A Lightweight Spatio-temporally Partitioned Multicore Architecture for Concurrent Execution of Safety Critical Workloads

***Abstract*—The computing hardware layer in modern industrial systems (such as avionics and automobiles) must be able to execute many concurrent workloads under tight deterministic execution guarantees to meet the safety standards. Single-chip multicores are attractive for safety-critical embedded systems due to their lightweight form factor. However, multicores aggressively share hardware resources, leading to interference that in turn creates non-deterministic execution for multiple concurrent workloads. We propose an approach to *remove on-chip interference* via a set of methods to spatio-temporally partition shared multicore resources. Our proposed partitioning scheme is bounded within the worst case execution, and ensures efficient performance and deterministic execution.**

I. Introduction

The distributed and embedded software in modern industrial systems (such as avionics and automobiles) is increasingly becoming a first order design constraint. For example, modern aircrafts deploy complex embedded software and hardware to manage various operational and management layers [1]. The complexity of the avionics software is exponentially increasing because of the addition of cyber layer to its system design, making a paradigm shift from a distributed on-board system architecture to an integrated modular avionics (IMA) architecture. IMA integrates several unrelated or loosely related software applications formerly distributed across multiple avionic systems, and lessen the system weight due to fewer avionics, wiring and energy consumption [4].

At the hardware layer, multicore technology has emerged as an ubiquitous computational primitive. These multicores integrate many discrete hardware components, such as compute processors, cache memories, interconnection networks, and memory controllers on a single chip. Through integration multicores deliver small form factor, and reduce the embedded hardware's area, power and energy footprint. The key question we seek to answer in this paper is whether multicore technology can be deployed in safety-critical systems, such as the IMA architecture for avionics.

Traditionally, safety-critical applications are executed on discrete processors. In case of a multicore processor, this implies the application may not fully utilize the hardware capabilities, such as the additional processing cores for computations, or the available memory bandwidth to access data. Therefore, it is beneficial to concurrently execute multiple applications and fully utilize the available hardware capacity. The challenge is that the aggressive integration leads to space-time sharing of hardware resources. This sharing exists even though the concurrently executed applications do not explicitly communicate across their control and dataflow. Sharing causes interference among the hardware resources, such as the on-chip Last Level Cache (LLC), memory controller, and the network-on-chip. These *interference channels* lead to non-deterministic timing and power behaviors across applications, leading to loss of performance guarantees. For example, the multicore LLC holds the working sets of concurrent applications. Improper allocation of the cache resources to any particular application can lead to unpredictable cache behavior that can be detrimental for performance. In the worst case an application can get locked out from accessing data that can lead to starvation like conditions for the embedded system. This is unacceptable for safety-critical applications, such as the ones executing under the IMA architecture for avionics. On the other hand, multicore's efficient form factor offers desirable capabilities for such embedded systems. Therefore, we propose a holistic approach to remove all on-chip interference channels via a set of lightweight methods to spatio-temporally partition shared multicore resources.

The key contributions of this paper are:

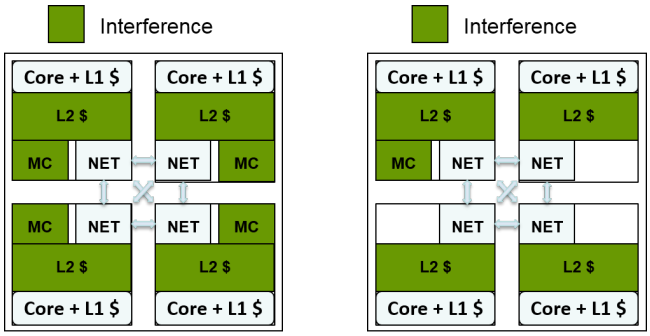
- Propose a holistic partitioning scheme for the LLC and memory controller for multicore system.
- Present an interference analysis, and justify that the proposed scheme fulfills the requirements of FAA Certification (CAST-32) [5] for on-chip interference channels.
- Improve the performance for deterministic execution of concurrent applications using the proposed partitioning scheme, which tightens the bound for worst case execution time (WCET). We show that the deterministic performance under the proposed scheme is competitive with respect to a non-deterministic scheme that exploits fine grain sharing of hardware resources.

