Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC

Conference Paper · July 2018

CITATIONS
4

READS
119

6 authors, including:

Ayoosh Bansal
University of Illinois, Urbana-Champaign
4 PUBLICATIONS **5** CITATIONS
SEE PROFILE

Giovani Gracioli
Federal University of Santa Catarina
30 PUBLICATIONS **157** CITATIONS
SEE PROFILE

Renato Mancuso
Boston University
41 PUBLICATIONS **425** CITATIONS
SEE PROFILE

Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC

Ayoosh Bansal*, Rohan Tabish*,
Giovani Gracioli‡, Renato Mancuso†, Rodolfo Pellizzoni‡, Marco Caccamo*
*University of Illinois at Urbana-Champaign, USA, {*ayooshb2, rtabish, mcaccamo*}*@illinois.edu*
†Boston University, USA, *rmancuso@bu.edu*
‡University of Waterloo, Canada, {*g2gracio, rpellizz*}*@uwaterloo.ca*

***Abstract*—This paper presents the evaluation of the memory subsystem of the Xilinx Ultrascale+ MPSoC. The characteristics of various memories in the system are evaluated using carefully instrumented micro-benchmarks. The impact of micro-architectural features like caches, prefetchers and cache-coherency are measured and discussed. The impact of multi-core contention on shared memory resources is evaluated. Finally, proposals are made for the design of mixed-criticality real-time applications on this platform.**

I. Introduction

The design of efficient computing platforms is essential to achieve real-time guarantees at low power consumption and cost in current and future real-time applications, from complex cyber-physical systems to mobile systems, and to ensure high-performance with acceptable quality of service (QoS) [1]. For instance, in autonomous vehicles, tasks such as steering control, fuel injection, and brake handling, are critical and have hard real-time requirements. Multimedia infotainment systems, however, demand high-performance and tolerate large variations in QoS (*i.e.,* best-effort requirements). Finally, vision-based driver assistance and navigation have become too complex to fit within the traditional development cycle of critical embedded systems, yet they cannot be handled as best-effort software components. Such tasks demand high-processing power and predictability at the same time [1].

Multi-Processor System-on-a-Chip (MPSoC) architectures provide an ideal trade-off between performance and cost to meet such requirements found in complex cyber-physical systems. The considered family of MPSoC architectures is composed of several heterogeneous processing elements with specific functionalities: general-purpose multi-core processors, DSPs, specialized processing cores, GPUs, and FPGA. They also feature a rich memory hierarchy, comprised of scratch-pads, DRAMs, Block RAM, and multiple levels of cache. A similarly rich I/O subsystem, with a number of interfaces, embedded devices, Direct-Memory Access (DMA) engines, shared buses and interconnects completes the picture.

It follows that on the one hand the considered MPSoC platforms provide a vast number of configuration options. On the other hand, however, they also make it difficult to design basic software components (real-time operating system – RTOS and hypervisor), and to understand all the sources of unpredictability. The most relevant sources of unpredictability in MPSoCs are:

- **Shared Memory Hierarchy:** several latency hiding mechanisms, including caches, buffers, scratchpads, and FIFOS are placed among the main memory, processors, and I/O devices. Such mechanisms enable latency and bandwidth demands to coexist in a hierarchy at the price of poor predictability [1]. Techniques such as private memory and cache coherency increase performance, but suffer from limitations in scalability, energy efficiency, and timing [1]. Thus, such techniques become the primary sources of unpredictability in modern MPSoCs [1, 2]. DRAM itself improves the average case performance by using row open arbitration policies or bank level interleaving but these in turn introduce further unpredictability.
- **Shared I/O Subsystem:** latency hiding mechanisms are also used in I/O subsystems. I/O subsystems deliver lower throughput compared to those designed to feed data-hungry CPUs [1]. Many systems are designed assuming that just a few I/O devices will be active at any given time, which is often a wrong assumption for large MPSoCs [1]. Then, delays and deadline misses can occur due to the contention in the I/O subsystem and the increased variation in the response time [1].
- **Shared Buses:** Multi-Processor systems use limited number of shared buses to communicate with the memory subsystems. These buses frequently become a hot spot for contention. The memory bandwidth available to a processor at any instant is affected by activity of other processors. Variable memory latency due to other processors running independent applications can cause any number of deadline violations for a processor. For this problem, various solutions have been proposed [3, 4] and analyzed [5, 6].

