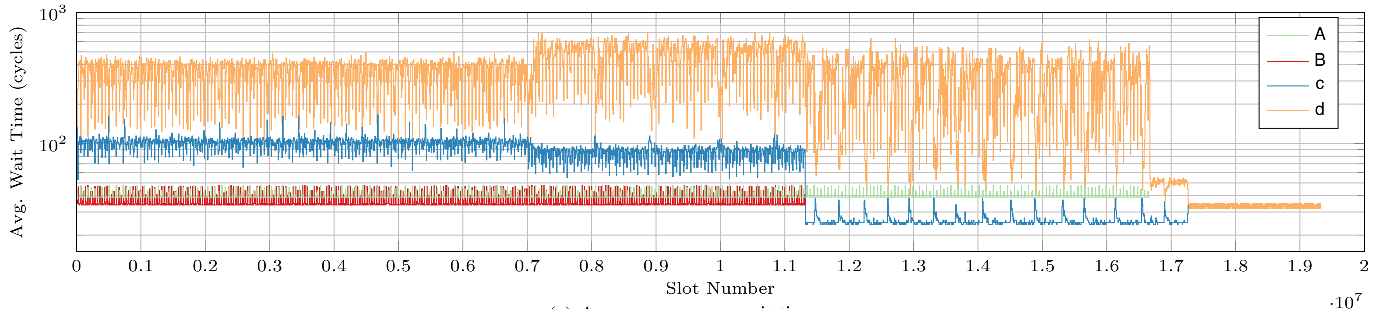
Closest to our work, Jun et al. [3] propose a slack-aware arbiter at the granularity of individual requests. However, slack



(a) Number of accesses.



(b) Slack counter.



(c) Average memory wait time.

Fig. 2: Trace of the concurrent execution of `rijndael` (A), `blowfish` (B), `djikstra` (c), `adpcm` (d).

is statically defined by a fixed parameter for each core (master) that is independent from the actual load on the main memory. Timing errors may thus occur, since it cannot be guaranteed that requests complete before their slack is entirely consumed. Also, slack cannot be accumulated across successive requests.

VII. Conclusion

We evaluated the behavior of `TDMds` using software simulation of a simple use case and obtained interesting results w.r.t. the dynamic behavior of slack counters, which allow a dynamic sharing of the memory resource between critical and non-critical tasks. Our evaluation considers a single use case only and does not capture realistic execution scenarios in a multi-task environment. As future work, we plan to investigate more realistic execution models, and implement our arbitration policies in hardware, using tree structures [6]. We also plan to allow other forms of slack to be accumulated, e.g., from execution time [5].

References

[1] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A globally arbitrated memory tree for mixed-time-criticality systems. *IEEE Trans. Comput.*, 66(2):212–225, Feb. 2017.

[2] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. of the Int. Workshop on Workload Characterization*, pages 3–14, 2001.

[3] M. Jun, K. Bang, H. J. Lee, N. Chang, and E. Y. Chung. Slack-based bus arbitration scheme for soft real-time constrained embedded systems. In *Asia and South Pacific Design Automation Conf.*, pages 159–164, 2007.

[4] A. Kostrzewa, S. Saidi, L. Ecco, and R. Ernst. Flexible TDM-based resource management in on-chip networks. In *Int. Conf. on Real Time and Networks Systems*, pages 151–160. ACM, 2015.

[5] A. Kritikakou, C. Pagetti, M. Roy, C. Rochange, M. Faugère, S. Girbal, and D. Gracia Pérez. Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems. In *Int. Conf. on Real-Time Networks and Systems*, 2014.

[6] Y. Li, B. Akesson, and K. Goossens. Architecture and analysis of a dynamically-scheduled real-time memory controller. *Real-Time Syst.*, 52(5):675–729, Sept. 2016.

[7] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, OASICS, pages 11–21, 2011.

[8] M.-K. Yoon, J.-E. Kim, and L. Sha. Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems. In *Real-Time Systems Symp.*, pages 227–238. IEEE, 2011.

[9] 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 *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2013.