Empirical Evidence for MPSoCs in Critical Systems: The Case of NXP's T2080 Cache Coherence

Roger Pujol[*†], Hamid Tabani[†], Jaume Abella[†‡], Mohamed Hassan[§] and Francisco J. Cazorla[†‡]
[*] Universitat Politecnica de Catalunya, Spain [†] Barcelona Supercomputing Center, Spain
[‡] Maspatechnologies S.L., Spain [§] McMaster University, Canada

***Abstract*—The adoption of complex MPSoCs in critical real-time embedded systems mandates a detailed analysis of their architecture to facilitate certification. This analysis is hindered by the lack of a thorough understanding of the MPSoC system due to the unobvious and/or insufficiently documented behavior of some key hardware features. Confidence in those features can only be regained by building specific tests to both, assess whether their behavior matches specifications and unveil their behavior when it is not fully known a priori. In this line, in this work we develop a thorough understanding of the cache coherence protocol in the avionics-relevant NXP T2080 architecture.**

I. Introduction

The use of multicore processors and COTS Multiprocessors Systems on Chip (MPSoCs) platforms [5, 17] is settling up in domains like avionics, automotive, and space [10, 14, 9] due to the need for higher performance. However, the complex hardware architecture of modern MPSoCs hinders their adoption in the safety-critical domain, because it is mandatory to achieve a thorough understanding of the system's behavior and how resources are shared (e.g., shared multi-level caches and complex cache coherence) to derive meaningful timing bounds and hence, ensure system predictability and certifiability [3].

Unfortunately, understanding the architecture of COTS MPSoCs is impeded by multiple factors, including the MPSoC complexity, intentionally hidden details for intellectual property reasons, and poor documentation. In this line, some authors already reported different observed behavior of the coherence protocol to that reported in the Technical Reference Manuals (TRMs) [12]. Others reported discrepancies in the hardware monitors between the observed values and the description provided in the TRM [1, 11, 16].

We contend that empirical evidence of the correctness of key multicore features affecting timing should be developed, instead of assuming the correctness of the descriptions in the TRMs. In this work, we develop such empirical evidence for the NXP T2080 SoC [5], which is relevant in the avionics domain [10]. In particular, we show how it can lead to a thorough understanding of one of the most complex architectural features: cache coherence. Apart from capturing the details of the coherence protocol deployed by the SoC, our experiments reveal some behavior that will affect the timing analysis and is not well-documented in the TRMs, including unexpected coherence-related messages, and incomplete and unobvious event monitor behavior.

II. Setup

The NXP T2080 SoC [5] (see Figure 1), assessed for its use in avionics, features four PowerPC e6500 cores [4], each with its private instruction and data cache (IL1 and DL1, respectively) as well as a private Memory Management Unit (MMU). Each core communicates with the shared L2 cache via the cache-core interface (CCI). The L2 cache is shared between all the cores. A "CoreNet" coherence fabric (CCF) provides access to the DDR SDRAM memory controller as well as other interfaces present in the board like the Direct Memory Access (DMA). The L2 has a single port to the CCF whose access is controlled by the Bus Interface Unit (BIU). The L2 cache is the point of coherency in the cluster. DL1 caches contain no modified data as they are write-through. The L2 is inclusive of the DL1 of each core, so if a data line is evicted from the L2 cache, it is invalidated in the corresponding DL1 caches. The T2080 implements a 4-state Modified-Exclusive-Shared-Invalid (MESI) cache coherence protocol. The coherence granularity is a L2 cache line, such that each line has its own coherence state information.

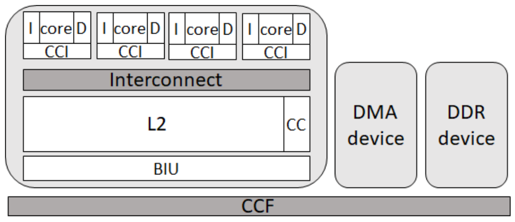

Fig. 1: Simplified block diagram of the T2080

DMA transfers can 'generate' snoop requests to the L2. This can lead to invalidations of data in the DL1 caches and/or the L2 since this data becomes not up to date with respect to the latest value written by the DMA to memory.

Interestingly, under MESI certain messages are exchanged among coherent elements (caches) to inform/request that a line has changed state in the cache sending the request. A starting hypothesis we made is that no such messages are sent in the T2080 as it is a single-cluster MPSoC and hence, there is a single coherent cache (the DMA has no internal coherent cache). This is a valid hypothesis as previous studies show that cache coherence increases processor energy profile [8], favoring predictability and tightening bounds as fewer messages are sent over the CCF.

