Design and Implementation of a Time Predictable Processor: Evaluation With a Space Case Study*

Carles Hernández†[1], Jaume Abella‡[2], Francisco J. Cazorla[3], Alen Bardizbanyan[4], Jan Andersson[5], Fabrice Cros[6], and Franck Wartel[7]

1 **Barcelona Supercomputing Center (BSC), Barcelona, Spain**
`carles.hernandez@bsc.es`

2 **Barcelona Supercomputing Center (BSC), Barcelona, Spain**
`jaume.abella@bsc.es`

3 **Barcelona Supercomputing Center (BSC) and IIIA-CSIC, Barcelona, Spain**
`francisco.cazorla@bsc.es`

4 **Cobham Gaisler, Gothenburg, Sweden**
`alen.bardizbanyan@gaisler.com`

5 **Cobham Gaisler, Gothenburg, Sweden**
`jan@gaisler.com`

6 **Airbus Defense and Space, Toulouse, France**
`fabrice.cros@airbus.com`

7 **Airbus Defense and Space, Toulouse, France**
`franck.wartel@airbus.com`

Abstract

Embedded real-time systems like those found in automotive, rail and aerospace, steadily require higher levels of guaranteed computing performance (and hence time predictability) motivated by the increasing number of functionalities provided by software. However, high-performance processor design is driven by the average-performance needs of mainstream market. To make things worse, changing those designs is hard since the embedded real-time market is comparatively a small market. A path to address this mismatch is designing low-complexity hardware features that favor time predictability and can be enabled/disabled not to affect average performance when performance guarantees are not required. In this line, we present the lessons learned designing and implementing LEOPARD, a four-core processor facilitating measurement-based timing analysis (widely used in most domains). LEOPARD has been designed adding low-overhead hardware mechanisms to a LEON3 processor baseline that allow capturing the impact of jittery resources (i.e. with variable latency) in the measurements performed at analysis time. In particular, at core level we handle the jitter of caches, TLBs and variable-latency floating point units; and at the chip level, we deal with contention so that time-composable timing guarantees can be obtained. The result of our applied study with a Space application shows how per-resource jitter is controlled facilitating the computation of high-quality WCET estimates.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the PROXIMA Project grant agreement no. 611085 (`http://www.proxima-project.eu`). This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P and the HiPEAC Network of Excellence.

† Carles Hernández is jointly funded by the Spanish Ministry of Economy and Competitiveness and FEDER funds through grant TIN2014-60404-JIN.

‡ Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

Keywords and phrases processor design, performance guarantees, multicore, industrial case studies, application of real-time technology in realistic systems

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.16

1 Introduction

Software is becoming the main competitive advantage in embedded real-time products fuelled by the goal of achieving autonomous (i.e. software-controlled) vehicles in market sectors such as automotive, aerospace and railway. In this line, software increasingly implements more complex functionalities with relentless demands for guaranteed computing power across different domains [18, 22]. This has motivated high-performance processor chip manufacturers (e.g. Intel, NVIDIA, and ARM) to start adding time-predictable features in their processor designs [46, 16]. In the same line, processor companies already targeting the embedded real-time domain, e.g. Infineon and Cobham Gaisler, have been motivated to evolve very rapidly from simple micro-controllers to more advanced processor designs [23, 10].

The guaranteed performance requirements of real-time systems challenges the adoption of advanced performance-improving hardware features: as resources become more statefull and interact in more complex ways, deriving tight timing bounds is more difficult. Furthermore, this complicates providing timing analysis techniques with information about hardware behaviour. For static timing analysis this includes access delays of hardware resources which are increasingly hard to derive from manuals, forcing practitioners to stick to measured values [33]. For measurement-based timing analysis (MBTA) assessing whether the execution scenarios captured at analysis effectively cover those bad (worst) conditions that can arise at operation requires dealing with more and more hard-to-track low-level hardware details.

Overall, the quality of the Worst-Case Execution Time (WCET) estimates derived with MBTA (the focus on of this paper and widely adopted in the real-time domain [48, 49]) depends on the ability of the user to build test scenarios at analysis time in which program's execution conditions are close to those that can lead to the WCET during operation. This requires capturing the impact of the sources of jitter (SoJ) in the measurement observations taken at the analysis phase. This ability had by users with simple hardware, diminishes with the advent of more complex hardware: users are increasingly forced to deal with low-level hardware SoJ (e.g. requests alignment and cache mapping), while their real focus is on problems at higher levels of abstraction (e.g. algorithm and end-to-end models). Users neither have the will, (and in many cases) nor the means to exercise this level of control on low-level hardware. This is in contrast to other high-level SoJ, such as execution path coverage, for which clear metrics are defined (e.g. DC and MC/DC) and tools exist to help the user to reach a given target coverage. Hence, solutions that help increasing confidence on measurements without requiring the user to deal with processor internals are fundamental to enable the use of more complex processors in real-time embedded domains.

