Journal of Systems Architecture 118 (2021) 102178

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

An Analyzable Inter-core Communication Framework for High-Performance Multicore Embedded Systems

Rohan Tabish [a,*,1], Jen-Yang Wen [b,1], Rodolfo Pellizzoni [c], Renato Mancuso [d], Heechul Yun [e], Marco Caccamo [f], Lui Raymond Sha [a]

[a] University of Illinois at Urbana-Champaign, United States of America
[b] Microsoft Corporation, United States of America
[c] University of Waterloo, Canada
[d] Boston University, United States of America
[e] University of Kansas, United States of America
[f] Technical University of Munich, Germany

ARTICLE INFO

Keywords:
High-performance computing
Communication
Inter-core
Multicore
Heterogeneous systems
Embedded systems

ABSTRACT

Multicore processors provide great average-case performance. However, the use of multicore processors for safety-critical applications can lead to catastrophic consequences because of contention on shared resources. The problem has been well-studied in literature, and solutions such as partitioning of shared resources have been proposed. Strict partitioning of memory resources among cores, however, does not allow intercore communication.

This paper proposes a Communication Core Model (CCM) that implements the inter-core communication by bounding the amount of intercore interference in a partitioned multicore system. A system-level perspective of how to realize such CCM along with the implementation details is provided. A formula to derive the WCET of the tasks using CCM is provided. We compare our proposed CCM with Contention-based Communication (CBC), where no private banking is enforced for any core. The analytical approach results using San Diego Vision Benchmark Suite (SD-VBS) for two models indicate that the CCM shows an improvement of up to 65 percent compared to the CBC. Moreover, our experimental results indicate that the measured WCET using SD-VBS is within the bounds calculated using the proposed analysis.

1. Introduction

Commercial-off-the-shelf (COTS) multi-core processors have been developed by industry to meet the ever growing processing requirements. These processors offer great average case performance, low power consumption compared to multiple single cores as well as cost effective design. However, the use of multi-core processors for safety-critical applications can lead to the unpredictable timing behavior of the task on the core under consideration. This unpredictability in a multi-core processor is because of the contention on the shared resources such as DRAM, LLC and the Memory controller by the other cores. The problem has been well studied in the research community [1–5] and so far has been acknowledged in industry by Federal Aviation Authority (FAA) [6].

Researchers in [4] demonstrated that strict partitioning of the shared resources (LLC, bus bandwidth and DRAM banks) in a multi-core environment is required to achieve predictable execution of the tasks running on each core. A similar approach has been proposed by MC^2 in [7] where predictability in a multicore processor is ensured by implementing different isolation techniques for each criticality level. Strict partitioning of the shared resources has been adopted by FAA in its recent CAST32 A position paper [6].

The work in [8] describes how to implement inter-core communication for mixed-criticality tasks using cache isolation and DRAM banks in a multi-core processor inside MC^2 framework. However, in their proposed model all the cores that need to communicate compete for the same DRAM bank. This is a problem (as shown in evaluation of this paper) because it introduces significant amount of contention, making the communication slow. We refer to the communication between all the cores using the same bank described in [8] as CBC in this work. Another limitation of the work proposed in [8] is that they provide no theoretical bounds for their intercore communication mechanisms.

* Corresponding author.
E-mail address: rtabish@illinois.edu (R. Tabish).
[1] Equal authors.

https://doi.org/10.1016/j.sysarc.2021.102178
Received 24 December 2020; Accepted 10 May 2021
Available online 31 May 2021

To address these limitations, we propose CCM model that minimizes the number of cores accessing the same bank to support intercore communication. Moreover, we also provide a mathematical expression to theoretically analyze the schedulability when running tasks in our proposed CCM model.

Our work follow the partitioning approach described in [4] to propose and implement inter-core communication framework. When designing such a framework, our design philosophy is to minimize the number of cores accessing a DRAM bank at any point in time to avoid communication slow-down. Our proposed design is implemented using standard Linux and POSIX based system calls. However, our implementation is compatibility to any real-time OS that is POSIX compliant.

There are many ways to implement the inter-core communication and it depends on the amount of data needed to be shared. For small communication messages, an intuitive approach is to use a portion of LLC and avoid accessing the main memory [8]. However, the implementation of locking chunks of messages in the LLC requires specific hardware support. This paper focuses on the scenarios where messages are large and hardware support for locking[2] LLC is not available. The main contributions of this paper include the following:

- A novel CCM that bounds the amount of contention on the DRAM banks for implementing shared memory communication inside the SCE framework is proposed.
- A mathematical expression on how to calculate WCET of a task under the proposed CCM is provided.
- Implementation details of the communication library (CommLib) and a communication task (CT) based on the proposed CCM are provided.
- An extensive evaluation of the proposed CCM is provided is compared with CBC where the inter-core communication is implemented without private banks.

The rest of this paper is organized as follows, Section 2 introduces the related work and background. Section 3 introduces the system model and assumptions. Section 4 presents how to bound the inter-core memory interference in SCE with the proposed CCM. Section 5 presents the delay analysis of the proposed system. Section 6 describes the details about the implementation of the proposed library and communication server. Section 7 presents the analytical results of the CCM and the CBC approach and provides the measurement-based results for CCM on the P4080 platform. Finally, Section 9 concludes our work.

2. Related work and DRAM background

This section is divided into various subsections. Section 2.1 describes the background related to the DRAM memory controller and some of the previous works proposed by the research community to analyze and predict the DRAM memory controller's behavior. In particular, we describe the work proposed in [9] that we use as a basis to analyze our proposed system model. Next, in Section 2.2 we describe the necessary background and related work required for the understanding of this paper.

2.1. DRAM background and related work

Main memory such as DRAM is a shared resource in a multicore processor, which greatly affects the system's overall performance. DRAM memory controllers are designed to generate DRAM specific commands to access data in the DRAM.

[2] In ARM Cortex-A9 platform, the cache locking feature is supported, but in ARM Cortex-A53 platform, this feature does not exist.

The DRAM controller and the DRAM module are connected through a command/address bus and a data bus. The DRAM memory is generally organized into a set of ranks; each rank is divided into multiple banks that can be accessed in parallel, provided that no collision occurs on either bus. Each bank has a two-dimensional array of memory organized into rows and columns. An activate (ACT) command must be issued to load the data in the row buffer to access the data in a row. Once loaded, all the read/write memory requests (CAS) accessing the row buffer data will constitute a row hit. However, if a memory request targets a different row, then the current buffer must be written back to the array with a pre-charge command (PRE) first before the second row can be activated. A request that is a hit in the row buffer (open row access) takes a much shorter time than that is a miss in the row buffer (close row access). The minimum time it takes to complete open row access, and close row access is device-dependent. Once the DDR device for the system is selected, the timing constraint values can be found on JEDEC standard documents, such as [10].

Scheduling algorithm in COTS memory controllers have been developed to offer low average-latency and maximize the throughput. One of the most common scheduling algorithms is the First-Ready First-Come-First-Serve (FR–FCFS) scheduling algorithm that prioritizes: (1) row-hit over row conflicts; (2) old commands over newer commands in case of row conflicts. Another widely used scheduling algorithm is round-robin.

In the real-time community, there has been a great effort to analyze the DRAM memory controller's behavior. The complexity of the COTS DRAM systems, however, has made such efforts difficult. To address such complexity, researchers have chosen to design hardware (HW) real-time DRAM controllers [11–17] that are easier to analyze. The problem with these HW real-time DRAM controllers is that they have lower performance than the COTS DRAM controllers. Moreover, these HW DRAM controllers have yet to be adopted by the industry. Thus, there is a need to analyze the DRAM memory controller accompanied by modern COTS processors.

To analyze COTS-based memory controllers' memory behavior, some researchers have proposed to model the main memory as a black box where each request from the memory controller takes a specific amount of time, and memory requests from other cores are serviced in a round-robin or first-come-first-serve (FCFS) basis. A variety of previous works addressing main memory as a black box include: [18–22]. However, the memory model assumed in these approaches is not safe for COTS multicore platforms because it hides critical details necessary to place an upper bound on its timing [23].

Recently, the authors in [9] proposed a new approach for bounding memory interference. In their approach, they considered the timing characteristics of major resources in the DRAM system, including the reordering effect of FR–FCFS and the rank/bank/bus timing constraints. Using this approach, the authors showed that they obtain a tighter upper bound on the worst-case memory interference delay for a task when it executes in parallel with other tasks. The presented technique combined two approaches: a **request-driven** and a **job-driven** approach. The request-driven approach focuses on the task's own memory requests, and the job-driven approach focuses on interfering memory requests during the task's execution. Combining two approaches yields a tight upper bound on the worst-case response time of a task in the presence of memory interference.