Our approach is a step towards satisfying deterministic performance guarantees for a system using multicore processors (MCP). Certifications, such as Federal Aviation Administration (FAA) Certification Authorities Software (CAST-32) [5] are essential for deployment of embedded multicore hardware for safety-critical applications.

II. Partitioning of Shared Resources

A. Overview

Figures 1a and 1b show two baseline multicore systems that we consider in this paper. Each system has four cores, where the compute pipeline and the level-1 (L1) caches are private for each core. So, the application has a private access to these resources without any interference. The Network-on-Chip (NoC) is a point-to-point network where the data packets sent by each core have a unique non-blocking path between the source and destination. This is a reasonable assumption since state-of-the-art multicore processors already implement crossbar interconnection networks [15]. Thus, the



(a) Baseline system 1: one memory controller for each core

(b) Baseline system 2: Single Memory Controller for all the cores

Fig. 1: Baseline systems with different number of memory controllers

point to point network avoids interference in NoC. The shared resources (highlighted in Figure 1) are the Last Level Cache and Memory Controller. These interference channels are accessible by each core in the system and traditionally fine-grain hardware schemes, such as cache insertion/replacement policies, dictate how they are accessed. Although such schemes work well in practice, they are highly unpredictable and do not guarantee bounded performance for safety-critical systems.

As a primitive approach to eliminate the effect of interference, a Worst Case Execution Time (WCET) Analysis is done such that determinism is ensured by not violating the WCET [8]. For our interference channels, WCET is modeled such that all the misses from L1 cache accesses bypass the LLC and access the shared memory controller. In this regard, the shared LLC access is disabled for allocation. The shared memory controller is temporally partitioned for WCET, similar to Wang et al. [17]. These two mechanisms allow us to consider a WCET implementation for both baseline multicore systems shown in Figure 1.

The key idea we propose is to tighten the WCET bounds to improve the performance of co-located safety-critical applications on a multicore processor. We achieve this through deterministic spatio-temporal partitioning of the LLC and memory controller interference channels. The following subsections explain the proposed partitioning schemes where applications are assigned fixed number of LLC cache slices, and a fixed number of memory controllers and/or memory bandwidth.

B. FAA CAST-32 Requirements and Evaluation

The rationale of the FAA certification is that applications executing on different cores of a multicore processor (MCP) do not typically execute independently from each other because the cores are sharing resources. A coupling exists on the platform level since they are implicitly sharing platform resources and could result in interference between these applications. The concern is that there may be channels through which the MCP cores or the software hosted on those cores could interfere with each other. Without mitigation being incorporated to deal with such interference, non-deterministic behavior of the hosted software may occur. Another concern is that using shared cache can result in WCETs of the software applications hosted on one core of an MCP increase greatly due to repeated cache accesses by the processes hosted on the other core [5]. Thus the following requirements are raised to avoid these situations. In this paper, we illuminate how the proposed scheme can fulfill these requirements for the MCP setup.

1) Conduct a functional interference analysis to identify all the interference channels between the software hosted on the cores of the MCP and has designed, implement and verify a means of mitigation for each of those interference channels [5]: As discussed in Section II-A, the interference channels of proposed system are shared LLC and memory controller(s). Since the proposed scheme spatio-temporally partitions these shared components, each application gets dedicated resources, and the interference is avoided. Evaluation results are shown in Section IV.

2) Describe the strategy for managing and verifying cache usage [5]: The shared LLC is partitioned based on the applications' demands. The applications whose WCET is more sensitive to LLC size get larger partitions, and vice versa. Thus the LLC is partitioned to get the best WCET of applications executing concurrently.

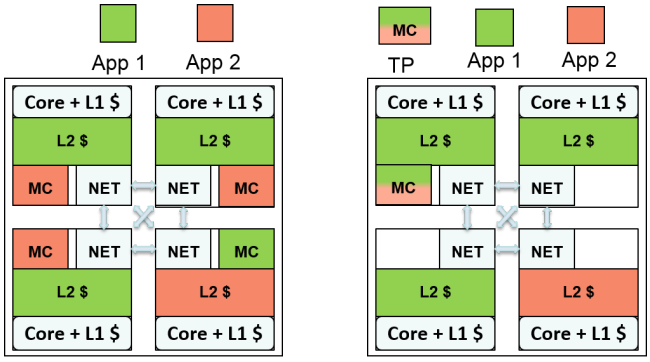
3) Conduct analyses and tests to determine the worst-case effects that the use of shared cache can have on the execution of the specific software applications hosted on the cores of the MCP, describe those effects, implement and verify a means to mitigate the effects of using shared cache [5]: The analyses are shown in Section IV.