In this applied study, we present the lessons learned in the PROXIMA EU project [38] designing and implementing LEOPARD (LEON-based probabilistically analyzable processor design), a 4-core processor based on Cobham Gaisler's LEON processor family (deployed in the Space domain). LEOPARD's design exposes the jitter of micro-architectural resources so that the execution time measurements taken at analysis factor in the impact of those resources. LEOPARD helps the user providing evidence, as needed for safety standards, that analysis-time execution scenarios upperbound those that can arise during operation. This is achieved by introducing several low-complexity features in the baseline processor that can be activated/deactivated to reduce impact on average performance. This helps the ultimate goal of having (from the hardware point of view) one design that fits the requirements of several domains and increasing the cost-effectiveness of MBTA since it reduces its application costs while helps achieving the level of confidence required by the domain prescriptions [2]. LEOPARD identifies and attacks the following low-level SoJ.

- The baseline floating-point unit takes variable latency for some operations depending on the particular values operated. To control this SoJ with standard MBTA, the user would need to control the particular values operated at analysis ensuring their representativeness w.r.t. those that can appear during operation. Instead, in the LEOPARD design all floating-point operations are made to work on their respective worst latency, making their impact on execution time to be captured in the analysis-time measurements.
- The use of cache-memory resources (i.e. the data and instruction cache and Translation Lookaside Buffers, TLBs) requires the end user to control memory allocation of code/data, and hence their cache layout at analysis, so that it is the same as during operation. However, even small variations in the order in which the object files are linked together and in other elements of the memory layout (e.g. environmental variables) may significantly affect memory layout – which hence must be controlled by the user. LEOPARD removes this requirement by implementing random placement (and enhancing already deployed random replacement), breaking the dependence among memory allocation of data/code and cache layout. As a result, by performing enough runs [31], the end user can probabilistically assess the impact caches have on execution time.
- At the chip level, we propose an AMBA-compatible time-composable random arbitration to handle contention. To balance time-composability and WCET tightness, we implement a credit-based random (permutation) arbitration policy that randomizes the impact of contention on request's timing behaviour while preserving fairness across cores.
- For the shared L2, we (1) implement random placement and replacement; and (2) assign different cache ways to the different cores (as supported in other architectures) to control contention, while preserving the effective management of cache coherence in the L2 cache.
- Further, we developed high-speed transparent Ethernet tracing features to simplify validation and verification and applicability of industrial timing analysis tools and methods.

Results with stressing applications and a space case-study from Airbus Defense and Space show that LEOPARD's performance guarantees are significantly better than those that could be achieved with Commercial off-the-shelf (COTS) LEON processors. LEOPARD also preserves average performance to the levels of the baseline design. Finally, implementation results show that LEOPARD incurs low area and delay overheads to achieve timing predictability and high-performance tracing capabilities.

The rest of the paper is organized as follows. Section 2 provides some background on MBTA and hardware design favoring it. Section 3 and Section 4 describe the baseline processor and the changes implemented to ease timing analyzability at core level and at chip level, respectively, along with some tracing support enhancements. Section 5 evaluates LEOPARD ability to control identified SoJ. Section 6 presents the most relevant related works. Concluding remarks are presented in Section 7.

2 Background and MBPTA requirements

2.1 Safety Standards

Criticality originally emanates from functional safety, with several existing safety-related standards in different domains: the generic IEC61508 and domain specific variants of it:

ISO26262 in automotive and EN-50126/50128/50129 in the rail domain; and others like ECSS-Q-ST-80C in Space and DO-178C/DO254 for software/hardware aeronautics.

A task overrun should never lead to an unsafe state of the system, which would mean a bad-designed safety solution. Instead, a safety process is defined (according to the corresponding standard) covering the definition of safety goals and requirements, and a safety strategy in general, to mitigate the risk that hardware or software misbehaviour causes a system failure. As the criticality of the software component under analysis increases, more mechanisms are put in place (replication, online monitoring, watchdog) to detect and react to undesired situations.

Many standards require hardware to provide means to demonstrate sufficient independence between different software units. Partitioning mechanisms and monitors are the two preferred means to reach these goals. The use of multicore processors, however, complicate this approach since, although time and space partitioning is achieved [1] (i.e. each task/partition is assigned slots in which only it can use the CPU and it cannot modify another partition's memory space and vice-versa), timing interference is not easily prevented. New requirements are imposed on the hardware and software such as controlling sources of jitter (interference channels in CAST32-A [9] for aerospace).