In this paper, we provide a benchmark-based analysis of a modern MPSoC considering the main sources of unpredictability and, based on the obtained results, we propose a basic software architecture to improve the predictability of real-time applications running on a MPSoC platform. In summary, the main contributions of this paper are:

- We benchmark the memory types available in a modern

heterogeneous MPSoC platform. We conclude that various memories exhibit varying characteristics and sensitivity to multi-core contention. We use this information to propose an architectural design paradigm in Section V.
- We propose a software/hardware architecture to improve the predictability in the modern MPSoC platforms. Our software architecture relies on a partitioning hypervisor, an RTOS, and several OS-related techniques, such as cache memory partitioning, hardware performance counters, memory bandwidth regulation, and DRAM bank-aware memory allocation.

II. Platform Overview

The selected platform ZCU102 [7] contains a Xilinx Ultrascale+ MPSoC [8]. The main components of this platform are:

1) Application Processing Unit, ARM Cortex A-53 [9]
   - Quad Core ARMv8-A Architecture
   - 32 KB each Private L1 Instruction and Data Cache per core
   - 1 MB Shared L2 cache
2) Real-Time Processing Unit, ARM Cortex-R5
   - Dual-Core ARMv7-R Architecture
   - 32 KB combined Private Instruction and Data Cache per core
   - 128 KB Tightly Coupled Memory (TCM) per core
3) Programmable Logic (PL)
4) Memory
   - *OCM*: 256 KB On-Chip Memory
   - *PS DRAM*: 4 GB DDR4 Kingston KVR21SE15S8/4
   - *PL DRAM*: 512 MB DDR4 Micron MT40A256M16GE-075E connected to Programmable Logic
   - *PL BRAM*: Block RAM in Programmable Logic
5) ARM Mali-400 Based GPU

Figure 1 presents a simplified block diagram of the targeted Ultrascale+ MPSoC. Note that the programmable logic can provide a DRAM controller to access a 512 MB DDR4 memory (here called as PL-DRAM) and a Block RAM (B-RAM). The OCM memory is accessed by the A-53 cores through two buses, and so is the 4 GB DDR4 memory (PS-DRAM). Block RAM (BRAM) [10] are embedded memory elements instantiated in the FPGA which are being used as RAM. We use up to 2 MB of BRAM in the experiments.

The Programmable Logic (PL) communicates with the A-53/R-5 cores and DRAM in the Processing System (PS) via AXI-4 [11] buses. The PS side interface contains 3 AXI Masters and 3 AXI Slaves which can be individually enabled and configured. In our experiments we use 2 AXI Masters on the PS side which connect to AXI Interconnects on the PL which provide the corresponding AXI Slave ports. AXI Master ports on these interconnects are connected to AXI Slave ports on PL DRAM and PL BRAM controllers respectively.

III. Benchmarks

Platform evaluation is performed using user space benchmarks available here [12]. The benchmarks create carefully controlled memory traffic and use timing information for those accesses to deduce platform characteristics.

A. Memory Mapping

Various memories are available on this platform as described in Section II. To benchmark specific memory from linux user space the benchmarks use the `/dev/mem` [13] interface which exposes the physical memory as a file. The `mmap` system call [14] is used to map the physical address space from `/dev/mem` to the virtual address space of the benchmark. The `mmap` system call in the kernel was modified to explicitly control the cacheability of the mapped memory. The mapped memory could be made cacheable or non-cacheable as desired. Due to the small size in the same order of magnitude as L2 Cache, PL Block RAM is always mapped as non-cacheable in all experiments.

B. Memory Latency

Memory Latency is defined here as the time difference between the processor issuing a read request and receiving the data. A strict data dependence is created between each load used to evaluate the latency. This effectively eliminates any parallelization that could be introduced by the compiler or processor and skew this metric. The average latency is calculated over a large number of such loads.

The behavior of this benchmark was verified by inspecting the assembly code generated by the compiler and using perf utility [15] at runtime. The benchmark was compiled with gcc -O2 optimizations.

