SCE-Comm: A Real-Time Inter-Core Communication Framework for Strictly Partitioned Multi-core Processors

Rohan Tabish[1+], Jen-Yang Wen[1+], Rodolfo Pellizzoni[2], Renato Mancuso[3], Heechul Yun[4], Marco Caccamo[5], and Lui Sha[1]
[1]University of Illinois at Urbana-Champaign, USA, {jwen11, rtabish, lrs}@illinois.edu
[2]University of Waterloo, CA, rpellizz@waterloo.ca
[3]Boston University, USA, rmancuso@bu.edu
[4]University of Kansas, USA, heechul.yun@ku.edu
[5]Technical University of Munich, USA, mcaccamo@tum.de
[+]These authors have equal contribution

***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 inter-core communication.**

In this paper, we propose Communication Core Model (CCM) that implements the inter-core communication by bounding the amount of intercore interference in a partitioned multi-core system. A system-level perspective of how to realize such CCM along with the implementation details is provided. We compare our proposed CCM with Contention-based Communication (CBC) model where no private banking is enforced for any core. For evaluation, we consider San Diego vision benchmark suite (SD-VBS). The results of the evaluation show that the CCM offers 56 percent improvement in worst case execution time (WCET) when compared with CBC.

I. 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 [14], [8], [13], [9], [5] and so far has been acknowledged in industry by Federal Aviation Authority (FAA) [1].

Researchers in [9] 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 CAST32A position paper [1].

The work in [4] 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 [4] as CBC in this work. Our proposed CCM model is for intended for hard-real time applications. Whereas, the CBC based approach is in [4] is designed for mixed criticality applications.

In this work, we follow the partitioning approach described in [9] 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. We acknowledge that Linux is not real-time. 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 [4]. 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 LLC is not available.

The main contributions of this paper include the following:

- A novel CCM that bounds the amount of interference on the DRAM banks when implementing shared communication in a strictly partitioned SCE frame work is proposed.
- Implementation details of the communication library (CommLib) and communication task (CT) using linux and POSIX based APIs in proposed CCM are provided.
- An experimental evaluation of the proposed CCM is provided and is compared with CBC.

The rest of this paper is organized as follows. Section II introduces the related work and background. Section III intro-

978-1-7281-6949-1/20/$31.00 ©2020 IEEE

duces the system model and assumptions. Section IV provides details on the implementation of the proposed CommLib and CT. Section V presents the experimental results of the CCM and the CBC approach on P4080 platform from NXP [2]. Finally, Section VI concludes our work.

II. Background and Related Work

In this section we describe the necessary background and related work that we used as the basis of partitioning shared resources in a multi-core environment.

Multi-core systems have shared resources such as LLC, bus bandwidth and DRAM banks. These shared resources can be partitioned among the cores to avoid conflicts. In literature, techniques such as [8], [12], [11], [3] have been proposed to partition the shared last level cache (LLC), MemGuard [14] to divide memory-controller bandwidth and PALLOC [13] to partition the DRAM into multiple banks.

MemGuard is a memory bandwidth reservation mechanism that is implemented at the Operating System (OS) layer. The main purpose of this mechanism is to distribute (evenly or unevenly depending upon application requirements) the bandwidth available from the memory controller among all the cores. It works periodically, and for each interval e.g., 1ms, a fixed memory budget (Q_p) is assigned to each core. During each period, the hardware performance counter (PMC) on each core measures the amount of memory requests or memory transactionsthat are generated by the core. In the rest of the paper we use the terms memory transactions or memory requests interchangeably. The PMC are 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 descheduling all the tasks. At the beginning of the new period, a new budget assignment takes place, and the previously descheduled 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. In order 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 [13].

III. System Model And Assumptions

A. 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.

B. Proposed Model

Using the hardware assumptions described in Section III-A, 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 Figure 1.

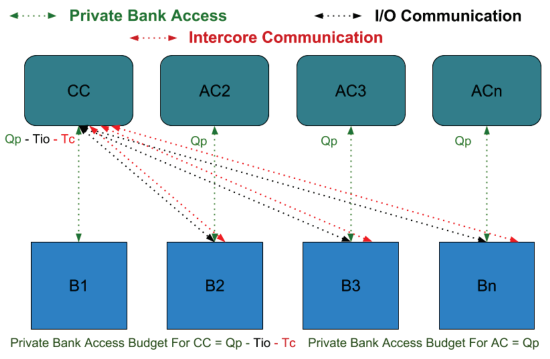

Fig. 1. Block Diagram

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 the system parameters and their values used for evaluations in Sec. V is provided in Table I. 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 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 [10]. In IMA standard, the zero partition (I/O partition) is used to handle all the I/O and inter-partition message exchanges. Existing work [6] 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 Figure 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 Figure 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.

C. 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 [14] and is found to be 1.2GB/s. If we split the bandwidth equally among the cores then each of the core will get 153MB/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 153MB/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.