Next, we describe our proposed spatio-temporal partitioning mechanisms for the two baseline multicore systems introduced in Figures 1a and 1b.

C. Spatial Partitioning of Last Level Cache

Certain applications are sensitive to LLC space and some are memory bandwidth sensitive. So, in the proposed partitioning mechanism both the LLC and memory controller are envisioned to be spatially partitioned across the competing applications. There is no state of the art partitioning scheme that performs temporal partitioning of the LLC and guarantee zero interference. It is because of the latency overhead in flushing the LLC data to off-chip memory and bringing it back during each partition's execution is prohibitive. It makes spatial partitioning the only viable way to guarantee deterministic resource sharing in LLC, which in turn potentially improves performance. Our baseline multicore has physically distributed and logically shared last level cache (LLC). Each core gets a slice of the LLC [14]. This distributed nature of the LLC places the onus on partitioning and placement of the data in the LLC, also know as the Non-Uniform Cache Access (NUCA) [16]. We exploit this NUCA capability to spatially partition the LLC slices across the concurrent applications. Although further fine-grain partitioning can be done at the cache way and/or set granularities, we leave such exploration for future work.

Each LLC slice consists of multiple physical cache banks. Before the start of execution, the number of LLC slices for allocation to each application are determined based on the expected behavior of that application. For example, a memory bound application is expected to work well with smaller number of LLC slices as compared to a compute bound application. Once the number of LLC slices is determined



(a) Application 1 gets 3 LLC slices and 1 memory controller. Application 2 gets 1 LLC slice and 3 memory controllers

(b) Application 1 and 2 gets 3 and 1 LLC slices accordingly. Memory controller is temporal partitioned.

Fig. 2: Example of partitioning schemes: (a) Spatial partitioning of LLC and memory controllers (b) Spatial partitioning of LLC and temporal partitioning of memory Controller

Architectural Parameter	Value
Number of Cores	4 @ 1 GHz
Compute Pipeline per Core	In-Order, Single-Issue
Physical Address Length	48 bits
Memory Subsystem	
L1-I Cache per core	8 KB, 4-way Assoc., 1 cycle
L1-D Cache per core	8 KB, 4-way Assoc., 1 cycle
L2 Cache per core	32 KB, 8-way Assoc., 8 cycle Inclusive, NUCA
Cache Line Size	64 bytes
Num. of Memory Controllers	4 for baseline-1, 1 for baseline-2
DRAM Bandwidth	0.2 GBps/cntrl. for baseline-1, 0.8 GBps/cntrl. for baseline-2
DRAM Latency	100 ns
point-to-point Network	
Hop Latency	2 cycles (1-router, 1-link)
Flit Width	64 bits
Header (Src, Dest, Addr, MsgType)	1 flit
Cache Line Length	8 flits (512 bits)

TABLE I: Architectural parameters for evaluation.

for an application, its address space is *statically* mapped to those LLC slices (through some architecture register setting). An address interleaving scheme at the cache line granularity is utilized to achieve optimal use of the available LLC capacity allocated for that application [2]. At runtime, when an application accesses a data word from its address space, the static mapping determines the LLC slice for allocating the corresponding cache line from off-chip memory. In this way the LLC interference channel is managed by isolating the application data to be mapped in unique LLC slices.