2.2 Timing

Predictability defines the ability of predicting (a priori) when an event or set of events will occur. While in general in real-time systems predictability is understood as determinism, it has been shown that predictability can be also achieved in probabilistic terms [8].

MBTA involves an operation phase in which the system is deployed, and a analysis (pre-deployment) phase comprising several test campaigns in which the application is run on the target hardware. The goal of MBTA is to derive WCET estimates from the execution runs of the program performed at analysis and provide evidence that those estimates hold valid during the operation of the system. This requires that the execution conditions exercised experimentally at analysis capture those worst-case conditions that can occur during operation. Interestingly, when evidence obtained is sufficient, MBTA can be used for high-integrity software, e.g. DAL-A functions in avionics [28].

With MBTA, user's ability to design stressful operation-representative test scenarios plays a fundamental role in the reliability of the derived WCET estimates. High-level SoJ such as path coverage can be tracked and controlled (as presented in the introduction). However, the control the user has on hardware SoJ diminishes with the advent of more complex features since the cost of controlling the entire design space of all hardware SoJ in such complex designs is unaffordable. For instance, i) the impact of FPU jittery operations would require the user to understand which operands result in longer latencies and software support to track operated values (since hardware support does not exist for that); and ii) capturing the execution time variability consequence of different cache layouts would require understanding the impact of cache layout on WCET estimates. However, in general, it is hard for the user to design experiments in which bad (worst) cache layouts are enforced when even small changes in their memory layout may cause significant jitter in the observed timing behaviour [30]. Fixing the memory layout is only possible during very late phases of the development process, going against the incremental software integration principle and the definition of a global (best-) worst-case memory layout for an application comprising several tasks is a generally intractable problem [30].

[1] Partitioning is no yet achieved or within under specific conditions on multicores.

Time composability is another desired property for derived WCET estimates when controlling the impact of SoJ. First, time composability across incremental software integration [30] ensures that early phase WCET estimates (ideally at the unit testing level) hold across integration reducing the risk of costly late detection of timing violations. And second, time composability at the multicore level ensures that the WCET estimate of a task does not depend on its co-runners' load on shared resources. This provides independence across tasks, that can easily be developed by different software providers in integrated systems (e.g. Integrated Modular Avionics, IMA), allowing parallel development and testing.

MBTA requires collecting execution time traces on the target platform. Transparent trace collection also requires hardware support so that time (or performance monitoring counter) readings can be collected without impact on programs execution. Furthermore, code instrumentation can be performed with hardware [14], causing no overhead on program execution time, but with high associated cost, or at software level. The latter, while it is more generic and portable, it can cause the *probe* effect: instrumentation code create discrepancies in terms of timing w.r.t. the non-instrumented code, complicating timing Validation and Verification. A recent work [13] shows that *nop* operations can be used to substitute instrumentation instructions in a way that simplifies qualification/certification and at the same time reduces the impact of instrumentation.

2.3 Requirements

LEOPARD controls the jitter of the different SoJ in two different ways. First, with deterministic bounding by forcing resources to work on their worst (deterministic) latency. And second with probabilistic upperbounding that makes resources have a randomized timing behaviour, resulting in a probabilistic distribution of execution times that hold during operation [26]. Hence, when enough runs are performed probabilistic upperbounds to execution time [31] can be derived. This principle emanates from probabilistic and statistics theory, where a random variable can be modelled based on a sample of observations with increasing confidence and accuracy as the size of the sample grows.

Probabilistic distributions are handled with a variant of MBTA, called measurement-based probabilistic timing analysis (MBPTA). MBPTA, which builds on representative execution time observations (obtained via the mechanisms to control jitter described above), deploys statistical analysis through Extreme Value Theory (EVT) [27]. EVT enables deriving the probability that bad behaviour of several hardware SoJ, whose impact has been captured in the analysis time runs, are triggered in the same run. This is a powerful solution that reduces MBTA application costs not needing the user to design experiments in which bad behaviour of all SoJ (e.g. bus, cache, FPU) are simultaneously triggered. Overall, the requirements to penetrate high-performance hardware designs while facilitating MBTA (in the form of MBPTA) are:

1. Exposing the impact of SoJ so that (i) representative operation-phase execution time measurements are collected during analysis without needing the end user to design complex experiments to control them, and hence enabling deriving high-quality WCET estimates at low cost; and (ii) derived time composable estimates hold across incremental software integration [30] and are independent of contender's load on the shared resources.
2. Incurring low-implementation overhead, specially in terms of processor complexity to minimize the cost of verification.
3. Reducing the impact on average performance by making time-predictable features to be activated/deactivated depending on the time predictability needs of the system instance.
4. Providing high-bandwidth transparent tracing with no interference on program's execution time. Enabling collecting traces from all cores simultaneously for increased observability,

Table 1 Input value examples triggering different latencies for FDIVD and FSQRTD.