A. Observability

Hardware Monitors. Several hardware event monitors in the T2080 [4] provide information about the L2 cache coherence activity, which we show in Table I.

Debugger Support. Using CodeWarrior, the standard IDE for the T2080, we access several flags for each cache line with information about their coherence state: Dirty (i.e. Modified), Valid, Shared, and Exclusive.

B. Experimental Setup

We seek to observe the behavior of the L2 cache when transitioning from one coherence state to another. To that end, we execute a benchmark during the warm-up phase that sets several lines in the L2 cache to a given coherence state. Afterwards, during the execution phase, another benchmark is executed to force the coherence state of those cache lines to change. Such benchmark accesses a subset of the addresses accessed in the warm-up phase. Moreover, the benchmark in the execution phase can access other

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. https://doi.org/10.23919/DATE51398.2021.9474078

TABLE I: Coherent related event counters in for the L2

Counter	Description
L2DA	Number of L2 data accesses.
L2DM	Number of L2 data misses.
L2SH	Number of L2 cache snoop hits.
L2SP	Number of L2 cache snoop pushes.
ESR	Externally generated snoop requests.
L2SM	L2 snoops causing MINT (Modified INTervention).
L2SS	L2 snoops causing SINT (Shared INTervention).
L2RC	Number of L2 reloads from CoreNet.
BL	BLINK requests from L2 to core (e.g. back invalidates)
CL	CLINK requests from L2 to core(CoreNet data forwarding)

addresses not shared with the benchmark executed in the warm-up phase. This is explained on a per experiment basis.

We set a breakpoint right after the warm-up phase (right before the execution phase) to confirm that the state is the one we expect. We also set another breakpoint right after the execution phase to verify that the cache lines have changed to the desired state.

Benchmarks. We use two micro-benchmark types, CPU and DMA, each with read (r) and write (w) variants, resulting in 4 combinations (CPUr, CPUw, DMAr, DMAw).

The CPU benchmarks perform 128,000 accesses mapped to a set of either 4K or 8K different addresses. In the former case it accesses 256KB of data with a 64B granularity, so it accesses 256KB/64B=4,096 cache lines; and in the latter it accesses 512KB or 512KB/64B=8,192 lines. We refer to these benchmarks as CPUx(4K) and CPUx(8K) respectively, where 'x' is either 'r' or 'w' and the number of accesses is omitted as it is the same in all variants.

The DMAr benchmark reads data from a memory address range including the one used by the CPU benchmarks and writes it to a not overlapping memory address range, while the DMAw benchmark reads from the non-overlapping range and writes to the overlapping one. Both benchmarks, transfer either 2MB or 4MB of data. Hence, they perform 32,768 or 65,536 64B accesses to the overlapping range and the same number to the non-overlapping. We refer to these benchmarks as DMAx(32K) and DMAx(64K) respectively. Note that DMAr and DMAw perform necessarily both read and write operations.

Workloads. A workload comprises the execution of one or several benchmarks during the warm-up phase and one during the execution phase. For instance, $[CPUw(4K), DMAr(32K); DMAw(32K)]$ shows that during the warm-up CPUw(4K) and DMA4r(32K) are executed one after the other; and in the execution phase the DMAw(32K) is executed. Note that, as the DL1 cache is write-through and the L2 is the only coherent cache, no coherence messages are sent to handle the shared data between tasks running simultaneously in the cores. Hence, we perform no experiment to capture this scenario.

III. Analysis

A. Coherence Protocol Analysis

Modified to Shared. During the warm-up phase, we run CPUw that causes the data to be dirty in L2. In the execution phase, we run DMAr to force snoops from the CoreNet addressed to the L2 that has to (i) send the data to the main memory, (ii) clean the dirty flag and (iii) move the cache line to the shared state. We evaluate four scenarios: (A) [`CPUw(4K); DMAr(64K)`], (B) [`CPUw(8K); DMAr(64K)`], (C) [`CPUw(4K); DMAr(32K)`], and (D) [`CPUw(8K); DMAr(32K)`].