2.2. Background on partitioning shared resources

In a multicore system, there are shared resources such as available bus bandwidth and DRAM banks. These shared resources can be partitioned among the cores to avoid conflicts. Researchers in the real-time community have introduced OS-based techniques to regulate access to the shared resources to bound the inter-core interference. For example, memory regulation techniques such as [1] proposed to regulate the amount of main memory access by each core in each

regulation periods to reduce the interference on the memory controller. Other researchers proposed to partition the shared resource to reduce the inter-core interference channels. Techniques such as [3,7] partition the DRAM banks in the shared main memory among cores. While other techniques such as [2,24,25] partitions the shared last level cache (LLC) to prevent cache evictions caused by inter-core interference.

In this paper, we use approaches similar to MemGuard [1] and PALLOC [3] to partitioned the shared resource in our system. However, the resource partitioning can also be achieved using the MC^2 work in [7,8].

MemGuard is a memory bandwidth reservation mechanism that is implemented at the Operating System (OS) layer. This mechanism aims to distribute the bandwidth available from the memory controller equally among all the cores. It works periodically, and for each interval, e.g., 1 ms, a fixed memory budget (Q_p) is assigned to each core. During each period, the hardware performance counter (PMC) on each core measures the number of memory requests generated by the core. The PMC is programmed to generate an overflow interrupt to the core once its assigned budget has been exhausted. Upon the reception of the overflow interrupt, MemGuard stalls the core by dequeuing all the tasks running on that core. At the beginning of the new period, a new budget assignment occurs. At the budget replenishment, previously dequeued tasks are scheduled again.

DRAM memory module contains multiple resources (banks) that can be accessed in parallel. In COTS multicore platforms, banks are typically shared among all the cores even though programs running on the cores do not share memory space. To partition the banks and assign each bank to a particular core, we rely on PALLOC. PALLOC allows partitioning of banks to avoid bank sharing among cores, thereby improving isolation on COTS multicore platforms without requiring any special hardware support. On P4080, we see a latency improvement of 1.6x times when we have different banks for each core.

3. System model and assumptions

3.1. Architectural/hardware assumptions

We assume a standard COTS-based multi-core processor with n cores. Each core in the system features a private cache. There is also a shared last-level cache (LLC). We also assume that the underlying main memory is a DRAM with B banks. CPUs access main memory through a shared interconnect. The platform provides a mechanism to measure the number of memory requests issued by each core to the main-memory. The platform is capable of counting aggregated read and write memory accesses. These assumptions are meet by various COTS based embedded platforms such as P4080 from NXP that we use in our evaluation, Intel Core2Quad Q8400 and many other platforms employ such hardware performance tools.

3.2. Proposed model

Using the hardware assumptions described in Section 3.1, we specifically partition the shared DRAM banks and the available memory bandwidth equally among all the cores. For simplicity, we partition the resource equally among all the cores. A system designer can always assign uneven partitioning of the shared resources depending upon the applications/workloads requirements. In our proposed CCM, out of n multi-core processors one core is dedicated for inter-core communication. This core is referred to in rest of the paper as Communication Core (CC). All the other cores are referred to as Application Cores (ACs). The ACs are only allowed to access their dedicated DRAM banks, whereas the CC is capable of accessing all DRAM banks. A block diagram of our proposed model is shown in Fig. 1.

In our proposed CCM the CC is responsible for copying data from the bank of one AC to the bank on another AC. The task responsible for this data movement is called communication task (CT). A summary of

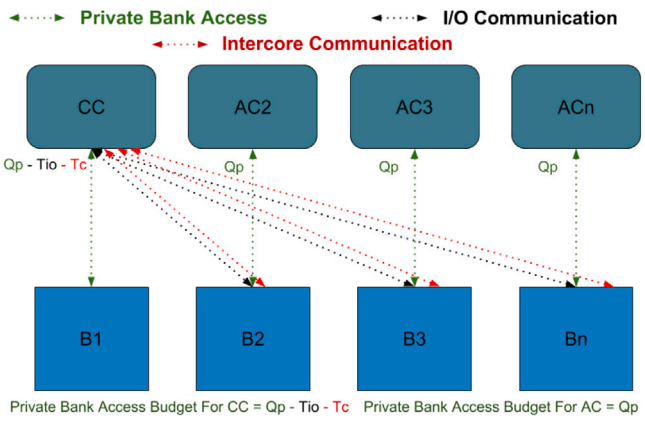

Fig. 1. Block diagram.

the system parameters and their values used for evaluations in Section 7 is provided in Table 1. Within each memory regulation interval, the CC is capable of accessing all the banks. There exist at most $(n-1)\cdot(n-2)$ communication sequences that need to be completed in one memory regulation period assuming all the ACs need to communicate with each other. For each pair of communicating cores, we assume CC issues at most t_c memory requests to the sender's private bank, and at most t_c memory requests to the receiver's private bank. The total number of memory requests made by the CC to banks of ACs during one memory regulation period is represented by $T_c = 2\cdot(n-1)\cdot(n-2)\cdot t_c$.

The CT is also responsible for communication between I/O devices and ACs. We specifically note that the proposed CCM is in accordance with the design principles of Integrated Modular Avionics (IMA) architecture. Originally, strict partitioning of shared resources in a multicore framework was designed to support the use of the standard IMA architecture on each core. The (single core) IMA architecture uses Time Division Multiplexing Access (TDMA) to run applications with different criticality in different partitions. Within each partition, tasks are scheduled by generalized rate-monotonic algorithm [26]. In IMA standard, the zero partition (I/O partition) is used to handle all the I/O and inter-partition message exchanges. Existing work [27] further proposed to consolidate the zero partitions from each core into a specific core, called I/O core, to manage the I/O accesses. It is natural to extend the I/O core architecture to implement inter-core communication using the model as shown in Fig. 1; here the CC takes the place of the I/O core, being responsible for moving I/O data between I/O devices and all the other ACs as well as the inter-core communication data between ACs.

More in details, using the CCM, one can handle the I/O data from I/O devices using the following two approaches: (i) either the communication core transfers data from/to device memory into its own private bank and move it from/to the private bank of AC that needs it; (ii) or the CC can directly transfer the data from the I/O device to the bank of the application core that needs it. For simplicity, we consider the second approach, shown as black arrows in Fig. 1. When an AC needs to access an I/O device buffer, CC issues at most t_{io} memory requests from the TX buffer in the sender's private bank (I/O output), and at most t_{io} memory requests to the RX buffer in the receiver's private bank (I/O input). The memory transactions required to move data to/from a device buffer to the private bank of ACs is represented by $T_{io} = 2\cdot(n-1)\cdot t_{io}$.

In summary, in each memory regulation period, the CC performs up to T_c memory transactions for inter-core communication, and up to T_{io} transactions for I/O transfers. The CC can then use the remaining regulation budget $(Q_p - T_c - T_{io})$ to execute tasks that access CC's own private banks. These tasks include OS related activities such as drivers, device bookkeeping and interrupt handling etc.

Table 1
System parameters.

System parameters	Symbol	Value
Number of cores	n	8
Number of ACs	$n-1$	7
Memory regulation period	P	1 ms

Table 2
Task parameters.

Description	Symbols
Core to which τ_i has been assigned	$Core_i$
Number of main memory requests of any job of task τ_i	H_i
Solo execution time	e_i
Period (and deadline)	T_i

3.3. Motivating example

In this subsection we provide a motivating example of our proposed model. The parameters used in this example are similar to what has been included in the evaluation section. Consider a system of eight cores ($n = 8$). Here one core is dedicated for communication purpose. The remaining seven cores are ACs. All the cores have their own DRAM bank. Let us assume that the minimum guaranteed bandwidth rate provided by the memory controller is computed experimentally using the approach in [1] and is found to be 1.2 GB/s. If we split the bandwidth equally among the cores then each of the core will get 153 MB/s. Let us assume that we have memory regulation implemented at the granularity of 1 ms. Given the minimum guaranteed bandwidth of each core is 153 MB/s, each core is assigned a Q_p of 2520 memory transactions per memory regulation period. Since the memory transactions are generated by the misses in the LLC, the transaction length is equal to the cache line size. The cache line size for the P4080 platform considered in our evaluation is 64 bytes. We assume same cache line size for this example.

For simplicity of this example, we assume that the whole memory budget is available to CC i.e. $T_c = Q_p$ and $T_{io} = 0$. These 2520 memory transactions will be divided equally between all the pairs of ACs. This gives us per-pair communication budget of $t_c = T_c/(2\cdot(n-1)\cdot(n-2)) = 30$ memory transactions. This translates to data size of 1920 bytes per memory regulation period. By assigning $t_c = 30$ memory transactions for one AC-pair, we can say that during each memory regulation period the maximum packet size that can be successfully transferred from the bank of one application core to the bank of another application core is 1920 bytes. In this example we assumed $T_c = Q_p$. However, in an actual OS implementation T_c is always less than Q_p. This is because some of the budget assigned to the CC is used for OS bookkeeping (such as I/O activity, interrupts handling etc.) activities. We empirically measure this overrhead in our evaluation.