TABLE I
SYSTEM PARAMETERS

System Parameters	Symbol	Value
Number of Cores	n	8
Number of ACs	$n-1$	7
Memory Regulation Period	P	1 ms

D. Application Task Model

We consider a partitioned and fixed priority scheduling policy, where each core has a set Γ of N periodic application tasks, $\{\tau_1, ..., \tau_N\}$, each with different priority whereby τ_1 has the highest priority and τ_N has the lowest priority.

The deadline of a task is equal to its period. Each AC deploys multiple application tasks (ATs). All the ATs belonging to a dedicated AC only access the private DRAM bank assigned to them. However, this private bank can be accessed by the CT running on the CC.

IV. 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 Figure 2(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 the Figure 2(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 receiving (RX) tasks on a particular destination AC would write to same TX buffer. For instance, Figure 2 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 Figure 2(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

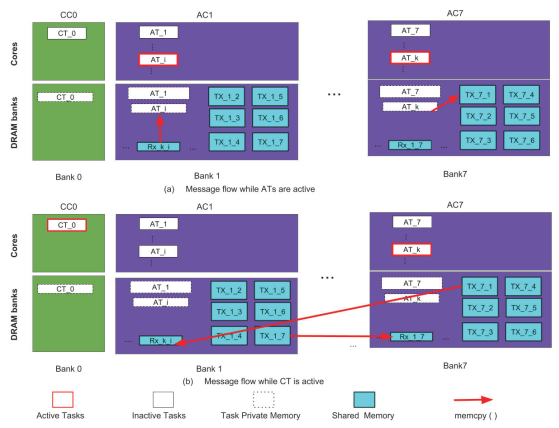

Fig. 2. Message Flow Diagram

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 Figure 2. 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 Sec. III, 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 circular buffers, as the **Message Schedule Queue** and the **Outgoing Message Queue** shown in Figure 3. 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 Figure 3 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 Figure 3.

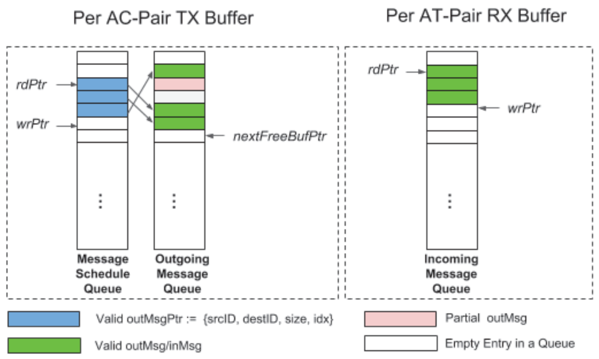

Fig. 3. Per AC-Pair TX Buffer and Per AT-Pair RX Buffer

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 Figure 3. 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.

Algorithm 1: Pseudo Code For send API

```
1  send(txTaskID, rxTaskID, txData, size)
2      txBufferPtr : = findtxBuffer(txTaskID,rxTaskID) ;
3      if txBufferPtr.full() then
4          return -1 ;
5      lock(txBufferPtr);
6      temp : = txBufferPtr.nextFreeBufPtr ;
7      Increment txBufferPtr.nextFreeBufPtr;
8      unlock(txBufferPtr);
9      memcpy (temp, txData, size);
10     lock(txBufferPtr);
11     txBufferPtr.wrPtr.idx := temp ;
12     Increment txBufferPtr.wrPtr ;
13     unlock(txBufferPtr);
14     return size;
```

Algorithm 2: Pseudo Code For receive API

```
1  receive(txTaskID, rxTaskID, rxData, size)
2      rxBufferPtr : =findrxBuffer(txTaskID,rxTaskID) ;
3      if rxBufferPtr.full() then
4          return -1; // No new data
5      memcpy(rxData, rxBufferPtr.rdPtr, rxBufferPtr.size);
6      Increment rxBufferPtr.rdPtr ;
7      size := rxBufferPtr.size;
8      return size;
```

V. Evaluation

We start the evaluation by describing the experimental setup and the benchmarks that we have used for evaluation. Next, we evaluate the communication bandwidth of the implemented CT based on our platform. Finally, we compare and show the benefit of CCM over the CBC approach for the considered benchmarks SD-VBS.

A. 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 128KB Level 2 backside cache. A 2MB of shared Level 3 cache is also present.

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 $sched\_setaffinity()$. For the proposed CCM, PALLOC [13] is enabled and configured such that all the ACs can only access a single dedicated DRAM bank; whereas, the CT on CC can access all the DRAM banks. We use MemGuard [15] with even budget assignment of 2520 memory transactions and a memory regulation period of 1ms.

For the proposed CCM, we consider a system with a single CC and 7 ACs. The parameters used to compute WCETare listed in Table I. 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 deployed on the CC 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. The I/O budget T_{io} is zero.