As we start measuring right after the warm-up (i.e. the execution of CPUw), there can be few accesses that are pending to L2, causing the few L2DA, as shown in (M2S) case in Table II (*conclusion-L2A*). A GetS/GetM message is sent for every address read or written by the DMAr [1]. Hence, the number of expected ESR is 65,536 in (A) and (B) and 32,768 in (C) and (D), matching the observed results in Table II (M2S) (*conclusion-ESR*). The GetS messages hit in the L2 for the memory locations that were accessed previously by the CPUw, which results in 4K[2] L2SH in (A) and (C) and 8K[3] in (B) and (D) (*conclusion-L2SH*).

TABLE II: PMCs for all transitions.

State		L2DA	L2SH	L2SP	ESR	L2SM	L2SS	L2RC	L2DM	BL	CL
M2S	A	13	3968	3968	65536	3968	0	6432	0	0	6447
	B	12	8064	8064	65536	8064	0	6418	0	0	6418
	C	13	3968	3968	32768	3968	0	3226	0	0	3241
	D	13	8064	8064	32768	8064	0	3226	0	0	3242
M2I	A	14	3968	0	65536	0	0	6418	0	3968	6418
	B	14	8064	0	65536	0	0	6418	0	8064	6418
	C	14	3968	0	32768	0	0	3213	0	3968	3213
	D	14	8064	0	32768	0	0	3212	0	8064	3212
E2M	A	127999	0	0	0	0	0	0	0	0	0
E2S	A	12	3968	3968	65536	0	3968	6418	0	0	6418
	B	12	8064	8064	65536	0	8064	6418	0	0	6418
	C	12	3968	3968	32768	0	3968	3212	0	0	3212
	D	12	8064	8064	32768	0	8064	3212	0	0	3212
E2I	A	13	3968	0	65536	0	0	6418	0	3968	6418
	B	13	8064	0	65536	0	0	6419	0	8064	6419
	C	13	3968	0	32768	0	0	3213	0	3968	3213
	D	13	8064	0	32768	0	0	3212	0	8064	3212
S2M	A	127998	0	0	0	0	0	3968	0	3968	0
	B	127998	0	0	0	0	0	8064	0	8064	0
I2M	A	127998	0	0	0	0	0	3968	3968	0	0
	B	127998	0	0	0	0	0	8064	8064	0	0
I2E	A	128001	0	0	0	0	0	3968	3968	0	7936
	B	128001	0	0	0	0	0	8064	8064	0	16128

Unexpectedly, the L2 responds to each of these snoop hits in the L2 by requesting other coherent devices to set such lines as shared, but there is no other coherent device in the T2080. This matches L2SP and L2SM values. Further conclusions about these events and counters are discussed in Section III-B. DMAr does not reload any data from the CoreNet to the L2. However, we observe L2RC and CL report around 6 and 3 thousand events for cases (A)-(B) and (C)-(D) respectively. This activity corresponds to the accesses performed by the core by polling the DMA controller to know when DMAr has finished as we describe later on in Section III-B, (*conclusion-L2RC_CL*).

Modified to Invalid. We run CPUw in the warm-up phase causing the data to be dirty in L2. In the execution phase, we run DMAw to force snoops from the CoreNet addressed to the L2 to invalidate its data because the DMA is overwriting it. We evaluate four scenarios: (A) [`CPUw(4K); DMAw(64K)`], (B) [`CPUw(8K); DMAw(64K)`], (C) [`CPUw(4K); DMAw(32K)`], and (D) [`CPUw(8K); DMAw(32K)`].

Conclusion-L2A, *conclusion-L2RC_CL* and *conclusion-ESR* apply to this scenario as shown in Table II (M2I). The GetM messages sent for each DMA write operation that hit in the L2 for the memory locations – that were accessed previously by the CPUw – make L2 invalidate those lines. Hence, we expect that each line invalidated has to send a back invalidate request, so BLINK matches L2SH as shown in Table II (M2I).

Exclusive to Modified. We run CPUr in the warm-up, loading the data to the L2, which changes the state to

[1]Recall that DMAr also performs writes that result in GetM requests.

[2]In reality, we expect $4,096 - 128$ and $8,192 - 128$ respectively, as we perform 31 and 63 iterations respectively of 128 accesses each.

Exclusive. In the execution phase, CPUw changes the state to *Modified*, and since it is the only owner of the data, it does not have to invalidate anything nor wait for a response before writing. We evaluate one scenario: (A) [`CPUr(4K)`; `CPUw(4K)`]. As the task in the execution phase runs in the CPU (CPUw), we see 128,000 L2 accesses (L2DA), see Table II (E2M). Since L2 is the only data owner, it does not send any message and thus, no requests are generated.