Op.	**Lat**	**Input 1**		**Input 2**	
		hexa	**decimal**	**hexa**	**decimal**
FDIVD	15	0xBFF0000000000000	-1.0	0x4000000000000000	2.0
FDIVD	18	0x001ABC0000000010	$3.717(...) \cdot 10^{-308}$	0x3FF000400A07610C	1.00006107(...)
FSQRTD	23	0x4030000000000000	16.0		
FSQRTD	26	0x4008000000000000	3.0		

.

3 Core Design

In this section we focus on the main SoJ at the core level, while we cover chip-level SoJ in Section 4. Both sections first describe the baseline design and then the proposed changes.

3.1 Baseline Design

The baseline design corresponds to an enhanced implementation of a LEON3 [17] resembling the NGMP processor [10], a multicore processor candidate for the European Space Agency missions in the next years.

Pipeline. The processor implements a pipelined architecture comprising the following stages: fetch (F), decode (D), register access (RA), execution of non-memory operations (Exe), DL1 access (M), Exceptions (Exc) and write back (WB). The execution units comprises an integer and a floating-point unit (FPU).

1. The FPU takes a variable latency depending on the particular values operated for divisions (FDIVD) and square roots (FSQRTD). Table 1 provides a summary of those jittery FP operations and their associated jitter.
2. The core incorporates a static branch-always predictor that starts fetching instruction from the branch target address. On a prediction hit, 1 or 2 clock cycles are saved. Under a mispredicted branch, instruction hits in IL1 change LRU replacement history (also hits in the L2), while misses in IL1 and the L2 pollute cache contents.

DL1 and IL1. The target processor comprises first level instruction (IL1) and data (DL1) caches, with the DL1 implementing a write-through no write allocate policy. The bus propagates DL1 and IL1 misses to the L2 cache (see Figure 3) is discussed in Section 4. IL1 and DL1 are 16KB with 4-way set-associative caches with modulo placement and LRU replacement. L1 caches also support cache freezing to ensure that the execution of the interrupt handler will not evict any cache line and when control is returned to the interrupted task, the cache state is identical to what it was before the interrupt.

TLBs and cache coherence support. To support memory translation (and space partitioning) the LEON processor is provided with a Memory Management Unit (MMU) comprising TLBs of 64 entries for instructions and data deploying LRU replacement.

The LEON3 processor uses Virtually indexed, virtually tagged (VIVT) first level caches so that virtual addresses are used for both the index and tag bits. This caching scheme results in fast lookups, since the MMU does not need to be looked up first to determine the physical address for a given virtual address. Since the LEON3 is a shared memory multiprocessor

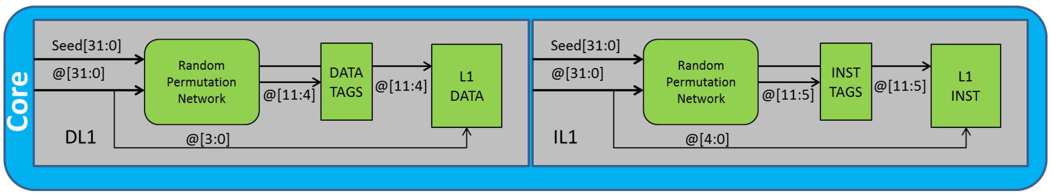


Figure 1 Sketch of the implementation of the Random Modulo technique.

(SMP), on every access to the on-chip bus, the address is snooped by all the caches to check if these data are present in the cache and, consequently, need to get invalidated. To speed up this process, the LEON3 cache also includes the physical tags in a separate SRAM structure so in every access to the on-chip memory, snoop hits are concurrently detected. In case of a snoop hit – a write operation is performed to data stored in the cache – the corresponding cache line is invalidated.

Since VIVT caches suffer from *aliasing*, the LEON3 fixes the cache way size to match (MMU) page size (4KB). With this restriction, that is imposed by hardware, synonyms are enforced to be placed in the same set with modulo placement and thus, can be safely invalidated when a snoop hit is detected. The MMU provides address translation of both instructions and data via page tables stored in memory. When needed, the MMU accesses the page tables to calculate the correct physical address. The latest translations are stored in TLBs. The MMU also provides access control, making it possible to "sandbox" unprivileged code from accessing the rest of the system.