3.4. Application task model

We consider a partitioned and fixed priority scheduling policy, where each core has a set Γ of N periodic application tasks, $\{\tau_1, \ldots, \tau_N\}$, each with different priority whereby τ_1 has the highest priority and τ_N has the lowest priority. Each task τ_i can be represented as $\tau_i = \{Core_i, H_i, e_i, T_i\}$. Where $Core_i$ is the core, τ_i is assigned to. H_i is the worst-case number of main memory accesses of τ_i. e_i is the worst-case execution time of the task measured in isolation, i.e., when no interference tasks are present and no memory regulation is enforced. The amount of communication data that a task needs to send to another task is included in H_i. T_i is the period of the task. The deadline of a task is equal to its period. Table 2 summarizes the task parameters.

An AT is a task deployed on an AC. It accesses only the private DRAM bank assigned to it. It only shares DRAM banks with ATs on the same core and the CT.

4. Bounding interfering memory requests in the proposed system

The maximum number of memory requests that each core can issue in a memory regulation period is Q_p. However, as discussed in [9,3], interfering memory requests that access the same bank (intra-bank interference) as the task under analysis produce more delay and lead to more pessimistic WCET, compared to memory requests that access different banks (inter-bank interference). In this section, we describe how to bound the interfering intra-bank (A^{intra}) and inter-bank memory requests (A^{inter}) in a memory regulation period according to the proposed CCM described in Section 3.

In order to calculate the total interference caused by the CC and all the ACs to the AC under analysis during a memory regulation period, we apply the following approach: first, we calculate the total interference caused by CC; second, we calculate the interference caused by all the ACs; and finally, we sum the two interferences to get the total interference.

4.1. Interference caused by CC

The amount of inter-bank and intra-bank interference caused by the CC in the CCM can be summarized as follows:

- The intra-bank interference (A^{intra}) caused by CC when moving I/O data to(input)/from(output) the bank under analysis. This intra-bank interference can be accounted as $2 \cdot t_{io}$.
- The inter-bank interference (A^{inter}) caused by CC when moving I/O data to(input)/from(output) other $(n-2)$ ACs. This (A^{inter}) can be accounted as $2 \cdot (n-2) \cdot t_{io}$.
- The intra-bank interference (A^{intra}) caused by CC when moving communication data to(receive)/from(send) the bank under analysis. This (A^{intra}) can be accounted as $2 \cdot (n-2) \cdot t_c$.
- The inter-bank interference (A^{inter}) caused by CC by moving communication data to(receive)/from(send) other ACs is $2 \cdot (n-2) \cdot (n-2) \cdot t_c$. This is due to the fact that CC accesses each private bank of an AC for at most $2 \cdot (n-2) \cdot t_c$ transactions, and there are $(n-2)$ banks belonging to other ACs that can cause inter-bank interference to the AC under analysis.
- The inter-bank interference (A^{inter}) caused by leftover CC budget after depletion of I/O and communication budget. This can be written as $Q_p - 2 \cdot (n-1) \cdot t_{io} - 2 \cdot (n-1) \cdot (n-2) \cdot t_c$

The total intra-bank and inter-bank interference caused by CC can be obtained by summing the various terms, as expressed in Eqs. (1) and (2) below. Note that the total memory interference caused by CC during a memory regulation interval is the sum of Eqs. (1) and (2) and is equal to the memory regulation budget (Q_p).

$$A^{intra}_{CC} = 2 \cdot t_{io} + 2 \cdot (n-2) \cdot t_c \tag{1}$$

$$A^{inter}_{CC} = Q_p - 2 \cdot t_{io} - 2 \cdot (n-2) \cdot t_c \tag{2}$$

4.2. Interference caused by other ACs to AC under analysis

In our proposed model, all the ACs only access their own bank with a memory regulation budget of Q_p. This means that the only interference introduced by all other ACs to the AC under analysis is inter-bank interference (A^{inter}).

The total intra-bank and inter-bank interference caused by all the ACs to the AC under analysis are expressed in Eqs. (3) and (4), respectively.

$$A^{intra}_{AC} = 0 \tag{3}$$

$$A^{inter}_{AC} = (n-2) \cdot Q_p \tag{4}$$

4.3. Total interference caused to AC under analysis

To obtain the total intra-bank interference caused by CC and the ACs to the AC under analysis, we simply add Eqs. (1) and (3) to obtain Eq. (5). Whereas, the total inter-bank interference can be obtained by adding Eqs. (2) and (4) to obtain Eq. (6).

$$\begin{aligned} A^{intra} &= A^{intra}_{CC} + A^{intra}_{AC} \\ &= 2 \cdot t_{io} + 2 \cdot (n-2) \cdot t_c \end{aligned} \tag{5}$$

$$\begin{aligned} A^{inter} &= A^{inter}_{CC} + A^{inter}_{AC} \\ &= (n-1) \cdot Q_p - 2 \cdot t_{io} - 2 \cdot (n-2) \cdot t_c \end{aligned} \tag{6}$$

From Eq. (5) we can see the value of A^{intra} is dependent on the system parameters t_{io} and t_c and that it is only a fraction of the overall memory regulation budget. This shows that the proposed CCM indeed reduces the intra-bank memory interference, compared to the CBC where we cannot use bank privatization while supporting inter-core communication in a strictly partitioned system.

In the CBC configuration, where intercore communication is implemented with no bank privatization. In the worst case we can have all cores issuing memory requests to the same bank. This results in a much higher intra-bank interference count as shown in Eqs. (7) and (8).

$$A^{intra}_{CBC} = (n-1) \cdot Q_p \tag{7}$$

$$A^{inter}_{CBC} = 0 \tag{8}$$

5. Response time analysis

The response time of a task or group of tasks in a memory regulated system is inflated compared to solo execution time because of: (1) memory contention caused by tasks on other cores; and (2) stall induced by memory regulation. During each memory regulation period, a core either makes Q_p memory accesses, exhausting all of its budget and being stalled, or it does not exhaust its full budget. We define a memory regulation period in which a core exhausts its full Q_p budget and is stalled because of regulation as a *stall period*; whereas, a period in which a core does not utilize its full memory regulation budget is defined as a *contention period*. During a regulation period, the faster a core exhausts its Q_p budget the more regulation delay it suffers. Hence, in the worst case we can assume that the core immediately performs Q_p memory accesses at the beginning of the period and is stalled for the entire period (P).

To compute the response time of the task in our proposed CCM model, we first measure the solo execution time of the task in isolation. The cores in our model are out-of-order; in the best case, the memory access latency can be hidden from the processor because in absence of data dependencies, the CPU pipeline will not stall waiting for the completion of a memory load. (i.e., the instruction level parallelism of the task is high). When measuring the execution time of task in isolation, it is not known how many memory requests generated by the task were reordered and overlapped with CPU instructions.

To obtain a safe upper bound to the total response time, one can simply assume that all memory requests had zero latency when measured solo, while all requests experience full memory latency: there is no MSHR available, or no instruction that can be reordered to hide the latency of this memory request, and access close rows when considering interference from other cores.

In order to compute the response time analysis of a task in the proposed CCM we thus proceed as follows:

(1) Similar to [28], for each task τ_j, we define a pure computation time c_j as the execution time of the task minus the minimum latency for the H_j memory requests of the task. As discussed above, the minimum latency is zero, therefore the pure computation time equals to the measured execution time ($c_j = e_j$).

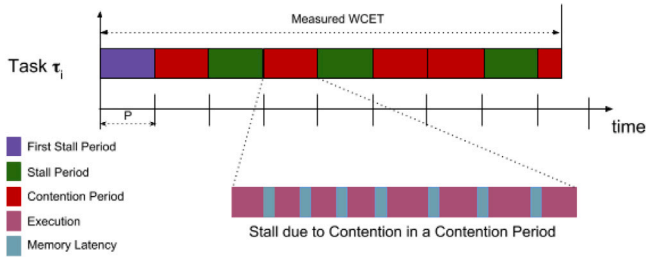

Fig. 2. Breakdown of measured WCET for a generic task with term of stall periods, contention periods. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

(2) Then, when considering the tasks that execute in the busy interval, we add an extra latency term P for each stall period (since they are just stalled for the whole period, without computation in the worst case). For memory requests issued in contention periods, we instead add a latency term that represents the maximum cumulative latency of all such requests (including the effects of contention). Let us call RL the total latency for stall periods, and CL the total memory latency (including contention effects) for contention periods. We also define the total memory latency as $ML = RL + CL$.

We can then perform response time analysis [29] of independent periodic tasks by computing a level-i busy interval as follows: Let R_i be the response time at the previous iteration. We define:

- $\bar{H}_i = \sum_{\forall j, j \leq i} H_j \cdot \lceil \frac{R_i}{T_j} \rceil$ as the total number of memory requests performed by all tasks on core under analysis that arrive in the interval R_i (including task under analysis).
- $\bar{c}_i = \sum_{\forall j, j \leq i} c_j \cdot \lceil \frac{R_i}{T_j} \rceil$ as the total computation performed by all tasks on core under analysis that arrive in the interval R_i (including the one under analysis).