D. Spatio-Temporal Partitioning of Memory Controller

An on-chip memory controller consists of multiple FIFO queues for read/write requests to the off-chip memory. Each memory controller queue is associated with the available memory controller bandwidth. Memory bandwidth decides the rate at which off-chip data packets are accessed, and is related to the number of pins that connect the multicore processor to the off-chip memory. Before the start of execution, memory controller bandwidth is divided among the concurrent applications. Each core then gets its allocated memory bandwidth, effectively either spatially and/or temporally partitioning the memory controllers among the applications. When the available memory controllers are greater than the number of concurrent applications, the memory controllers are spatially allocated to the applications using the same strategy described in Section II-C for the LLC. When the number of memory controllers is less than the concurrent applications, it is not possible to only rely on spatial partitioning. In this case, the memory controller is temporally partitioned such that each application gets its respected space for a deterministic interval of time [17].

Figure 2a shows an example of spatial partitioning for the baseline system shown in figure 1a. $LLC(3,1)$ implies application 1 is allocated three LLC slices, whereas application 2 is allocated one LLC slice. $MC(1,3)$ implies application 1 is allocated one memory controller (or one-fourth of the memory bandwidth), whereas application 2 is allocated three memory controllers (or three-fourth of the available memory bandwidth). Likewise, there are several possible combinations of LLC partitioning together with memory controller partitioning. Figure 2b shows an example of spatio-temporal partitioning for the baseline system shown in figure 1b. $LLC(3,1)$ shows LLC is spatially partitioned, similar to the previous example. However, since the baseline-2 system has a single memory controller, it is temporally partitioned (MC_TP) for deterministic intervals of time.

E. Selecting the Optimal Partition

The proposed spatio-temporal partitioning scheme is suitable for embedded multicores with low core counts since the number of partitioning combinations are manageable. The optimal partitioning scheme is selected offline through a *roofline model* [9] that classifies the applications either as compute bound or memory bound. The compute bound applications discard single LLC slice option and are allocated either two or three LLC slices. The rationale here is that compute bound applications rely on the LLC capacity to deliver good performance. On the other hand, memory bound applications discard the one fourth memory controller bandwidth and are allocated either two-fourth or three-fourth the memory controller bandwidth. The rationale here is that memory bound applications rely on higher memory bandwidth to deliver good performance.

III. Experimental Methology

A. Performance Modeling

All experiments are performed using our modified version of the Graphite multicore simulator [3]. We modified Graphite to support multiple multi-threaded applications to execute concurrently. The simulator is configured as a four-core shared memory multicore, as shown in Table 1. Each core consists of a compute pipeline, private L1 instruction and data caches, a physically distributed shared LLC with integrated directory, and a network router. A memory controller can also be placed on any of the tiles. Cache coherence is implemented using the directory based MSI protocol. The on-chip NoC is a point-to-point network with a 2-cycle per hop (router + link) delay. We

also account for the appropriate pipeline latencies associated with loading and unloading a packet onto the network.

1) ***Baseline****:* The baseline executes a single two-threaded application on the 4-core system, with accesses to all the shared resources. Thus 2 compute pipelines, 2 L1 instruction and 2 L1 data caches, 4 LLC slices and 4 memory controllers are available to the application. Each multithreaded application is run to completion, and we measure the completion time, i.e., the number of cycles to execute the application. The baseline performance is the maximum completion time of the two applications when executed to two physically separate multicores. All results of concurrent application executions are normalized to this baseline performance.

*2) Uncontrolled Sharing (****U_Sharing****):* The uncontrolled sharing (U_Share) configuration executes two applications concurrently without any partitioning scheme. Both two-threaded applications access the 4 LLC slices and 4 memory controllers without any restrictions.

3) Worst Case Execution Time (WCET): The notion of WCET is that in order to eliminate interference, no shared on-chip channels can be used. Two applications are isolated on-chip by only allowing accesses to the private components. Thus it is configured in a way that all L1 cache misses access the memory controller(s), avoiding allocation in the LLC. The available memory controllers are temporally partitioned among both applications.

4) LLC and Memory Controller Partitioning Schemes: Our proposed spatio-temporal partitioning scheme is implemented for the LLC and memory controller interference channels. For each application combination, there are 3 ways to partition LLC slices (**(LLC(1,3), LLC(2,2), LLC(3,1)**), and 4 ways to partition memory controller(s) (**(MC(1,3), MC(2,2), MC(3,1), MC_TP**). For example, when 4 memory controllers are available, spatial partitioning is used. However, when 1 memory controller is implemented, the only way to partition it is temporal partitioning.