3.2 LEOPARD design

The design presented in this section builds on the design we presented in [20] that focused on single core sources of jitter and included only the implementation of random placement [26] instead of random modulo placement in the L1 caches.

Cache resources. Random replacement has been implemented in IL1 and DL1 caches and TLBs (ITLB and DTLB) building on the *random seeds* provided by the pseudo-random number generator (PRNG) described later in this section. DL1 and IL1 also deploy random placement to release the user from controlling the placement of all programs and memory objects at analysis and during operation. For that purpose we have implemented random modulo (RM) [21], fitting the requirements of the specific processor implementation (e.g. number of cache sets, delay constraints).

RM placement performs a random permutation of the bits used to index the cache set (see Figure 1). By doing so, like modulo, RM retains spatial locality properties. Let W be the way size and A and address such that $(A \mod W) = 0$. With RM any pair of cache line addresses in the range $[A, A + W)$, which are said to belong to the same *segment*, are prevented from conflicting into the same cache set. For instance, if addresses A and B belong to the same cache segment (i.e. $\lfloor A/W \rfloor = \lfloor B/W \rfloor$) and with modulo are mapped to different sets (k_A and k_B respectively), RM randomizes the index bits such that (in every run) with a seed $seed_i$, A is mapped to any (random) set $l_A = set_{rm}^{seed_i}(A)$ and B to $l_B = set_{rm}^{seed_i}(B)$ and l_A and l_B are necessarily different. Hence, RM removes the dependence between memory mapping and cache layout by ensuring that the index permutation covers cache conflicts probabilistically during the analysis phase. During the analysis phase a different seed is employed to cover the cache conflicts that can occur during operation.

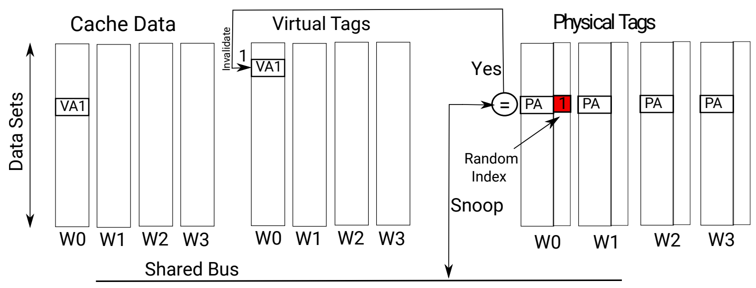


Figure 2 Randomised LEON3 cache configuration. General approach for invalidation.

In the baseline processor, we have detected a single source of timing anomalies. It arises when an instruction i that would has missed in DL1 and hit in L2, actually misses in L2 because before it accesses L2, a younger instruction j misses in both IL1 and L2 and evicts the L2 line where i would hit. Hence, delaying j would allow i to hit in L2 and execute faster. With cache randomization j can evict i line in L2 with a probability $1/S_{L2}$, where S_{L2} stands for the number of L2 cache sets. Hence, if enough runs are performed the impact of this situation would be captured in the measurements. It is part of our future work enforcing IL1 misses to wait for accessing the bus until all older instructions have been resolved in DL1 to avoid any reordering. Users stick to their current practice to handle this situation.

Cache Coherence. The support for cache coherence in a randomised cache design using a MMU introduces some complexities in the cache configuration. In a cache with random placement the index does not only depend on the modulo operation, like in a regular cache with modulo, but also on the upper bits of the address. Then, since virtual and physical addresses referring to the same data have different upper bits (e.g. PA=0x40000004 and VA=0x00000004) the index computed for the virtual address using the random placement function will not necessarily correspond to the one where the physical tag is located. This leads to a conflict for resolving the invalidation of the data affected by snoop hits. Furthermore, the mismatch between indexes makes that synonyms, i.e. two virtual addresses that are mapped to the same physical address, are placed at arbitrary locations in the cache rather than in the same cache set.

To solve this, we have designed a software/hardware solution that requires on one hand, moderate hardware changes in the cache configuration and on the other hand, forcing the OS to flush caches on every context switch. Hardware modifications of the generic solution consist of extending cache contents to keep the randomised index bits that are required to identify the cache set that needs to be evicted in case of a snoop hit. Figure 2 shows the required hardware changes. The randomised index bits are written in the same SRAM structure where the physical tag is and are updated every time new data are fetched into the cache. Finally, the flush on context switch functionality has to be implemented by the OS to ensure that only one address space is present in the cache at a time[2].