We then compute R_i for the next iteration as:

$$R_i \leftarrow P + \bar{c}_i + ML(R_i, \bar{H}_i), \tag{9}$$

and continue to iterate until convergence, or $R_i > T_i$. Note that we are summing the total computation with the overall latency. We also need to add, however, an extra term P to account for the fact that the critical instant might start right after the memory regulation budget has been exhausted (by previous tasks not in the busy interval). The challenge is how to compute ML at each iteration, which we discuss in Section 5.2. As reported in the equation, we will show that ML is a function of the length of the busy interval R_i, and the total number of memory requests $\bar{H}_i$.

Fig. 2 gives an example of the breakdown of a measured WCET of a task in a memory regulated system. In Fig. 2 we can see that the measured WCET can be broken down into 4 stall periods (blue and green blocks) and 5 contention periods (red blocks). Note that all the periods except the last one must take up an entire memory regulation period. The last memory regulation period of a task may finish before the end of the memory regulation period. Inside a contention period, the task executes and suffers memory latency due to contention. The first regulation period (blue block) represents the initial stall term due to the memory regulation budget being already exhausted when the task under analysis is activated. The sum of the last three stall periods (green blocks) in Fig. 2 is the RL of the example task. The sum of all the memory latency (light blue blocks) within each of the 5 contention periods in Fig. 2 is the CL of the example task. The total memory latency (ML) is the sum of RL and CL.

5.1. Contention latency

Before detailing how to derive the total memory latency ML, we need to discuss the contention latency CL. In general, the contention

latency is a function of the number of memory requests, as well as the DRAM device being used, the behavior of the DRAM controller, and the employed latency analysis, as discussed in Section 2.1. In this paper, we adopt the Job-Driven delay analysis proposed in [9]. We discuss only Job-Driven delay because the Request-Driven delay analysis leads to extremely pessimistic bounds for out-of-order execution cores [30]. Based on [9], contention latency is a function of three parameters:the number of memory requests issued by the core under analysis (which we denote with J), the interfering memory requests issued from other cores targeting the bank accessed by the core under analysis (I^{intra}), and the number of interfering memory requests issued by other cores targeting banks that the core under analysis does not access (I^{inter}). Thus, we write $CL(J, I^{intra}, I^{inter})$ to denote an upper bound on the cumulative memory latency of J requests of the core under analysis. We now show how to derive $CL(J, I^{intra}, I^{inter})$ based on the analysis in [9]. It is important to note, however, that any memory analysis able to derive such a function could be used instead, without any change to the rest of the analysis. Hence, our general framework is independent of the specific characteristics of the DRAM controller being used, as long as inter-bank requests cause less interference than intra-bank requests.

Based on the observations in [9], the worst case access latency for a close-row memory access, due to row conflict caused by a previous access to the same bank, can be expressed as L_{conf}. Since conflicting accesses to the same bank cannot proceed in parallel, an interfering intra-bank memory access can at most cause L_{conf} delay to the core under analysis. An inter-bank memory interference can at most cause $L_{inter} = L^{PRE}_{inter} + L^{ACT}_{inter} + L^{RW}_{inter}$ delay to the core under analysis. Where L^{PRE}_{inter}, L^{ACT}_{inter}, and L^{RW}_{inter} can be derived from DRAM device timing constraints, as discussed in [9].

Assume there are I^{intra} interfering memory accesses to banks that the core under analysis can access, and there are I^{inter} interfering memory accesses to banks that the core under analysis cannot access. Then, the time taken by the core under analysis to perform J memory accesses with an FR–FCFS scheduling algorithm is bounded by:

$$CL(J, I^{intra}, I^{inter}) = (J + I^{intra}) \cdot L_{conf} + I^{inter} \cdot L_{inter}. \tag{10}$$

First we consider the case where $I^{inter} = 0$. The worst case memory latency for a system with out-of-order processors is when there is no reordering and overlapping instructions available in the micro architecture so that the memory latency cannot be concealed from the processor. As the authors of [9] observed, every intra-bank memory request suffers a worst case latency of L_{conf} due to bank conflict; Hence, $CL(J, I^{intra}, 0) = J \cdot L_{conf} + I^{intra} \cdot L_{conf}$ is the time it takes for a DRAM bank to serve $J+I^{intra}$ memory requests when all the consecutive accesses are row-conflicts and the memory latency is not optimized by out-of-order processors.

Then, we consider the case when there are $I^{inter} > 0$ inter-bank memory accesses. Based on the inter-bank interference delay derived in [9], each inter-bank memory interference causes at most L_{inter} additional delay to a memory transaction accessing another bank because of the timing constraints defined in the specifications [10]. Therefore, I^{inter} inter-bank memory accesses create at most $I^{inter} \cdot L_{inter}$ memory delay.

By combining the two cases, we derive Eq. (12).

The value of L_{conf} and L_{inter} are related to the DRAM device timing parameters. When a specific DRAM device instance is selected, these values can be treated as constants. Throughout this paper we use DDR-1333H as the selected device, based on the timing constraints defined in [10], we have $L_{inter} = 37.5$ ns, $L_{conf} = 58.5$ ns. We refer interested readers to [9] for the details on how to derive the value of L_{conf} and L_{inter} from the DRAM timing constraints.

5.2. Latency computation

Based on the function $CL(J, I^{intra}, I^{inter})$, we now detail how to determine $ML(R_i, \bar{H}_i)$. Given a response time (R_i), the number of memory regulation periods that the tasks in the busy interval extend on (excluding the first one that is fully stalled due to previous tasks) is equal to $K = \lceil (R_i - P)/P \rceil$. As explained earlier in this section, out of these K periods, some are regulation, and some are contention. Let us call K^{reg} the number of regulation periods, and K^{cont} the number of contention ones. Since we do not know the number of such intervals that results in the worst case latency (ML), we will treat K^{reg} and K^{cont} as variables and use them to write an optimization problem with the objective of maximizing ML. Clearly, it must hold by definition: $K^{reg} + K^{cont} = K$.

Similarly, we can split the memory requests of the tasks in the busy interval between requests in regulation periods and contention periods. Let us call $\bar{H}^{reg}_i$ and $\bar{H}^{cont}_i$ the requests in regulation and contention periods, respectively. We then also have by definition: $\bar{H}^{reg}_i + \bar{H}^{cont}_i = \bar{H}_i$. Furthermore, since we need to have Q_p memory requests for each regulation period, it must also hold: $\bar{H}^{reg}_i = K^{reg} \cdot Q_p$. That implies: $K^{reg} \leq \lfloor \bar{H}_i / Q_p \rfloor$, and $\bar{H}^{cont}_i = \bar{H}_i - K^{reg} \cdot Q_p$.

We can now derive the latency. Given K^{reg} regulation periods, the regulation latency is simply: $RL(K^{reg}) = K^{reg} \cdot P$. For the contention latency, note that since we have K^{cont} contention periods, there are a total of $I^{intra} = A^{intra} \cdot K^{cont}$ and $I^{inter} = A^{inter} \cdot K^{cont}$ intra-bank and inter-bank memory requests, respectively (based on Eqs. (5), (6) derived in Section 4). We can plug in the values we obtained in the CL function to obtain a contention latency: $CL(\bar{H}^{cont}_i, A^{intra} \cdot K^{cont}, A^{inter} \cdot K^{cont})$. Finally, summing the regulation and contention latencies yields Eq. (11):

$$\begin{aligned} ML(R_i, \bar{H}_i) &= RL(K^{reg}) \\ &\quad + CL(\bar{H}^{cont}_i, A^{intra} \cdot K^{cont}, A^{inter} \cdot K^{cont}) \\ &= K^{reg} \cdot P \\ &\quad + CL(\bar{H}_i - K^{reg} \cdot Q_p, A^{intra} \cdot (K - K^{reg}), \\ &\quad A^{inter} \cdot (K - K^{reg})) \end{aligned} \tag{11}$$

Finally, we need to discuss the contention latency CL. In general, the contention latency is a function of: (1) how many memory requests the core under analysis makes during contention periods, which is $\bar{H}^{cont}_i$; (2) the number of memory requests made during contention periods by other cores.

In this paper, we adopt the Job-Driven delay analysis proposed in [9] as the CL function. We discuss only Job-Driven delay because the Request-Driven delay analysis leads to pathologically pessimistic bound for out-of-order execution cores [30]. Based on [9], the CL is a function of three parameters: the number of memory requests issued by the core under analysis (J), the interfering memory requests issued from other cores targeting the bank that core under analysis accesses (I^{intra}), and the number of interfering memory requests issued by other cores targeting the banks that core under analysis does not access (I^{inter}).

Based on the observations in [9], the longest time it takes for a close row memory access with conflict can be expressed as L_{conf}. An intra-bank memory interference can at most create L_{conf} delay to the core under analysis. An inter-bank memory interference can at most create $L_{inter} = L^{PRE}_{inter} + L^{ACT}_{inter} + L^{RW}_{inter}$ delay to the core under analysis.