B. Benchmarks

The benchmarks and their input sizes are shown in Table II. We used three SPLASH-2 [23] benchmarks (FFT, RADIX, and CHOLESKY), and a matrix multiplication (MM) benchmark to represent applications for safety-critical systems. The notion is that the inner-product of basic building block in all matrix computations is used widely in the vector based computations, which is an important component of such systems. The matrix multiplication (MM) is loop-tiled, so is optimized for cache accesses. FFT is matrix based, and also important kernel in several signal processing applications. Real-time time applications such as audio/video processing use FFT as the basic transform for frequency domain analysis. CHOLESKY is also a matrix decomposition algorithm. It is used in systems where the model reduction is done during the state-space analysis. Consider a helicopter scenario where it needs to optimize its rotor speed according to the environmental conditions. The environment data needs to be reduced for comprehension and efficient computation. RADIX employs a divide-and-conquer methodology for sorting datasets, such sensor data that needs to be sorted according to the magnitude.

Application	Problem Size
Cloned Latency Critical Benchmarks	
RADIX	1M Integers, radix 1024
FFT	1M points
MM	512×512 matrix, 16×16 blocks
CHOLESKY	512×512 matrix, 16×16 blocks

TABLE II: Problem sizes for the parallel benchmarks.

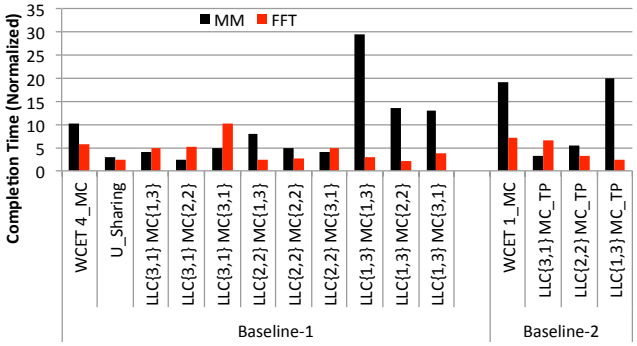


Fig. 3: Partitioning combinations for Matrix Multiplication-FFT. All completion times are normalized to the **Baseline**.

IV. Results and Analysis

Figure 3 shows the normalized completion time of MM-FFT application combination. When 4 memory controller are used (baseline-1), MM can use a spatial partitioning configuration. For example, the **(LLC(1,3) MC(3,1))** in Figure 3 shows the performance of the partitioning combination where MM is mapped to 1 LLC slice and 3 memory controllers, while FFT is mapped to 3 LLC slices and 1 memory controller. When 1 memory controller is used (baseline-2), the only way to partition it is temporal partitioning. For example, **(LLC(1,3) MC_TP)** in Figure 3 shows that MM and FFT are mapped to 1 and 3 LLC slices respectively. However, the memory controller is temporally partitioned among both MM and FFT. All the *twelve* possible spatio-temporal partitioning combinations are shown in Figure 3.

All results are normalized to the completion time of the worst one of the two applications using Baseline (FFT in this case). It is clear that MM is sensitive to the LLC size since all 1 LLC slice allocations for MM result in significant performance

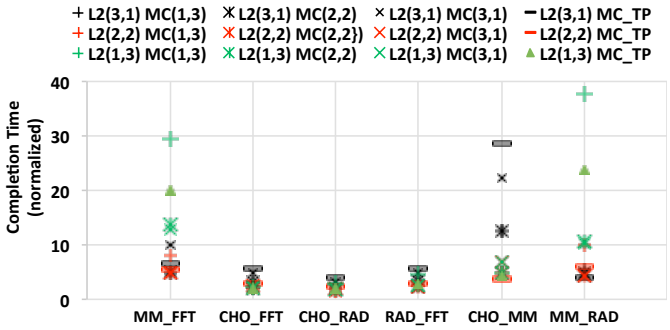


Fig. 4: Cache and memory bandwidth sensitivity of application combinations using different partitioning schemes. The combinations are: Matrix Multiplication-FFT, Cholesky-FFT, Cholesky-Radix, Cholesky-Matrix Multiplication and Matrix Multiplication-Radix accordingly.