Branches. The default configuration of the processor allows issuing fetch requests to the L2 cache on a IL1 miss under branch speculation. As explained before, if the branch is mispredicted, this may pollute IL1 and L2 cache contents, and their replacement history even

[2] Flush on context switch is also a common way to solve the synonym problem in regular cache designs where page size does not match cache way size.

on hits. By using random replacement, replacement becomes stateless and hence, cache hits do not change its state. On the other hand, in order to avoid any cache state modification due to mispredicted branches, we have modified the bitstream to forbid cache misses to be served under speculated branches. This is done by programming the appropriate bit of the ASR17 configuration register. Note that jitter caused by different paths is handled by timing analysis techniques, and hence it is not covered in this hardware paper. An example of one of those techniques is Extended Path Coverage [50] that derives upper bounds of the probabilistic execution time of the complete program under analysis even when the user-provided input vectors do not exercise the worst-case path.

Worst-latency FPU. The implementation of FDIV/FSQRT operations for double precision has been modified so that during the analysis phase, they exhibit a fixed latency that matches their highest latency. In particular, those operations are non-pipelined and iterate in some internal stages of the FPU until the result is produced, thus allowing early termination of some operations. In analysis mode, the early termination signal is inhibited, thus enforcing all those operations to experience their highest latency regardless of the input values operated. At operation time, those operations are allowed to take a variable time depending on the values operated. The net result is that their jitterless timing behaviour at analysis time upper-bounds that during operation, thus releasing the end user from having to control the impact on execution time of the particular values involved. Note that the same approach of delaying execution until its worst case have been used to handle contention in shared resources [36][4].

Creating a source of randomization. A SIL-3 IEC-61508 pseudo-random number generator (PRNG) [3] has been incorporated in the design to feed appropriately the components requiring time-randomized behaviour. The PRNG is based on linear feedback shift registers [6] and consists of a pool of programmable random numbers. In general, the sequence of numbers provided by the PRNG must be long enough to ensure repetition occurs after a period long enough for any potential correlation between the outcomes of the system at different time instants to be probabilistically irrelevant. The degree of randomness of the used PRNG was validated statistically by checking the lack of meaningful patterns, repetitions, imbalance between different values, etc. for a number of bit sequences generated. This was measured with the tests provided by the US National Institute of Standards and Technology [40]. The PRNG provides randomised bits for random replacement: 2 for DL1 (4-way), 2 for IL1 (4-way), and 6 for DTLB and 6 for ITLB (64-entry fully-associative both of them), so 16 bits per core plus few extra bits for the random arbitration in the bus.

4 Chip Design

4.1 Baseline Design

At the chip level the components with the highest jitter impact are the L2 cache controller, the on-chip bus, and the memory controller.

On-chip Bus. An AMBA AHB compatible bus is included in the processor to handle concurrent requests to the different slaves included in the system. The arbiter employs a round-robin arbitration between the different masters, including the processor cores, to determine the one that gets access to the bus. In the baseline design, round-robin is implemented by rotating the priority after every bus transfer.

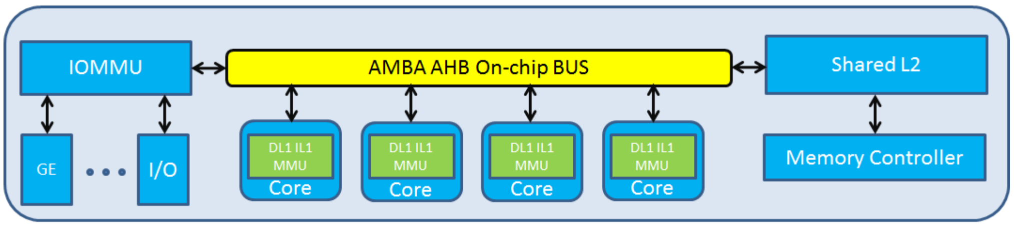


Figure 3 LEOPARD processor block diagram.

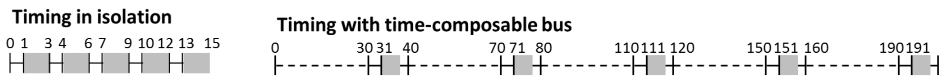


Figure 4 Example of high contention caused by the original scheme with random permutations.

Shared L2 cache. The baseline processor includes a 4-way shared L2 cache. The replacement policy can be configured as LRU (least-recently-used) or master-index (the way in which the replacement occurs is determined by the master index). The cache way is 32KB with a cache line size of 64 bytes. Requests from the cores to the shared L2 cache are arbitrated at the on-chip bus and the bus is kept locked, i.e. no further request are accepted, until the current request is processed by the L2.