C. Bandwidth

This benchmark evaluates the read or write bandwidth available to the processor for specific physical memory address ranges. The benchmark accesses the memory range under evaluation in a sequential manner with the corresponding type of access (read or write). This is done for *5 seconds* and the average bandwidth is calculated. The benchmark evaluates the processors' ability to read or write sequential address ranges. Every access made to the memory is 64 bits wide. The benchmark was compiled with gcc -O2 optimizations.

D. Cache Coherence

The effect of cache coherence on memory access time is also evaluated. The benchmark considered is similar to the one used in [16]. The benchmark in [16] uses two tasks. Each task accesses a fixed memory range with reads and writes in a sequential manner. The two tasks can be arranged with respect to each other in the following arrangements:

- *Sequential*: The two tasks are run one after the other on the same processor;
- *Parallel*: The two tasks run on two different processors but access private data only. There is no coherence dependence between the tasks;

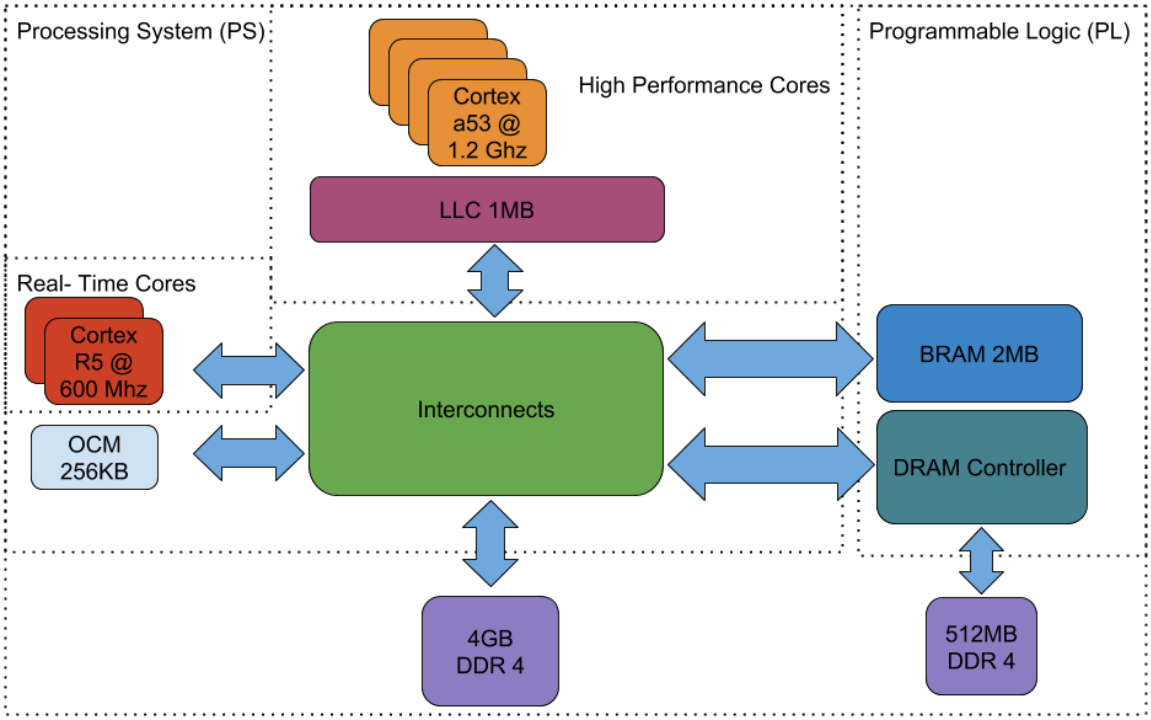

Fig. 1. Simplified Block Diagram of the UltraScale+ MPSoC

- *Concurrent*: The two tasks run on two different processors and access shared data leading to overheads due to coherence traffic.

IV. Platform Evaluation

This section provides a summary of the evaluation results.