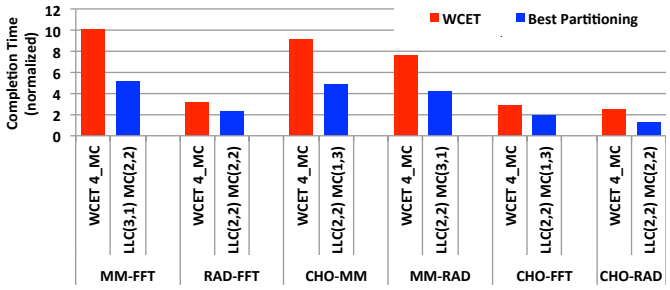
Benchmark Combinations	Eliminated LLC Partitioning	Eliminated Memory Controller Partitioning	Worst Partitioning
MM-FFT	LLC(1,3)	MC(3,1)	LLC(1,3) MC(1,3)
CHOLESKY-FFT	LLC(1,3)	MC(3,1)	LLC(3,1) MC(3,1)
CHOLESKY-RAD	LLC(1,3), LLC(3,1)		LLC(3,1) MC(3,1)
RADIX-FFT	LLC(1,3)	MC(3,1)	LLC(1,3) MC(1,3)
CHOLESKY-MM	LLC(1,3), LLC(3,1)		LLC(3,1) MC(3,1)
MM-RADIX	LLC(1,3), LLC(3,1)		LLC(1,3) MC(1,3)

Fig. 5: Eliminated partitioning schemes using roofline model

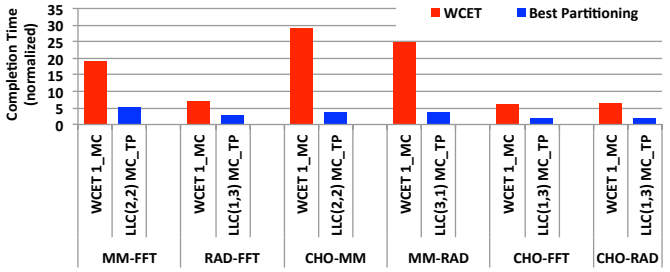
slowdowns. For example, for **LLC(1,3) MC_TP**, MM has more than 20× slowdown, while FFT only has 2× slowdown. This also meets the roofline model [9] whereby MM trends to be compute bound, while FFT is memory bound. The selection of the appropriate partitioning combination is crucial since wrong decision can lead to a slowdown that is worse than the WCET, as observed in the case of **LLC(1,3) MC(1,3)** partitioning combination. The proposed decision making method using the roofline model is effective. Since MM is classified as compute bound, while FFT is memory bound, MM is assigned at least two LLC slices while FFT is assigned at least two memory controllers. Thus the portioning combinations with **LLC(1,3) or MC(3,1)** are eliminated. All the other partitioning combinations can guarantee the completion time, which is much smaller than WCET. When using 4 memory controllers and after applying roofline model, the selected partitioning scheme has 5× slowdown **LLC(3,1) MC(2,2)**, while **WCET 4_MC** incurs a 10× slowdown. This gap between WCET and the proposed partitioning grows further for the baseline-2 system that implements 1 memory controller. The selected partitioning scheme **LLC(2,2) MC_TP** incurs 5× slowdown, while **WCET 1_MC** incurs a massive 20× slowdown.

Using the roofline model, we further classify the benchmarks as following: RADIX, MM, and CHOLESKY are compute bound, while FFT is memory bound. Figure 4 shows the sensitivity of benchmark combinations to different partitioning schemes. The notion of this plot is to justify that using the proposed portioning combinations with roofline model can always eliminate the worst results. In order to justify that, the partitioning eliminated, as well as the worst partitioning combinations are listed in Figure 5. Thus for all combinations, the worst case can always be eliminated correctly. For example, for CHOLESKY-MM, **LLC(1,3) and LLC(3,1)** are eliminated, while **LLC(3,1) MC(3,1)** gives the worst completion time. In Figure 4, it is also shown that LLC(2,2) in general gives better completion time for CHOLESKY-MM. When considering both figures, it is perceptible that when executing two compute bound applications, LLC(2,2) is the most beneficial partitioning scheme, which also meets our perspective.