Shared Memory Controller. The memory controller included in the baseline design is a DDR2 SDRAM controller with AMBA AHB back-end. The controller interfaces a 64-bit wide DDR2 memory with the L2 and acts as a slave on the AHB bus where it occupies a configurable range of the address space for DDR2 SDRAM access. The memory implements a FIFO with room for two write bursts to maximize throughput, since the second write can be written into the FIFO while the first write is being written to the DDR memory.

IOMMU. To ensure global spatial partitioning is provided efficiently, the baseline LEON3 COTS design already implements an IOMMU. This is significantly important for properly handling direct memory access (DMA) transfers and interrupts [32]. The IOMMU functionality of the core implemented in this processor provides address translation and access protection on the full 4GiB AMBA address space.

4.2 LEOPARD design

On-chip Bus. We modify the arbiter to implement random permutations [25]. Interestingly AMBA standard does not specify any arbitration policy for AHB buses, so our bus is fully AMBA compliant. Further, while the baseline platform we use builds on the AMB AHB specification [7], the modifications we have implemented in the bus can also be applied to more recent bus protocols like the AXI [7]. The random permutation arbiter defines windows with as many slots as arbitrated cores, N_c, with all slots having the same duration. Slots are allocated randomly to cores in each window so each core has *exactly* one slot per window and it can access the bus only during its assigned slots. This allows obtaining time-composable pWCET estimates since no assumption is made on the number and duration of the requests of the other contenders. With random permutations each core gets on average $\frac{1}{N_c}$ of the slots and the maximum waiting time due to contention is shorter than 2 arbitration windows: $MaxContention < L \cdot (2 \times N_c - 1)$, where L stands for the slot duration. The maximum contention occurs when one core is allocated the first slot in one window and the last slot in the following window, since all remaining cores are arbitrated twice in between.

Random permutations is fair granting cores access to the bus with the same frequency. However, it is not fair in terms of the bus time granted to each core. This is so because to achieve composable bounds, slot duration (L) must be large enough to allow the longest potential request to be served. For instance, if the maximum duration of a request is 56 cycles, then $L \geq 56$, otherwise, the longest requests could not be granted access to the bus without using slots belonging to other cores. However, using a long, fixed slot may cause a significant impact in the performance of tasks with requests whose duration is much lower than L. This is better illustrated with the example in Figure 4 that shows how a program with 1-cycle requests runs for 15 cycles in isolation and takes 193 cycles when $N_c = 4$ and $L = 10$: when a given software Unit of Analysis (UoA) accesses the bus, it may have to wait for ($N_c - 1 = 3$) slots assigned to the contenders. Let assume that the UoA accesses the bus in cycle 30 and releases it in cycle 31; further from 31 to 33 it performs some computations and eventually, at cycle 33 the UoA needs to access the bus again. However since it has only 7 cycles remaining in its slot and the duration of the requests is unknown a priori (e.g. it could be 1 cycle for a L2 hit and 10 for a L2 miss in this example), the request is not allowed to proceed and has to wait until the beginning of its next slot in cycle 70. The same scenario repeats for each request so that the last one is granted access in cycle 190, served in 191 and the program completes in cycle 193. As we can see, even if the UoA is granted access during 25% of the time to the bus, its slowdown is 12.9x in a 4-core setup. Note that for the sake of this example we have assumed that slots are allocated homogeneously in time to the core of the UoA. Randomly allocating slots to tasks would bring the very same results as in the example on average.

In order to mitigate the impact that dealing with long requests may have on the quality of time-composable WCET estimates we modify the arbiter to implement a credit-based arbitration (CBA) scheme [43], fitting it to the particular characteristics of the LEON3 multicore and its bus arbiter. CBA allocates each core a given credit (budget) matching $MaxL$ – the longest time a request can occupy the bus, and which can be derived either analytically or by measurements. Then, arbitration is performed across all cores with pending requests and an available budget of exactly $MaxL$ cycles. When a request is granted access to the bus, the budget of the corresponding core is decreased by the bus hold time. For instance, this is implemented by decreasing by 1 the budget of the core using the bus. In parallel, every cycle all cores get their budget increased as shown in Equation 1, where $Budget_i(t)$ stands for the credits (cycles) of core i in cycle t

$$Budget_i(t+1) = \min(Budget_i(t) + 1/N_c, MaxL)\,. \tag{1}$$

The budget assigned to each core saturates at $MaxL$ to prevent the case in which one core spends long time not using the bus and then it hogs the bus during a long time period. This approach reduces the maximum contention experienced by a given core but at the expense of wasting some bandwidth (cores cannot accumulate budget beyond $MaxL$). A similar approach that allows cores to go beyond $MaxL$ was proposed in [5]. Note also that although conceptually the budget is increased by a fraction, this can be implemented by multiplying all factors in Equation 1 by N_c. In that case, when using the bus, the budget should also be decreased by N_c every cycle instead of by 1.