A. Measuring Latency and Bandwidth

Using the latency benchmark as described earlier in Section III-B, we measured latency of different memory subsystems. The results of the experiments for serialized versus random access pattern to measure latency for PS DRAM, PL DRAM and PL Block RAM are shown in Figure 2. Memory accesses in this experiment bypass caches as described in Section III-A. The experimental results reveal that both PS-DRAM and PL-DRAM show less latency in serialized access compared to random access. The PL-BRAM does not exhibit any latency difference between serialized versus random access. BRAM accesses latency is independent of access pattern as it lacks constructs like banks and row buffers that are common in DRAMs.

We also ran the latency benchmark with caching enabled for varying working set sizes. Figure 3 shows the results. At the lowest working set size of 16 KB, all accesses hit in private L1 d-Cache of the processor. The access latency for the L1 d-Cache is hence around **3ns**. The shared L2 Cache has a capacity of 1 MB. Until the working set is increased beyond the 1 MB mark, the majority of memory accesses hit in L1 or L2 cache. The sharp latency increase for working sets larger than 1 MB are due to actual DRAM accesses. L2 cache latency

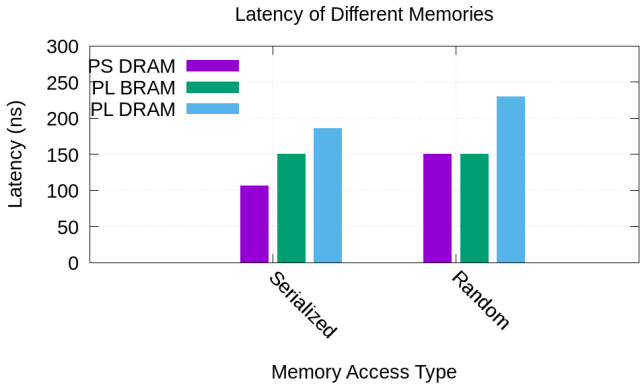

Fig. 2. Stress Results of PL Versus PS DRAM

is hence around **20ns**. Serialized read latency is substantially lower than random read latency. Additionally, note that the read latency for serialized memory accesses, even for large working sets, is comparable to L2 cache latency. This is the impact of speculative prefetching. Recall that the results in Figure 2 were obtained by defining non-cacheable buffers. At large working set sizes the latency for randomized accesses to cacheable memory (see Figure 3) converges to the latency observed for non-cached memory (Figure 2), as all accesses miss in L1/L2 cache.

Similar to the latency benchmark, we run the bandwidth benchmark described in Section III-C on A-53 core to measure

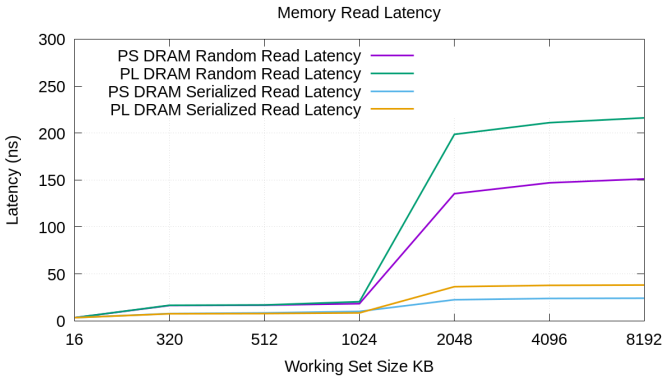

Fig. 3. Random and Serial Read Latency With Different Working Set Size

the bandwidth of different memory sub-systems as reported in Table I. From these results we draw several conclusions. In general PS DRAM is better then both PL DRAM and PL Block RAM. This is due to shorter line distance to the PS DRAM and higher clock rates in the PS subsystem. Reads with caching enabled are boosted by speculative prefetching as the accesses are strictly serial. Multiple loads are issued per cache line leading to further boost in Read bandwidth with caches as compared to without caches. Reads without caches fetch data from underlying memory on every access and hence suffer a low bandwidth. Writes without caching return asynchronously i.e. the store instruction returns without waiting for the data to be committed to the underlying memory. Without caching there is not a requirement to allocate a cache line to complete a store. In case of writes with caches enabled, stores frequently lead to dirty cache line evictions and cache line allocate for the first write to a cache line (write-allocate policy). Hence we see the large write bandwidth when caches are disabled but a low write bandwidth with caching enabled. PL Block RAM is only accessed with caches disabled. Read bandwidth from PL Block Ram is greater than PL DRAM as the logic to reach Block RAM in Programmable Logic is smaller than that to reach PL DRAM. Block RAM is also inherently faster than DRAM for single access latency which is a good approximation for the traffic pattern of the read bandwidth benchmark. On the other hand, write bandwith benchmark without caches bombards the underlying memory with write requests. In this case PL Block RAM provides a lower throughput than the PL DRAM. This is due to lack of parallelization of memory accesses and limited buffering in the access path to PL Block RAM, as compared to PL DRAM.