Figures 6a and 6b show the best partitioning schemes when using baseline-1 (4 memory controllers) and baseline-2 (1 memory controller), respectively. In both the figures, the best partitioning scheme for each benchmark combination has a better completion time than the WCET. These results also meet our perspective that compute bound applications need more LLC slices, while memory bound applications need more memory controller bandwidth. However, in certain cases, due to the inherent completion time difference, the application with more work dominates the overall completion time. Such as in



(a) Best partitioning schemes when using 4 memory controllers



(b) Best partitioning schemes when using 1 memroy controller

Fig. 6: Best schemes for spatial partitioning of LLC and spatial/temporal partitioning of memory controller(s)

CHOLESKY-FFT, when the memory controller is temporally partitioned, FFT needs more LLC slices to achieve a better overall completion time. Although CHOLESKY has more slowdown, its completion time is still less than FFT.

Overall, our evaluations show strong potential for reducing the performance overheads of WCET when multiple applications are concurrently executed on a multicore processor. In that respect the proposed spatio-temporal partitioning of the LLC and memory controller interference channels guarantee deterministic execution, while consistently delivering better than WCET performance for safety-critical applications.

V. Related Work

Moinuddin et al [19] partitions shared LLC among multiple applications based on each applications utility and the way in which LLC allocation affects its performance. The above scheme known as Utility based Cache Partitioning (UCP) technique is achieved with the help of Utility MONitor Circuits (UMON) circuits which sample a minimal amount of the cache lines having counters to track the hits and misses across each block. The lookahead algorithm [19] allocates ways to each application studying previously recorded applications' performance dependence on increasing the number of ways. Jigsaw [20] addresses the scalability issues for UCPs for CMPs using software-defined LLC shares by the controlling the partitions. Jigsaw uses UMONs to partition the LLC which is divided into managed and unmanaged regions. The managed region is only partitioned periodically after each interval. After each interval, when each application borrows space Peek-ahead algorithm.[20] Ubik proposes a dynamic partitioning scheme that predicts and exploits the transient behavior of latency-critical applications while increasing the LLC space for batch(throughput) applications. All these schemes introduce fine-grain LLC partitioning, they all have hardware overhead.[21]. All these schemes also execute a run-time

partitioning algorithm which modifies the space allocated to these application in the shared last level cache. For a small-scale embedded multicore, we propose a simplistic static spatial partitioning scheme for the LLC interference channel. However, we plan to explore the fine-grain mechanisms as future work.

In Wang et al.[17] each bank in the memory controller is statically allocated to each application. The DRAM transaction scheduler temporally allocates shared memory controller bandwidth across various security application domains. The static periodic time interval is decided based on the frequency of DRAM row-buffer access and dead-time between 2 subsequent accesses. The disadvantage of this approach is that the each application needs to wait while the channel bandwidth is time multiplexed even though the application which has control of the channel doesn't have any request. So, there is under-utilization of the shared channel to the DRAM.The worst-case latency is double the optimal case where the idle-time is more for the memory controller. Muralidhara et al.[22] dynamically partitions the DRAM banks based on the LLC miss characteristics and row-buffer locality of each applications which resolves interference at the DRAM but does not resolve interference at the shared memory controller in a multicore. The above mentioned works periodically reallocate the shared memory controller bandwidth constantly but do not remove the interference in the interference channels by overlooking shared LLC.

Nowotsch et al.[10] evaluates a multicore system in an avionics platform and stresses the interference problem in shared resources in a multicore. Bui et al.[11] treats the cache partitioning as an optimization problem and uses genetic algorithm globally and simulated annealing locally to temporally partition the cache with a cyclic scheduler. However, it only claims to reduce interference and improve schedulability. Slijepcevic et al.[12] propose a Time-Randomized(TR) cache approach where a probabilistic timing analysis helps in controlling shared LLC eviction frequency. Time Randomization helps in improving the performance of shared cache alone but does not help in co-ordinating the interference with other shared resources. All these approaches are suitable for a uniprocessor environments only. In this work, we propose a cache and memory controller partitioning scheme such that it eliminates interference in the shared channels.