B. Throughput of the CT

In order to measure the throughput of the CT task running on CC, we use the system parameters described in section V-A. 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. These overhead include bookkeeping tasks of the linux OS. In our setup, we reconfigure the linux to send all I/O device drivers interrupts to the CC. We then run the measure the average overhead of these bookkeeping tasks when nothing is running in the system. It should be noted that for different OSes these overheads might be different. Here, we just demonstrate how one can remove such overheads from the total assigned budget, if they exist. Table II summarizes the distribution of Q_p on the CC. The results indicate that the OS related overhead for bookkeeping tasks on our linux configured P4080 platform during one memory regulation period on average was 604 memory transactions. This indicates that in our implementation the maximum value of T_c available to CT is $Q_p - 604 = 1916$. In order to evenly distribute the remaining budget 1916 among all the 84 AC pairs, we find the closest number that is multiple of 84 and assign that as the total budget $T_c = 1848$ for communication. 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 reveals that 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.

C. CCM and CBC

In this section, we compare the WCET of the CCM with the CBC. For each of these scenarios, the task on core under analysis runs one of the SD-VBS benchmark over 10,000 times.

TABLE II
Budget Distribution on CC

Total Budget Assignment (Q_p)	2520
Average OS Overhead	604
Communication Budget (T_c)	1848
Metadata Overhead (Percentage)	13.6

WCET among 10,000 runs while the other cores run memory intensive program is considered.

In CBC no private banking is enforced and there is no CC. Therefore, if an AC wants to communicate with another AC. It will read the data from its private bank and will directly write it to the bank of other AC. Since all the ACs can directly write to the bank of any other AC. The worse case for CBC occurs when all the ACs are sending data to DRAM bank of one AC. In order to have a fair comparison with CCM, we use six stress cores and one core under analysis. All seven ACs are accessing the same DRAM bank. All the cores are assigned a Memguard budget of Q_p = 2520.

As described earlier, in CCM the communication between the ACs is accomplished using the CC. The worse case in CCM happens when core under analysis runs the benchmark from its private bank with six other ACs running the memory intensive benchmark to stress their private bank and the CC moves data between all pairs of the ACs. In this experiment all the cores are assigned a budget of Q_p = 2520. Some of the assigned Q_p budget on CC is used for bookkeeping. We experimentally found it be 604. As described earlier that inorder to evenly distribute the communication budget equally among all 84 pairs we pick a total budget value of 1848 (a multiple of 84)for highest priority CT. The remaining budget of 68 was used by CC to stress its private bank.

In Figure 4 we provide the WCET comparison of various SD-VBS benchmarks using CBC and CCM approach described above. The results of experiments show that for most SD-VBS benchmarks the CCM on average provides 56% smaller WCET when compared to CBC. The *localization* benchmark in addition shows a further reduction of 65%.

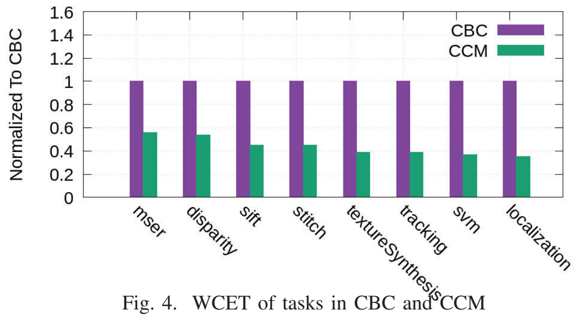

Fig. 4. WCET of tasks in CBC and CCM

VI. Conclusion

We propose an inter-core communication framework for strictly partition multi-core platforms. 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. 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.

Acknowledgment

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1646383, NSF CNS 18-15891 and Navy N00014-17-1-2783. M. Caccamo was also 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.

References

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

[2] Nxp semiconductors. https://www.nxp.com/. Accessed: 2019-10-16.

[3] Muhammad Ali Awan, Konstantinos Bletsas, Pedro F Souto, Benny Akesson, and Eduardo Tovar. Mixed-criticality scheduling with dynamic memory bandwidth regulation. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 111–117. IEEE, 2018.

[4] M. Chisholm, N. Kim, B. Ward, N. Otterness, J. Anderson, and F.D. 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)*, December 2016.

[5] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H., and D. M. Johnson. Rtos support for multicore mixed-criticality systems. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 197–208, April 2012.

[6] Jung-Eun Kim, Man-Ki Yoon, Richard Bradford, and Lui 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*, pages 321–331. IEEE, 2014.

[7] N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, J. H. Anderson, and 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)*, pages 1–12, April 2016.

[8] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 45–54. IEEE, 2013.

[9] Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russel Kegley, Dennis Perlman, Greg Arundale, et al. Single core equivalent virtual machines for hard real—time computing on multicore processors. Technical report, 2014.

[10] Lui Sha, Ragunathan Rajkumar, and Shirish S Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, 1994.

[11] Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 300–303. IEEE, 2008.

[12] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 157–167. IEEE, 2013.

[13] H. Yun, R. Mancuso, Z.P. Wu, and 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*, pages 155–166. IEEE, 2014.

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

[15] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308. IEEE, 2012.