Exclusive to Shared. During the warm-up, we run CPUr that causes the data to be loaded to the L2 in Exclusive state. In the execution phase, we run DMAr to force snoops from the CoreNet addressed to the L2 to send the data to the main memory and move the cache line to the shared state. We evaluate the following four scenarios: (A) [`CPUr(4K)`; `DMAr(64K)`], (B) [`CPUr(8K)`; `DMAr(64K)`], (C) [`CPUr(4K)`; `DMAr(32K)`], and (D) [`CPUr(8K)`; `DMAr(32K)`]. As shown in Table II (E2S), the results are almost the same as in Modified to Shared transition, where the only difference is switching L2SM to L2SS, which makes sense because the original state is not Modified in this case.

Exclusive to Invalid. During the warm-up, we run CPUr that causes the data to be loaded to the L2 in Exclusive state. In the execution, we run DMAw to force snoops from the CoreNet addressed to the L2 to invalidate the data. We evaluate the following four scenarios: (A) [`CPUr(4K)`; `DMAw(64K)`], (B) [`CPUr(8K)`; `DMAw(64K)`], (C) [`CPUr(4K)`; `DMAw(32K)`], and (D) [`CPUr(8K)`; `DMAw(32K)`].

As shown in Table II (E2I), *conclusion-L2A*, *conclusion-L2RC_CL* and *conclusion-ESR* apply. For every request made by the DMAw, a GetS/GetM message is sent, with GetM messages hitting in the L2 for the memory locations that were accessed previously by CPUw. Then, the L2 invalidates all the lines where the snoops have hit. This results in back invalidation messages sent to the core for all lines in L2, which results in BL being equal to L2SH.

Shared to Modified. During the warm-up, we run CPUw that causes the data to be dirty in L2, and then we run DMAr, which sends a request for the data and the L2 sends the dirty data to the CoreNet moving to Shared state. In the execution phase, we run CPUw, which sends GetM requests to the CoreNet to invalidate all the valid copies in any potential device with that data set as shared, despite there is none. As a simultaneous modification of this data could have occurred in another device with a shared copy of the data, the L2 sends a data forward request to CoreNet. This request might bring either no new data or an updated copy if a remote modification occurred between the local modification and its notification to other coherent devices. Those unexpected messages to non-present coherent devices are further discussed in Section III-B. We evaluate the following four scenarios: (A) [`CPUw(4K)`,`DMAr(64K)`; `CPUw(4K)`], and (B) [`CPUw(8K)`,`DMAr(64K)`; `CPUw(4K)`].

As in the E2M transition, the task analyzed (CPUw) generates accesses from the CPU as captured by the L2DA counter in Table II (S2M). Each of these accesses triggers the L2 to send a GetM message to ask all other sharers to invalidate their own copy of the shared data to the CoreNet, generating around 4K BL in (A) and around 8K in (B). As explained, since other coherent devices could be performing simultaneous modifications of the shared data, the L2 performs 4K L2RC in (A) and around 8K in (B). Those L2RC receive no answer since there is no other coherent device in the platform.

Shared to Invalid. This scenario is exactly the same as Exclusive to Invalid transition in all the results. The only difference is how warm-up the cache to get it to Shared state (CPUw+DMAr).

Invalid to Modified. In the warm-up, we run CPUr that causes the data to be loaded in L2, and then we run DMAw to overwrite and invalidate it in the L2. In the execution, we run CPUw, the CPU misses in the L2, and triggers it to send GetM requests to CoreNet, which grants modification access to the L2. We evaluate four scenarios: (A) [`CPUr(4K)`,`DMAw(64K)`; `CPUw(4K)`], and (B) [`CPUr(8K)`,`DMAw(64K)`; `CPUw(8K)`].

In this case, the task analyzed (CPUw) performs 128k accesses to the L2 that are shown in the L2DA counter (Table II I2M). From these accesses, only around 4K in (A) and 8K in (B) miss in the L2 (L2DM), and these misses cause the reloads in the L2 from CoreNet (L2RC). Since DMA does not own the data, it does not have to back invalidate the data (thus BL is 0), and also, since the CPU is overwriting the cache lines, it does not have to wait for the data to be forwarded (thus CL is 0).