TABLE I
BANDWIDTH MEASUREMENTS FOR DIFFERENT MEMORIES

Access Type	PS DRAM (MB/s)	PL DRAM (MB/s)	PL BRAM (MB/s)
Write With Cache	1881	880	xx
Read With Cache	2493	1414	xx
Write W/O Cache	12000	5440	4568
Read W/O Cache	556	320	406

B. Measuring Latency Under Stress

In this section we report the memory latency seen by a core under analysis running the latency benchmark when other cores are stressing the same memory as core under analysis using the bandwidth benchmark. The bandwidth benchmark on other cores is configured to stress with write, whereas the latency is configured to perform read. In Figure 4, we show the amount of read latency seen by the core under analysis on PS DRAM and PL DRAM as we increase the stressing cores from one to up to three. Compared to solo case, the stress case of three cores shows a slow down of 1.85 times for the PS DRAM and a slow down of 5.37x for the PL DRAM. This slow-down can be explained by the DRAM specs of the PL and PS DRAM and the interconnect between the two. We also note that BRAM access latency is largely unaffected by the increasing contending traffic.

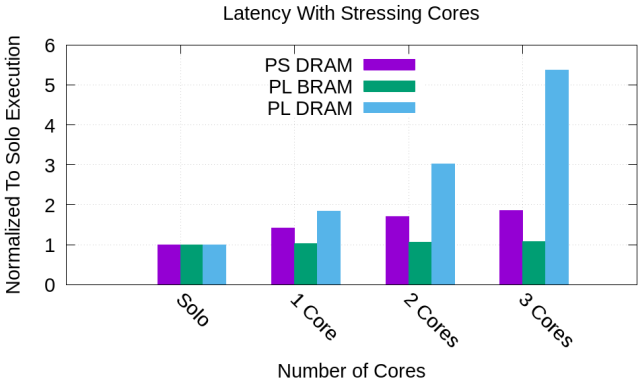

Fig. 4. Stress Results of PL BRAM/DRAM Versus PS DRAM

C. Evaluating Cache Coherence

Figure 5 shows the results of cache contention experiments as described in section III-D. We can clearly note the effects of the cache coherence protocol on the performance. The concurrent benchmark version, which runs two threads in different cores at the same time accessing the same data array, is about **3.6** times slower than the parallel version. When the second thread accesses the shared data, it gets an invalid access and must ask (snoop request) for the most recent copy of the data or recover it from a higher memory level [16]. Whenever a snoop request must be completed, it takes time, which may lead to unexpected increase of the task's execution time and deadline misses [2, 16]. According to [17], the time to complete a snoop request is considerably slow (comparable to access the off-chip RAM).

ARM Cortex-A53 processor uses the MOESI protocol to maintain data coherency between multiple cores [9]. Coherency is maintained between the cores, cache, I/O master, PL, and DRAM using the cache coherence interconnect (CCI).

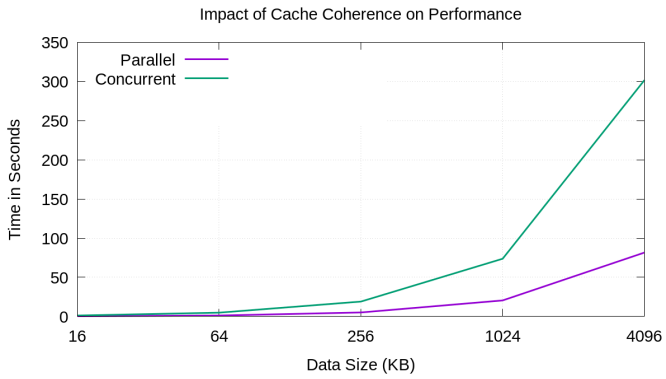

Fig. 5. Cache Coherence Results

V. Proposed Software/Hardware Predictable Architecture

In order to provide strong temporal isolation among high performance cores on the considered heterogeneous MPSoC, we propose the software architecture as shown in Figure 6.