VI. Conclusion

We propose an approach to remove on-chip interference via a set of methods to spatio-temporally partition shared multicore resources. This approach guarantees deterministic execution of safety-critical applications on a multicore, while delivering better than WCET performance. This paper is a step towards fulfilling the FAA requirements for deployment of multicores in safety-critical system, such as avionics and automobiles.

References

[1] Nuzzo, Pierluigi, et al. "A contract-based methodology for aircraft electric power system design." Access, IEEE 2 (2014): 1-25.

[2] Qingchuan Shi; Hijaz, F.; Khan, O., "Towards efficient dynamic data placement in NoC-based multicores," in Computer Design (ICCD), IEEE 31st International Conference on , vol., no., pp.369-376, 6-9 Oct. 2013.

[3] Miller, J.E.; Kasture, H.; Kurian, G.; Gruenwald, C.; Beckmann, N.; Celio, C.; Eastep, J.; Agarwal, A., "Graphite: A distributed parallel simulator for multicores," in High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on , vol., no., pp.1-12, 9-14 Jan. 2010.

[4] Krishna and Poovendran. "Aviation cyberphysical systems: foundations for future aircraft and air transport." Proceedings of the IEEE 101.8 (2013): 1834-1855.

[5] Certification Authorities Software Team(CAST), CAST-32 Multi-core Processors, May 2014(Rev 0) http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf

[6] Cyber Physical Systems Design Challenges Report No. UCB/EECS-2008-8 http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html

[7] NASA Avionics Architectures for Exploration (AAE) and Fault Tolerant Computing, http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20140008709.pdf

[8] Wilhelm, Reinhard, et al. "The worst-case execution-time problemoverview of methods and survey of tools." ACM Transactions on Embedded Computing Systems (TECS) 7.3 (2008): 36.

[9] Samuel, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." Communications of the ACM 52.4 (2009): 65-76.

[10] Nowotsch, Jan, and Michael Paulitsch. "Leveraging multi-core computing architectures in avionics." Dependable Computing Conference (EDCC), 2012 Ninth European. IEEE, 2012.

[11] Bui, Bach Duy, et al. "Impact of cache partitioning on multi-tasking real time embedded systems." Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on. IEEE, 2008.

[12] Slijepcevic, Mladen, et al. "Time-analysable non-partitioned shared caches for real-time multicore systems." Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE. IEEE, 2014.

[13] Yan, Kaige, and Xin Fu. "Energy-efficient cache design in emerging mobile platforms: the implications and optimizations." Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition. EDA Consortium, 2015.

[14] Tilera $TileGx9$, http://www.tilera.com/files/drim_TILE-Gx8009_PB036-02_WEB_7663.pdf

[15] OCTEON II CN68XX Multi-Core MIPS64 Processors , http://www.cavium.com/OCTEON-II_CN68XX.html

[16] Hardavellas, Nikos, et al. "Reactive NUCA: near-optimal block placement and replication in distributed caches." ACM SIGARCH Computer Architecture News. Vol. 37. No. 3. ACM, 2009.

[17] Timing Channel Protection for Memory Controllers Yao Wang, Andrew Ferraiuolo, and G. Edward Suh, *20th IEEE International Symposium on High Performance Computer Architecture*(HPCA), February 2014.

[18] Daniel Sanchez and Christos Kozyrakis.Vantage: Scalable and Efficient Fine-Grain Cache Partitioning, *Proc. of the 38th annual International Symposium in Computer Architecture* (ISCA-38),2011

[19] Moinuddin. K Qureshi, Yale N. Patt. Utility Based Cache Partioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*(MICRO), 2006

[20] Beckmann, Nathan and Sanchez, Daniel Jigsaw: Scalable Software-defined Cache *Proc. of International Conference on Parallel Architectures and Compilation Techniques* (PACT-22), 2013

[21] Harshad Kasture and Daniel Sanchez.Ubik:Efficient Cache-Shariing with strict QoS for Latency-Critical Workloads *Proceedings of the seventeenth edition of ASPLOS on Architectural support for programming languages and operating systems*,(ASPLOS),2014

[22] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda,Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning *Proceedings of the 44th International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, December 2011*

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Intl Symposium on Computer Architecture, 1995.*