Invalid to Exclusive. During the warm-up, the CPUr causes the data to be loaded in L2. Then we run DMAw to overwrite and invalidate it in the L2. In the execution, CPUr misses in the L2 and triggers it to send GetS requests to CoreNet, which sends the data exclusively to the L2. We evaluate the following scenarios: (A) [`CPUr(4K)`,`DMAw(64K)`; `CPUr(4K)`], and (B) [`CPUr(8K)`,`DMAw(64K)`; `CPUr(4K)`].

This case is very similar to the I2M transition, however, here the L2 has to receive the data (since the CPU is reading it), so the data is forwarded by the CoreNet, thus increasing CL. Note that CL value is 2x that of the L2DM and L2RC. Section III-B provides light on this behavior.

Invalid to Shared. This transition requires another component outside the core cluster to have the data in exclusive, shared, or modified state when cores access the data. Since such a component does not exist in the T2080, such a transition cannot occur.

B. Hardware Monitor Analysis

L2SM, L2SS, and L2SP. Coherence messages to remote devices can either invalidate their copies if data is modified locally or mark them as shared if a remote device reads some data present in L2 either as exclusive or modified. The latter matches exactly L2SP, as shown in M2S and E2S transitions, meaning that those events are snoop pushes, whereas invalidations are not counted as snoop pushes. Moreover, L2SM counts exactly M2S transitions and L2SS E2S transitions across all possible state transitions. Interestingly, while those messages are expected when there are multiple coherent devices, the NXP T2080 has just one, so it should not send those messages. A potential hypothesis is that since the e6500 core cluster is also used in multi-cluster platforms (e.g. NXP T4240), it is reused with no modifications in the T2080 despite generating unnecessary messages. This would also explain the unexpected messages to other non-existing coherent devices in the S2M transition.

L2RC and CL when DMA runs during the execution phase. Running DMAr or DMAw in the execution phase, we observe that L2RC and CL count events, and such events do not relate to the coherence protocol. To handle DMA transfers, which are asynchronous with the core, we build a solution that polls the DMA controller, specifically a non-cacheable control register, to identify when the DMA

TABLE III: L2RC and CL specific experiments under E2I

		DMA 32K			DMA 64K					
	chunk	L2RC	CL		L2RC	CL	L1M	L2DemA	L2DA	L2DM
A	64B	5599	5599	C	11359	11359	11369	11371	12	0
B	256B	3282	3282	D	6419	6419	6568	6575	12	0

transfer is complete. Every read operation of such register produces CL and L2RC events to fetch its value[3]. To verify this behavior of L2RC and CL, we run DMA transfers with 64B and 256B chunks. Table III shows some results for the E2I scenario. When we double the DMA transfer size from 32K to 64K, L2RC and CL values double since the transfer duration doubles. Thus, 2x number of polls occur. If we decrease chunk size from 256B to 64B, the number of polls increases (but not by 4x) since 4x transactions are needed, but each one of them is faster to transfer 64B instead of 256B of data.

L2DA and L2DM. In these very same experiments, our results also show that L2DA and L2DM are almost 0 despite the number of L2RC and CL. To complete our analysis, we have verified that L2RC and CL values match quite precisely the number of L1 cache misses (L1M), which match load accesses since DMA control register accesses are uncacheable, and L2 demand accesses (L2DemA). L2DemA corresponds to all L2 accesses except prefetch and snoop ones. Hence, access to the DMA control register can be traced end-to-end from the core to the CoreNet with very similar values for L1M, L2DemA, L2RC, and CL. However, L2DA and L2DM only count cacheable data accesses despite, to the best of our knowledge, public documentation does not reflect that casuistic. Hence, undocumented features (e.g. L2DA and L2DM do not count uncacheable data accesses) call for a thorough validation of hardware monitors if used to validate any hardware feature or build any type of certification evidence for critical systems.

CL in CPUx scenarios during the execution phase. As shown in the Invalid to Exclusive scenario, CL is 2x the number of L2DM and L2RC. This occurs since, as explained in [5], despite cache lines are 64B, they are transmitted over 2 cycles since the CoreNet can only transmit up to 32B per transaction simultaneously. Hence, a single L2 cache miss produces 2 CL events to fetch a 64B cache line. This is further corroborated with the data in Table III, where we analyze the behavior of DMA polls, where a single 8B register is fetched, hence needing a single transaction, and thus matching CL and L2RC values.