Our proposed architecture uses Jailhouse hypervisor [18] that provides physical isolation of hardware devices including processors, among the different OSes. We propose to use a general-purpose OS such as Linux for non-critical tasks on one of the high performance core and a Real-Time operating system (RTOS) such as Erika [19] for safety-critical tasks. Like any hypervisor, the communication between different OSes running on different cores is achieved using Jailhouse. We propose prohibiting direct access to the shared resources from different cores. This eliminates unbounded contention which could make the system unpredictable.

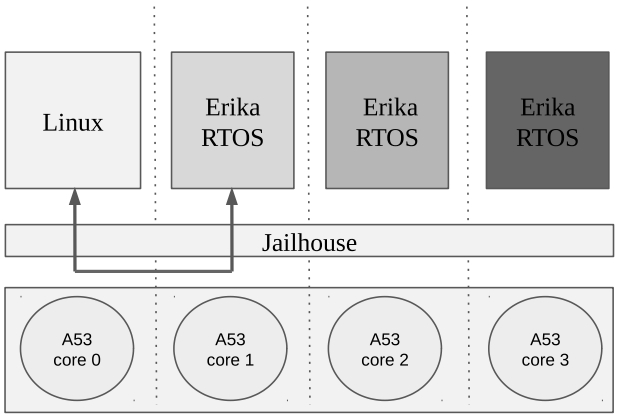

Fig. 6. Overview of the envisioned software architecture.

From our experimental results, it is clear that the main sources of contention in our system are shared memory resources such as LLC and the main memory such as PS DRAM. We propose to partition the LLC using page coloring with the help of Jailhouse. This removes the contention that can be introduced by the LLC. In order to avoid the contention at the DRAM, we propose the use of DRAM bank-aware memory allocator (PALLOC) [20]. Using cache partitioning and PALLOC we can assign a specific amount of cache and dedicate DRAM banks to a specific core and enforce strong isolation between the OSes running on different cores. For shared memory, we propose to use PL block RAM (BRAM). This is because, as shown by the experimental results in Figure 4, the BRAM does not suffer any contention when accessed using different cores.

VI. Related Work

Our proposed software architecture is similar to one proposed in [21]. However, in our proposal, Jailhouse would be responsible for providing cache partitioning (possibly through page coloring) and also DRAM bank-aware memory allocator (through PALLOC [20]). Modica et al. also proposed a similar hypervisor-based architecture targeting critical systems [22]. Cache partitioning is used to provide spatial isolation, while a DRAM bandwidth reservation mechanism provides temporal isolation. Both cache partitioning and memory reservation mechanisms were implemented in the XVISOR open-source hypervisor [23] and tested in a quad-core ARM A7 processor. Our proposed hypervisor-based approach, instead, uses a MPSoC platform, which gives the possibility to explore other features, such as specific FPGA DMA blocks (to handle data transfer between PS and PL sides for instance) and data prefetching. Another difference is that our approach will also use DRAM bank-aware memory allocator, which can provide better predictability in terms of main memory accesses.

MARACAS addresses shared cache and memory bus contention through multicore scheduling and load-balancing on top of the Quest OS [24]. MARACAS uses hardware performance counters information to throttle the execution of threads when memory contention exceeds a certain threshold. The counters are also used to derive an average memory request latency to reduce bus contention. vCAT uses the Intel's Cache Allocation Technology (CAT) to achieve core-level cache partitioning for the hypervisor and virtual machines running on top of it [25]. vCAT was implemented in Xen and $LITMUS^{RT}$. Although interesting, this approach is architecture dependent and uses non real-time basic software support (Linux and Xen).

Kim and Rajkumar proposed a predictable shared cache framework for multicore real-time virtualization systems [26]. The proposed framework introduces two hypervisor techniques (vLLC and vColoring) that enables cache-aware memory allocation for individual tasks running running in a virtual machine. CHIPS-AHOy is a predictable holistic hypervisor [1]. It integrates shared hardware isolation mechanism, such as memory partitioning, with an observe-decide-adapt loop to achieve predictability and energy, thermal, and wearout management.