Assume there are I^{intra} interfering memory accesses to the banks that the core under analysis can access, and there are I^{inter} interfering memory accesses to the banks that the core under analysis cannot access. Then, the time taken by the core under analysis to perform J memory accesses with a work-conserving FR–FCFS scheduling algorithm is bounded by the CL function as expressed in Eq. (12),

$$CL(J, I^{intra}, I^{inter}) = J \cdot L_{conf} + I^{intra} \cdot L_{conf} + I^{inter} \cdot L_{inter} \tag{12}$$

First we consider the case where $I^{inter} = 0$. The worst case memory latency for a system with out-of-order processor is when there

is no reordering and overlapping instructions available in the micro architecture so that the memory latency cannot be concealed from the processor. As authors in [9] observed, every interfering intra-bank memory request can cause at most L_{conf} memory delay; Hence, $CL(J, I^{intra}, 0) = J \cdot L_{conf} + I^{intra} \cdot L_{conf}$ is the time it takes for a DRAM bank to serve $J+I^{intra}$ memory requests when all the consecutive accesses are row-conflicts and the memory latency is not optimized by out-of-order processors. This is the bound for memory access time when only one bank is accessed during the busy interval.

Then, we consider the case when there are $I^{inter} > 0$ inter-bank memory accesses. Based on the inter-bank interference delay proposed in [9], each inter-bank memory interference causes at most L_{inter} delay to a memory transaction accessing another bank because of the timing constraints defined in the specifications [10]. Therefore, I^{inter} inter-bank memory accesses create at most $I^{inter} \cdot L_{inter}$ memory delay.

By combining the two cases, we derive Eq. (12).

The value of L_{conf} and L_{inter} are related to the DRAM device timing parameters. When a specific DRAM device instance is selected, these values can be treated as constants. Throughout this paper we use DDR-1333H as the selected device,[3] based on the timing constraints defined in [10], we have $L_{inter} = 37.5$ ns, $L_{conf} = 58.5$ ns. Readers are encouraged to refer to [9] for the details on how to derive the value of L_{conf} and L_{inter} from the DRAM timing constraints.

In summary, at each iteration performed to compute the task response time we need to solve the following optimization problem to determine ML:

Maximize (over variable K^{reg}):

$$K^{reg} \cdot P + CL(\bar{H}_i - K^{reg} \cdot Q_p, \\ A^{intra} \cdot (K - K^{reg}), A^{inter} \cdot (K - K^{reg})) \tag{13}$$

Subject to:

$$0 \leq K^{reg} \leq \min\left(K, \left\lfloor \frac{\bar{H}_i}{Q_p} \right\rfloor\right) \tag{14}$$

For a general CL function, one could try all possible values of K^{reg} subject to constraint in Inequality (14) and find the one that maximizes Eq. (13). However, when employing the CL in Eq. (12), the problem can be simplified: Note that in this case Eq. (13) is equivalent to: $K^{reg} \cdot \left(P - (Q_p + A^{intra}) \cdot L_{conf} - A^{inter} \cdot L_{inter}\right) + (\bar{H}_i + A^{intra} \cdot K) \cdot L_{conf} + A^{inter} \cdot K \cdot L_{inter}$. Hence, if $P - (Q_p + A^{intra}) \cdot L_{conf} - A^{inter} \cdot L_{inter}$ is positive, then ML is maximized by taking the maximum value $K^{reg} = \min\left(K, \left\lfloor \frac{\bar{H}_i}{Q_p} \right\rfloor\right)$; otherwise, by taking the minimum $K^{reg} = 0$.

6. Implementation

For proof-of-concept, we implement our CommLib using POSIX APIs on Linux because of its ease to use, open source and easy portability. We know that Linux is not real-time OS. However, for proof-of-concept it is a fair choice. Our implementation is still valid for any POSIX compliant real-time OS. As explained that for I/O data either the CC directly transfers data from/to device memory into its own private bank and move it from/to the private bank of AC that needs it; or the CC can directly transfer the data from the I/O device to the bank of the application core that needs it. In this paper, we are not concerned about the movement of I/O data and communication between the CC and ACs. The rest of this section describes inter-core communication between ACs using proposed CCM.

As depicted in Fig. 3(a), when a task running on one AC wants to send data to another task running on a different AC, it writes the data to sending (TX) buffer in its private DRAM bank. In Fig. 3(a), the TX buffer that stores outgoing messages from ACi to ACj is named TX_i_j. It should be noted that all the tasks on an AC sending data to the other

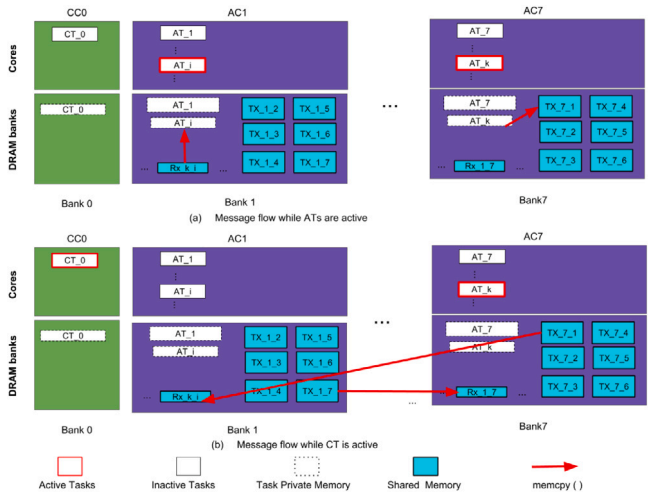

Fig. 3. Message flow diagram.

receiving (RX) tasks on a particular destination AC would write to same TX buffer. For instance, Fig. 3 shows that AC1 has separate TX buffers to send to different ACs. The situation is symmetric on the other cores. The main reason for having separate TX buffers per AC pair is to reflect the fact that we assign t_c for each AC pair.

For the receiver task there is a separate RX buffer for each pair of communicating ATs. We name the RX buffer that stores the incoming messages from AT_k to AT_j as RX_k_j. The data from the TX buffer is copied into the RX buffer of a destination AT in another AC using the CC, as depicted in Fig. 3(b). The TX and RX buffers are non-cacheable to the ACs. In the next subsection, we provide the details of how the TX buffer and RX buffers are implemented.

The TX/RX buffers are created/implemented in the private banks of ACs using POSIX *shm_create()*. The CT as a part of the initialization process creates these buffers. The buffers are mapped to the ATs running on ACs using *mmap()*. All ATs that need to send inter-core messages to receiving ATs need to access the corresponding TX buffer in their dedicated bank as shown in Fig. 3. The receiving ATs access their local RX buffers to read any data produced by ATs on a different core. In order for the ATs running on the ACs to access TX/RX buffers we have implemented a shared library, named CommLib.

We assume that there is a system configuration file, provided by the system administrator, that specifies all the possible inter-core communication channels, message sizes, and periods, between the ATs in the system. Based on parameters recorded in the system configuration file, the TX/RX buffers are created and initialized with appropriate size so that the buffers will never overflow as long as all ATs use the library according to the parameters recorded in the configuration file. When the CT and the ATs that use the CommLib initialize, they read the same configuration file to obtain the names of the buffers they interact with, and stores the list of buffers along with other metadata in their own local data structure. The ATs use CommLib to write/read data to/from the TX/RX buffers. The CT running on CC has access to all the TX/RX buffers. As discussed in Section 3, all the TX and RX buffers are mapped to be non-cacheable. In our implementation, we make the buffers non-cacheable by modifying the *mmap()* system call so that we can make the tasks in our system always access the TX/RX buffers as non-cacheable.

As described in earlier subsections, an inter-core communication budget (t_c) is assigned for each pair, therefore we implemented a TX buffer for each AC-pair in our proposed CCM. The TX buffer is shared by the CT and all the ATs running on the same AC that want to send data to a specific AC. Hence, access to the shared data structure needs to be protected to avoid race conditions. To reduce the long blocking times for tasks accessing the TX buffer, we propose the use of two

[3] This is the DDR device used in the evaluation platform P4080.

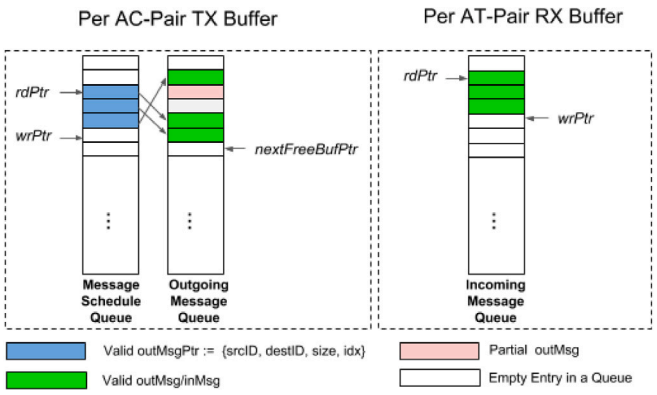

Fig. 4. Per AC-pair TX buffer and per AT-pair RX buffer.

circular buffers, as the **Message Schedule Queue** and the **Outgoing Message Queue** shown in Fig. 4. Using two circular buffers results in less blocking. In fact in this case, it is enough to acquire a mutex only for the amount of time required to update the metadata of the TX buffer, rather than for the entire duration of a send operation. The **Outgoing Message Queue** in Fig. 4 is used to store the actual TX packet data sent. The sent data is written to a free memory location pointed in the next free entry in the queue (*nextFreeBufPtr*). The data written to the *nextFreeBufPtr* location can be less than or equal to the packet size supported by our CCM as described in Fig. 4.