IV. Related Work

Given its known potential to improve the performance of data sharing, cache coherence is one of the key features studied in recent works [15, 7, 13, 6]. [15] provides an analysis of the MESI protocol and its drawbacks concerning time predictability and demonstrates ways to implement a MESI protocol that better suites critical systems. Authors in [7] proposed invariants that guarantee predictable behavior upon adopting cache coherence in real-time systems and demonstrated the application of these invariants by proposing the predictable MSI protocol (PMSI). [13] proposes a time-based and configurable cache coherence protocol targeting mixed-criticality systems. [6] introduced a solution that improves latency bounds of coherence without degrading the system's performance. These works are in line with this paper's focus, highlighting the importance of cache coherence in modern critical systems.

Authors in [12] analyze the coherence between the different e6500 clusters of the NXP T4240 processor and conclude that it actually implements MESIF instead of MESI as specified in the e6500 TRM [4]. We cannot assess this hypothesis in the NXP T2080 as it has a single e6500-based CPU cluster, and hence, there is only one coherent L2 cache that can have the cache line in S/F state. The L2cache will answer coherence requests whether it keeps the data in S/F state.

[3]An implementation based on interrupts to signal the finalization of DMA transfers would not show this problem. On the other hand, interrupts are known to be problematic to analyze, in addition to impacting tasks execution time as they are not subject to scheduling [2].

V. Conclusions

Our empirical analysis of cache coherence in the T2080 brings some lessons learned. First, we can identify the events triggered by each coherence state transition, providing a clearer understanding of the implemented cache coherence behavior. Second, there are some hardware monitors with ambiguous or incomplete descriptions of the events tracked. And third, we detect unexpected coherence messages for a single L2 coherent cache processor. All these elements help validate the cache coherence protocol itself and allow building other further validation evidence on top of it.

VI. Acknowledgments

This work has been partially supported by the Spanish Ministry of Science and Innovation under grant PID2019-107255GB; the European Union's Horizon 2020 research and innovation programme under grant agreement No. 878752 (MASTECS) and the European Research Council (ERC) grant agreement No. 772773 (SuPerCom); the HiPEAC Network of Excellence; and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] J. Barrera et al. On the reliability of hardware event monitors in mpsocs for critical domains. In *ACM SAC*, 2020.

[2] B. B. Brandenburg et al. Accounting for interrupts in multiprocessor real-time systems. In *RTCSA*, 2009.

[3] G. Fernandez et al. Contention in multicore hardware shared resources: Understanding of the state of the art. In *WCET Workshop*, 2014.

[4] Freescale semicondutor. e6500 Core Reference Manual. https://www.nxp.com/docs/en/reference-manual/E6500RM.pdf, 2014. E6500RM. Rev 0. 06/2014.

[5] Freescale semicondutor. QorIQ T2080 Reference Manual, 2016. Also supports T2081. Doc. No.: T2080RM. Rev. 3, 11/2016.

[6] M. Hassan. Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems. In *ECRTS*, 2020.

[7] M. Hassan et al. Predictable cache coherence for multi-core real-time systems. In *RTAS*, 2017.

[8] M. Loghi et al. Exploring the energy efficiency of cache coherence protocols in single-chip multi-processors. In *GLSVLSI*, 2005.

[9] O. Notebaert. On-Board Payload Data Processing requirements and technology trends. In *OBDP Workshop - ESA/ESTEC*, 2019.

[10] D. Radack et al. (Rockwell Collins). Civil Certification of Multi-core Processing Systems in Commercial Avionics, 2018.

[11] N. Semiconductors. Chip Errata for the i.MX 6SLL. Document Number: IMX6SLLCE, 2019.

[12] N. Sensfelder et al. On how to identify cache coherence: Case of the NXP qoriq T4240. In *ECRTS*, 2020.

[13] N. Sritharan et al. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *RTSS*, 2019.

[14] K. Suleman. Intel paves the road for BMW's iNEXT autonomous cars in 2021. 2017.

[15] S. Uhrig et al. MESI-Based Cache Coherence for Hard Real-Time Multicore Systemsgh. In *ARCS*, LNCS, 2015.

[16] Xilinx. Zynq UltraScale+ MPSoC, APU - PMU Counter Values Might Be Inaccurate When Monitoring Certain Events. Document Number: AR# 68878, 2017.

[17] Xilinx. Zynq UltraScale+ Device Technical Reference Manual. https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf, 2019. UG1085 (v2.1).