VII. Conclusions

In this paper we have evaluated the different memory subsystems of the Xilinx Ultrascale+ platform. The results of the experiments show that the platform has significant

contention at LLC, PS DRAM and PL DRAM. Therefore, it cannot be used as is for multi-core applications requiring hard-real time guarantees. To provide strong isolation among the cores, we propose the use of cache coloring using JailHouse (a hypervisor) and DRAM bank partitioning using PALLOC. With strict partitioning of shared resources we can run Real Time OS on any core unaffected by application running on other cores.

REFERENCES

[1] T. Mück, A. A. Fröhlich, G. Gracioli, A. Rahmani, and N. Dutt. Chips-ahoy: A predictable holistic cyber-physical hypervisor for mpsocs. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 1–8, Samos Island, Greece, 2018.

[2] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2), 2015.

[3] 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 *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2013.

[4] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 299–308, July 2012.

[5] R. Mancuso, R. Pellizzoni, N. Tokcan, and M. Caccamo. WCET Derivation under Single Core Equivalence with Explicit Memory Budget Assignment. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:23, Dagstuhl, Germany, 2017.

[6] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *Proceedings of the 2015 27th Euromicro Conference on Real-Time Systems*, ECRTS '15, pages 174–183, Washington, DC, USA, 2015. IEEE Computer Society.

[7] Inc. Xilinx. Ultrascale+ MPSoC ZCU102. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, 2018. [Online; accessed 16-April-2018].

[8] Inc. Xilinx. Ultrascale+ MPSoC Overview. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf, 2018. [Online; accessed 16-April-2018].

[9] Arm Holdings. Cortex A-53. https://developer.arm.com/products/processors/cortex-a/cortex-a53, 2018. [Online; accessed 16-April-2018].

[10] Xilinx. Block RAM. https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/ond1517252572114.html, 2018. [Online; accessed 22-May-2018].

[11] AXI. https://www.arm.com/products/system-ip/amba-specifications, 2018. [Online; accessed 24-April-2018].

[12] Heechul Yun. Benchmarks. https://github.com/heechul/misc, 2018. [Online; accessed 20-April-2018].

[13] mem. http://man7.org/linux/man-pages/man4/mem.4.html, 2018. [Online; accessed 24-April-2018].

[14] mmap. http://man7.org/linux/man-pages/man2/mmap.2.html, 2018. [Online; accessed 24-April-2018].

[15] perf. https://perf.wiki.kernel.org/index.php/Main_Page, 2018. [Online; accessed 24-April-2018].

[16] G. Gracioli and A. A. Fröhlich. On the influence of shared memory contention in real-time multicore applications. In *2014 Brazilian Symposium on Computing Systems Engineering*, pages 25–30, Nov 2014.

[17] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[18] Siemens. Jailhouse. https://github.com/siemens/jailhouse, 2018. [Online; accessed 22-May-2018].

[19] Erika Enterprise. Erika Enterprise RTOS v3. http://www.erika-enterprise.com/, 2018. [Online; accessed 22-May-2018].

[20] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014.

[21] Alessandro Biondi, Mauro Marinoni, Giorgio Buttazzo, Claudio Scordino, and Paolo Gai. Challenges in virtualizing safety-critical cyber-physical systems. In *Proceedings of Embedded World Conference 2018*, pages 1–5, Feb 2018.

[22] Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT 2018)*, pages 1–7, Feb 2018.

[23] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691, March 2015.

[24] Y. Ye, R. West, J. Zhang, and Z. Cheng. Maracas: A real-time multicore vcpu scheduling framework. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 179–190, Nov 2016.

[25] M. Xu, L. Thi, X. Phan, H. Y. Choi, and I. Lee. vcat: Dynamic cache management using cat virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222, April 2017.

[26] Hyoseung Kim and Ragunathan (Raj) Rajkumar. Predictable shared cache management for multi-core real-time virtualization. *ACM Trans. Embed. Comput. Syst.*, 17(1):22:1–22:27, December 2017.