The pseudo code of the send API that takes *txTaskID*, *rxTaskID*, pointer to the *txData* and *size* is shown in Algorithm 1. Based upon the *txTaskID* and *rxTaskID* passed in the send API, an array of metadata holding information about all the TX buffers that the current AT may access, and their corresponding metadata are searched to find the correct TX buffer (*txBufferPtr*) to which the send data must be written to, as shown in line 2 of Algorithm 1. Once the correct TX buffer has been identified the task tries to acquire the mutex. There can be multiple ATs that call send and try to write to the same TX buffer. Therefore, synchronization is required in the form of a mutex lock.

Once a lock has been acquired the send procedure saves the current *nextFreeBufPtr* in the *temp* variable, increments the *nextFreeBufPtr*, and releases the lock. The sent data is then copied to the address pointed by *temp* (see lines 5 through 9 in Algorithm 1). After data copy has been completed via the *temp* pointer, the address in the *temp*, along with other metadata such as *txTaskID*, *rxTaskID* and *size*, have to be stored into the **Message Schedule Queue**. The **Message Schedule Queue** is also shared between all the ATs that access the same TX buffer. As such, the send procedure acquires a lock on the metadata of the **Message Schedule Queue**. The metadata of the **Message Schedule Queue** are *rdPtr* and *wrPtr*. The only metadata that needs locking as a part of the send call is *wrPtr*. After a lock has been acquired on the metadata of the **Message Schedule Queue** the *temp* pointer is written at the *wrPtr*, *wrPtr* is then incremented and the lock is released (line 10 to 13 in Algorithm 1). The CT only reads *wrPtr* to determine if the queue is full, it never updates the value of *wrPtr*, therefore it does not have to acquire the mutex. Note that in our implementation, the critical sections contain only an update for the shared pointers. Therefore, the blocking time between ATs due to synchronization is short and is independent of the packet size. In addition, no synchronization between the CT and the ATs is required.

For the RX API, we create per AT-pair RX buffers so that the interference among all the receiving tasks can be minimized. Each RX buffer is only shared between the CT and a single receiving task. Therefore, a **Incoming Message Queue** with a *rdPtr* and a *wrPtr* is implemented. Since only the RX AT updates the *rdPtr*, whereas the CT only updates the *wrPtr*, there is no mutex required at the RX buffer.

The pseudo code of the receive API is shown in Algorithm 2. Similar to the send API, the receive API searches all the RX buffers linked to this task as shown in line 2 of Algorithm 2. Upon match, it checks if there is new data in the RX buffer by comparing the *rdPtr* and *wrPtr* pointers as shown in Fig. 4. Since we design the receive to be non-blocking, in case no new data is found in the receive buffer, the call returns -1 as shown in line 3 and 4 of pseudo code in Algorithm 2. Each receiving task has its own **Incoming Message Queue**s, no synchronization is required. Lines number 5 through 7 in Algorithm 2 describe this. When there is an incoming message in the queue, it is read into the buffer pointed by *rxData* passed to the receive API. The *rdPtr* of RX buffer is incremented. The number of bytes being read is returned.

Note that both the send and the receive interact with the buffers on the caller AT's private bank, no inter-bank memory interference is introduced by these functions.

The CT running on CC has its per AC-pair communication bandwidth replenished every memory regulation period (P). It then iterates over all the TX buffers in the private banks of all the ACs. For each TX buffer, based on the sender and receiver information contained in the **Message Schedule Queue**, the CT is responsible for copying the data: from the **Outgoing Message Queue** to the **Incoming Message Queue** of corresponding RX buffer in the private DRAM bank of the RX core. When the **Message Schedule Queue** is empty, or the communication bandwidth for this particular TX buffer is exhausted, the CT moves to the next TX buffer. After all the TX buffers have been processed, the CT sleeps for the rest of the regulation period.

```
send(txTaskID, rxTaskID, txData, size)
    txBufferPtr : = findtxBuffer(txTaskID,rxTaskID) ;
    if txBufferPtr.full() then
      | return -1 ;
    lock(txBufferPtr);
    temp : = txBufferPtr.nextFreeBufPtr ;
    Increment txBufferPtr.nextFreeBufPtr;
    unlock(txBufferPtr);
    memcpy (temp, txData, size);
    lock(txBufferPtr);
    txBufferPtr.wrPtr.idx := temp ;
    Increment txBufferPtr.wrPtr ;
    unlock(txBufferPtr);
    return size;
```

Algorithm 1: Pseudo Code For send API

```
receive(txTaskID, rxTaskID, rxData, size)
    rxBufferPtr : =findrxBuffer(txTaskID,rxTaskID) ;
    if rxBufferPtr.full() then
      | return -1; // No new data
    memcpy(rxData, rxBufferPtr.rdPtr, rxBufferPtr.size);
    Increment rxBufferPtr.rdPtr ;
    size := rxBufferPtr.size;
    return size;
```

Algorithm 2: Pseudo Code For receive API

7. Evaluation

This section provides a detailed evaluation of our proposed CCM and compares it with the CBC where no private bank is enforced, as described in Section 4. We start by describing the experimental setup and the benchmarks that we have used for evaluation. We then analytically show how different memory budget assignments (Q_p) impact the WCET. Next, we evaluate the communication bandwidth of the implemented CT based on our platform. Using the analysis discussed in Section 5, we then compare and show the benefit of CCM over the CBC approach for the considered benchmarks. Finally, we show that proposed analysis for CCM provides a safe WCET bound.

Table 3
SD-VBS benchmark solo measurements.

Benchmark	e_i (ms)	H_i	Memory access rate (1/ms)
disparity	318	4448615	13989
localization	244	668	3
mser	44	719914	16362
sift	521	2668107	5121
stitch	293	1588683	5422
svm	290	214138	738
texture_synthesis	25	42342	1694
tracking	176	289821	1647

7.1. Experimental setup and benchmarks

Our experimental setup considers P4080 platform from Freescale that employs eight Power Architecture e500mc cores operating at frequencies up to 1.5 GHz. Each core in the system has its dedicated 32 KB I/D Level 1 cache and a 128 KB Level 2 backside cache. A 2 MB of shared Level 3 cache is also present. As discussed in Section 3, we cannot formally prove that the considered HW platform is timing compositional; as is the case with most available COTS platforms, no precise micro architectural model is available. Therefore, in the paper we rely on an experimental evaluation based on measurements to show that the derived WCET provides safe bound. If such architectural model was available, we argue that the proposed communication scheme and analysis could still be applied after deriving tasks parameters (H_i and e_i) based on static program analysis [31].

A Linux-3.0.6 operating system that supports resource partitioning is installed on the evaluation platform. The task under analysis and all the stressing tasks are statically allocated to each core by $sched\_setaffinity()$. Only one DDR controller is enabled. For the proposed CCM, PALLOC [3] is enabled and configured so that all the ACs can only access one single private DRAM bank, while the CC can access all the DRAM banks. We use MemGuard [19] to enforce memory regulation on every core in the system, and every core is regulated by the same memory budget. memory regulation period is configured to 1 ms. and the memory regulation budget is 2520 memory transactions for each core. The 2520 memory transactions per MemGuard period correspond to a memory bandwidth of 153 MB/s per core.

For the proposed CCM, we consider a system with a single CC and 7 ACs. The WCET is obtained by using the equations derived in Section 5. The parameters used to compute WCET are listed in Table 1. The worst case scenario for CCM is when the task under analysis runs on an AC, while there are 6 interfering ACs, each issuing Q_p memory requests towards its own DRAM bank during every memory regulation period.

Whereas, a periodic CT is deployed on the CC and accesses private banks of each AC and generates communication memory traffic of T_c at every MemGuard regulation period. The CC is also using its remaining memory budget to stress its own bank.

In order to evaluate the system, we use San Diego Vision Benchmark Suite (SD-VBS) [32]. We use the on-chip event processing unit (EPU) provided by P4080 to profile the memory access counts (H_i) of each task under analysis. We measured the solo run time (e_i) and memory access count of the benchmarks with cif (352×240) input resolution on the evaluation platform. The memory regulation budget is set to a number that is larger than the available bandwidth, so no regulation is enforced. The measured parameters are listed in Table 3, the value is the maximum value observed of 200 instances on the platform.

7.2. Task WCET with different memory regulation budget assignment

As discussed in Section 5, in a memory regulated system, the WCET of a task is dependent on the memory budget assigned to the core. When the memory budget is small, the task tends to suffer more memory regulation and less memory contention from the interfering cores. Whereas, if the memory budget is large the task tends to suffer memory contention from the interfering cores rather than regulation [33]. Depending upon the characteristics of tasks, the optimal memory budget assignment for different tasks can be different.

Table 4
Budget distribution on CC.

Total budget assignment (Q_p)	2520
Average OS overhead	604
Communication budget (T_c)	1848
Metadata overhead (percentage)	13.6

We analyzed the WCET of the tasks in SD-VBS benchmark with various memory budget assignment for the proposed CCM. Fig. 5 shows the WCET of three selected tasks with various memory budget assignment. All the Q_p assignment are a multiple of 84 because there are 42 communication pairs among the 7 ACs, and each transaction compose a read and a write memory access. In this experiment we assume $T_c = Q_p$ for all the Q_p assignment so that the number of intra-bank inter-core interference (A^{intra}) is maximized and thus the bound is safe. Note that in real-world implementations, $T_c = Q_p$ cannot be achieved since the CC might be using some of the memory transactions for its local computation and the OS related overhead. More details are described in next subsection.

The inverted bell curves of disparity and tracking show that the WCET of the tasks under analysis increase rapidly when the assigned memory budget is very small or very large. The budget assignment that determines the shortest WCET is different for the two tasks that gives the smallest WCET are different. $disparity$ has smallest WCET when the memory budget is around 2520 while $tracking$ has smallest WCET when the system has memory budget around 1344. $localization$ is a special case in Fig. 5. It is extremely computation intensive, the average memory access rate is 3 access per millisecond. The curve shows that it provides the smaller WCET when memory budget is smaller, since it is very unlikely that it can exhaust the memory budget and get regulated even with a extremely small memory budget.

A memory budget assignment that produces relatively small WCETs can be found experimentally. For example, authors in [34] discussed how to obtain better system performance by assigning uneven memory regulation budgets to different cores. The development of a near optimal memory budget assignment algorithm is beyond the scope of this paper.

For our experimental and analytically results, we pick a per-core memory regulation budget (Q_p) to 2520 which corresponds to the minimum guaranteed bandwidth as used in the previous research [28], and it provides a reasonable WCET for the benchmarks we evaluated.

7.3. Throughput of the CT

Considering the system parameters in Section 7.1 for the considered P4080 platform with CCM. When assigning a $Q_p = 2520$ to CC in our implementation some of the memory transactions are used by the CC to manage the OS related overhead. Table 4 summarizes the distribution of Q_p on the CC. With CT not deployed on CC. We find out that on average 604 memory transactions on CC within a memory regulation period are used to deal with OS related overhead. This indicates that in our implementation the maximum value of T_c available to CT is $Q_p - 604 = 1916$. In our evaluation, we pick a value of $T_c = 1848$ because it is the exact multiple of 84 that does not exceed the maximum available T_c. Using a communication budget of 1848, the actual amount of memory transactions used to move the data between different pairs of ACs are 1596. This means around 13.6 percent of memory transactions issued by CT are used in dealing with the metadata. The memory transactions of 1596 per memory regulation period can move data at a rate of 389 Mbps between all pairs of ACs.

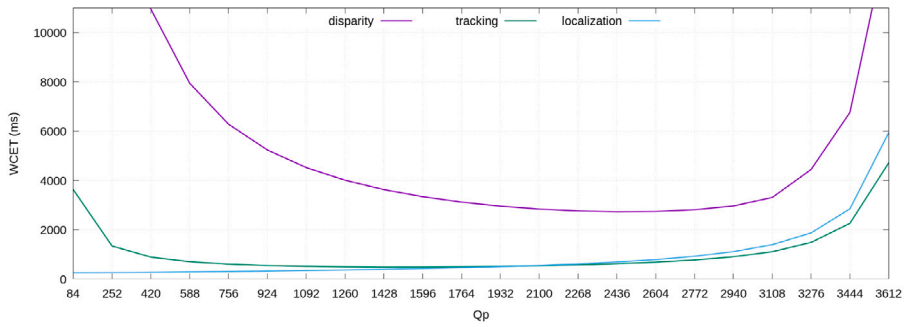

Fig. 5. WCET with different memory regulation budget assignments.

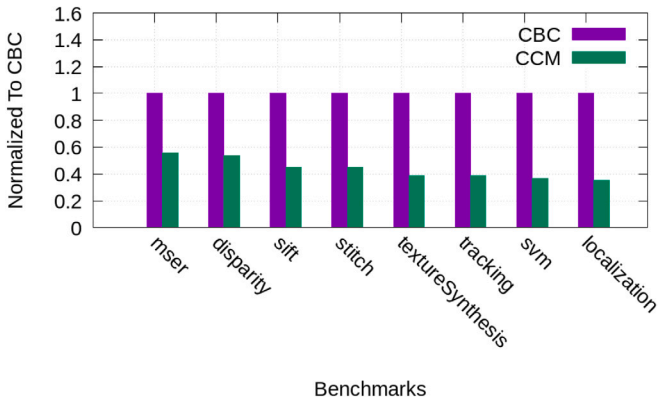

Fig. 6. WCET of tasks in CBC and CCM.

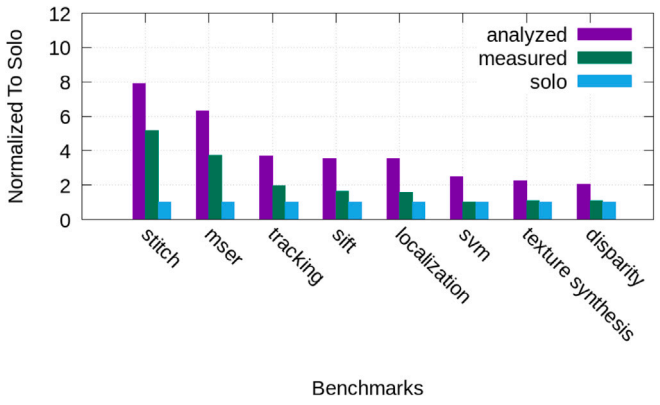

Fig. 7. Analyzed and measured WCET.

7.4. CCM and CBC

In this section, we compare the WCET of tasks deployed on the target P4080 platform with our proposed CCM versus the one with CBC that does not employ private bank.

The WCET of tasks in the CCM is obtained by assigning a Memguard budget of Q_p to all the cores.

For simplicity, all the cores are assigned a budget of $Q_p = 2520$ in CCM. The six interfering ACs with their assigned budget stress their private banks. Out of the total budget of $Q_p = 2520$, CC uses a communication budget of $T_c = 1848$ to move the data between all the pairs of banks used by ACs. Whereas, the remaining memory budget of $Q_p - T_c = 672$ is used by CC to access its own private bank. The WCET of the task under analysis is measured on the seventh AC that runs different benchmarks from SD-VBS.

For CBC, the WCET is obtained by considering the following worst case. The task under analysis runs in one AC, while 6 memory intensive interfering ACs stress the memory, each with all its memory budget. The ACs are assigned the same Memory Budget ($Q_p = 2520$) as in the CCM experiment. The CC is assigned memory budget of 0 and stays idle. Since CC is not required in CBC scheme, we make it stay idle to get a fair comparison between the two approaches.

Since there is no private bank enforced in the CBC, the worst case scenario corresponds to the case in which, during the busy interval, the memory access of all the active cores are issued to the same DRAM bank and all the interfering memory access are considered to cause intra-bank contention delay. From Fig. 6 can see that for all the benchmarks, CCM provides a smaller WCET compared to CBC, with an average of 56% WCET reduction. For the *localization* benchmark, the WCET on CCM is reduced by 65% compared to on CBC.

7.5. Analytical bound and measurement

In this section we show that the proposed WCET bound for CCM is safe for the target platform. We configure the PALLOC and MemGuard to the parameters as described in Section 7.1. For the 6 interfering ACs, we run a memory intensive $bandwidth$ [1] benchmark to stress the private banks of the ACs. We also deploy a $CommTask$ on the CC to periodically access the private DRAM banks of all ACs to stress the memory controller with $T_c = 1848$ communication traffic at every regulation period.

The analytical and measured WCET of CCM normalized to solo runtime of the SD-VBS is shown in Fig. 7. The results in Fig. 7 show that the analyzed WCET safely bounds the execution time when measured on the platform.

8. Discussion and future work

In this section, we list some of the limitations of our work. First of all, the CCM model divides the memory bandwidth equally among all the ACs, this might not scale well as the number of cores increase. This is something we plan to address in our future work. Another issue is the pessimism in the theoretical bound we derived in this paper. The two major factors that contribute to the pessimism are: (1) We assume all the DRAM access of the task under analysis in the worst case hit a closed row in the DRAM bank and the latency is not optimized by the out-of-order micro-architecture, (2) We assume that the solo execution time measured contains only CPU executions, all the memory access are optimized away by the out-of-order processor. These assumptions helped greatly simplify our analysis and represent a conservative, safe upper-bound on real behavior of the system. However, we believe that the bound can be further improved by relaxing some of these assumptions. We also plan to integrate I/O and provide end-to-end system.

9. Conclusion

In this paper, we complete the strictly partitioned multi-core framework by bringing inter-core communication into the picture. For our evaluation, we considered two communication models that are CBC and CCM. Compared to the CBC where all the cores can access all the DRAM banks, the CCM where at most only two cores access any DRAM bank can help improve the worst-case system performance. This approach provides tighter upper bounds on the inter-core interference that can be easily factored into schedulability analysis. The presented results show the gain of CCM over the CBC. Moreover, our presented approach and model gives system level prospective of how to move networked single core processors into a single multi-core architecture without breaking the hard-real time requirements that need to be met within a single core.

Acknowledgments

The material presented in this paper is based upon work supported by the National Science Foundation (NSF), United States under grant numbers NSF CNS 1815891 and in part by NSF CNS 1932529 and NSF CNS 1815959. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF and other sponsors.

References

[1] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms, in: Real-Time and Embedded Technology and Applications Symposium, RTAS, 2013 IEEE 19th, IEEE, 2013, pp. 55–64.

[2] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, R. Pellizzoni, Real-time cache management framework for multi-core architectures, in: Real-Time and Embedded Technology and Applications Symposium, RTAS, 2013 IEEE 19th, IEEE, 2013, pp. 45–54.

[3] H. Yun, R. Mancuso, Z. Wu, R. Pellizzoni, Palloc: DRAM bank-aware memory allocator for performance isolation on multicore platforms, in: Real-Time and Embedded Technology and Applications Symposium, RTAS, 2014 IEEE 20th, IEEE, 2014, pp. 155–166.

[4] L. Sha, M. Caccamo, R. Mancuso, J.-E. Kim, M.-K. Yoon, R. Pellizzoni, H. Yun, R. Kegley, D. Perlman, G. Arundale, et al., Single Core Equivalent Virtual Machines for Hard Real—Time Computing on Multicore Processors, Tech. Rep., 2014.

[5] J. Herman, C.J. Kenna, M.S. Mollison, J. H., D.M. Johnson, RTOS support for multicore mixed-criticality systems, in: 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium, 2012, pp. 197–208.

[6] FAA position paper on multi—core processors, CAST-32A (rev 0), 2017, Accessed: 2017-10-16 https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32A.pdf.

[7] N. Kim, B.C. Ward, M. Chisholm, C.Y. Fu, J.H. Anderson, F.D. Smith, Attacking the one-out-of-m multicore problem by combining hardware management with mixed-critcality provisioning, in: 2016 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2016, pp. 1–12.

[8] M. Chisholm, N. Kim, B. Ward, N. Otterness, J. Anderson, F. Smith, Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems, in: 2016 IEEE International Real-Time Systems Symposium, RTSS'16, 2016.

[9] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, R. Rajkumar, Bounding memory interference delay in COTS-based multi-core systems, in: Real-Time and Embedded Technology and Applications Symposium, RTAS, 2014 IEEE 20th, IEEE, 2014, pp. 145–154.

[10] A. JEDEC, Va, USA, JESD79-3F: DDR3 SDRAM specification, 2012.

[11] B. Akesson, K. Goossens, M. Ringhofer, Predator: A predictable SDRAM memory controller, in: Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, ACM, 2007, pp. 251–256.

[12] L. Ecco, S. Tobuschat, S. Saidi, R. Ernst, A mixed critical memory controller using bank privatization and fixed priority scheduling, in: Embedded and Real-Time Computing Systems and Applications, RTCSA, 2014 IEEE 20th International Conference on, IEEE, 2014, pp. 1–10.

[13] S. Goossens, B. Akesson, K. Goossens, Conservative open-page policy for mixed time-criticality memory controllers, in: Proceedings of the Conference on Design, Automation and Test in Europe, EDA Consortium, 2013, pp. 525–530.

[14] Y. Krishnapillai, Z.P. Wu, R. Pellizzoni, A rank-switching, open-row DRAM controller for time-predictable systems, in: Real-Time Systems, ECRTS, 2014 26th Euromicro Conference on, IEEE, 2014, pp. 27–38.

[15] Z.P. Wu, Y. Krish, R. Pellizzoni, Worst case analysis of DRAM latency in multi-requestor systems, in: Real-Time Systems Symposium, RTSS, 2013 IEEE 34th, IEEE, 2013, pp. 372–383.

[16] J. Reineke, I. Liu, H.D. Patel, S. Kim, E.A. Lee, PRET DRAM Controller: Bank privatization for predictability and temporal isolation, in: Hardware/Software Codesign and System Synthesis, CODES+ISSS, 2011 Proceedings of the 9th International Conference on, IEEE, 2011, pp. 99–108.

[17] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, M. Schmidt, Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement, in: Real-Time Systems, ECRTS, 2014 26th Euromicro Conference on, IEEE, 2014, pp. 109–118.

[18] D. Dasari, B. Andersson, V. Nelis, S.M. Petters, A. Easwaran, J. Lee, Response time analysis of COTS-based multicores considering the contention on the shared memory bus, in: Trust, Security and Privacy in Computing and Communications, TrustCom, 2011 IEEE 10th International Conference on, IEEE, 2011, pp. 1068–1075.

[19] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, Memory access control in multiprocessor for real-time systems with mixed criticality, in: Real-Time Systems, ECRTS, 2012 24th Euromicro Conference on, IEEE, 2012, pp. 299–308.

[20] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, L. Thiele, Worst case delay analysis for memory interference in multicore systems, in: Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association, 2010, pp. 741–746.

[21] S. Schliecker, M. Negrean, R. Ernst, Bounding the shared resource load for the performance analysis of multiprocessor systems, in: Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association, 2010, pp. 759–764.

[22] B. Andersson, A. Easwaran, J. Lee, Finding an upper bound on the increase in execution time due to contention on the memory bus in COTS-based multicore systems, ACM Sigbed Rev. 7 (1) (2010) 4.

[23] H. Yun, R. Pellizzon, P.K. Valsan, Parallelism-aware memory interference delay analysis for cots multicore systems, in: Real-Time Systems, ECRTS, 2015 27th Euromicro Conference on, IEEE, 2015, pp. 184–195.

[24] B.C. Ward, J.L. Herman, C.J. Kenna, J.H. Anderson, Outstanding paper award: Making shared caches more predictable on multicore platforms, in: 2013 25th Euromicro Conference on Real-Time Systems, ECRTS, IEEE, 2013, pp. 157–167.

[25] V. Suhendra, T. Mitra, Exploring locking & partitioning for predictable shared caches on multi-cores, in: Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE, IEEE, 2008, pp. 300–303.

[26] L. Sha, R. Rajkumar, S.S. Sathaye, Generalized rate-monotonic scheduling theory: A framework for developing real-time systems, Proc. IEEE 82 (1) (1994) 68–82.

[27] J.-E. Kim, M.-K. Yoon, R. Bradford, L. Sha, Integrated modular avionics (ima) partition scheduling with conflict-free i/o for multicore avionics systems, in: Computer Software and Applications Conference, COMPSAC, 2014 IEEE 38th Annual, IEEE, 2014, pp. 321–331.

[28] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, H. Yun, WCET(m) estimation in multi-core systems using single core equivalence, in: Real-Time Systems, ECRTS, 2015 27th Euromicro Conference on, 2015, pp. 174–183.

[29] J. Lehoczky, L. Sha, Y. Ding, The rate monotonic scheduling algorithm: Exact characterization and average case behavior, in: Real Time Systems Symposium, 1989, Proceedings, IEEE, 1989, pp. 166–171.

[30] R. Pellizzoni, H. Yun, Memory servers for multicore systems, in: Real-Time and Embedded Technology and Applications Symposium, RTAS, 2016 IEEE, IEEE, 2016, pp. 1–12.

[31] S. Hahn, M. Jacobs, J. Reineke, Enabling compositionality for multicore timing analysis, in: Proceedings of the 24th International Conference on Real-Time Networks and Systems, ACM, 2016, pp. 299–308.

[32] S.K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, M.B. Taylor, SD-VBS: The San Diego vision benchmark suite, in: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, IEEE, 2009, pp. 55–64.

[33] G. Yao, H. Yun, Z.P. Wu, R. Pellizzoni, M. Caccamo, L. Sha, Schedulability analysis for memory bandwidth regulated multicore real-time systems, IEEE Trans. Comput. 65 (2) (2016) 601–614.

[34] R. Mancuso, R. Pellizzoni, N. Tokcan, M. Caccamo, WCET derivation under single core equivalence with explicit memory budget assignment, in: LIPIcs-Leibniz International Proceedings in Informatics, Vol. 76, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.