CBA operation is illustrated in Figure 5 where each core always has requests ready and the random permutation generated by the arbiter are as follows: [A, B, C, D], [B, A, D, C], [D, B, C, A], [B, C, A, D], [A, C, B, D], [A, B, C, D]. Requests from cores A, B and C

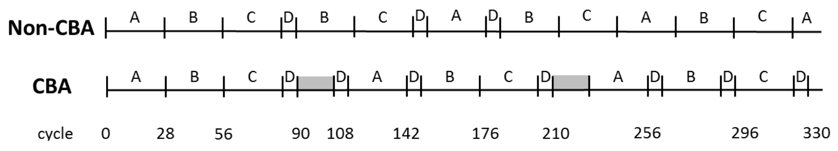


Figure 5 Chronogram showing requests arbitrated with and without CBA. The time scale at the bottom is only approximate since time intervals in the chronogram are not exactly proportional to the time they take for the sake of readability.

take 28 cycles and from D take 6 cycles. We focus in the first 336 cycles (the time needed to hypothetically send 3 28-cycles requests from each core). As shown, without CBA only 3 requests from core D are served in 336 cycles. However, with CBA, in this time frame core D gets 7 requests arbitrated. In the example, the arbiter grants access to a core if it has enough budget. Otherwise, the random list of core ides is searched until a core with enough budget is found. For instance, after the first request of D is served (cycle 90) no core has $MaxL$ budget. At that point the one recovering its budget earlier is D (in cycle 108), so in cycle 108 D is granted access and we skip B and A in the sequence (and also D since it is arbitrated).

Shared L2. The shared L2 has been configured to implement per-way partitioning (master-index replacement). With this configuration each core is provided with one out of the 4 ways available. As for L1 caches, we implement random placement to make WCET estimates hold regardless of the actual memory layout that will be at system deployment. However, unlike for L1 caches, we use hash-based random placement [26] in the L2. The reason is that random modulo [21] forces memory objects to preserve the cache way alignment. For the L1 cache the way size is equal to the OS page size (4KB) and thus, keeping such restriction is not only doable but preserved by default in the general case. However, imposing such way alignment for L2 caches is not realistic since L2 cache ways are generally much bigger than the page size. Finally, we have also implemented random replacement, but when the L2 is fully partitioned, random replacement is not required since each core is only entitled to evict lines from its (single) corresponding way.

Shared Memory Controller. The access latency to shared resources needs to be deterministic upperbounded or randomized to control jitter, relieving the end-user from the burden of controlling how requests from the different tasks align. In our design, no modification was required since the L2 included in the design does not allow any further request to be sent to memory while another request is being served. Also, response time of the memory controller needs to be made constant for each request type to remove dependencies across requests from different cores. This feature was implemented following the same principle as for the FPU unit in the core: delaying requests so that they experience the maximum latency allowed. We are currently in the process of enabling split requests in the bus to increase memory-level parallelism. Besides the functional technicalities, this feature is an important SoJ with potential side effects: at least it is required to use separated request queues for each core in the memory controller. Further, a suitable arbitration mechanism (e.g. random permutations in our case) across cores (so across queues) is also required in the memory controller. Using separate queues, prevents one core to clog the others, despite they have independent cache partitions, as it has been shown for some ARM architectures [45].

High-Speed Tracing. We extended the baseline tracing capabilities to support powerful timing analyses minimizing (or eliminating) timing interferences. In particular, all instructions can be dumped into the corresponding trace buffers in the cores, and sent immediately to a separate DRAM region through a dedicated memory controller using the Debug Support Unit (DSU) interface, thus not interfering with the AMBA bus used for L2 cache and memory accesses. Once instructions information (including instruction and data addresses) is placed in that DRAM memory region, an ad-hoc trace controller reads those traces and sends them asynchronously through the Ethernet interface to the host. In this setup, execution time can only be interfered if the DRAM region is filled in before the trace controller can send the data to the host. However, this allows plenty of room for collecting large traces without creating any interference since the DRAM region is typically large (e.g. 512MB in our experiments), thus allowing tracing full programs or large regions of them.

5 Evaluation

We present LEOPARD evaluation results with benchmarks and a representative space application from Airbus Defense and Space. Hardware overheads numbers for the baseline and LEOPARD configuration are also provided.

Note that the goal of this paper is not to provide a fair comparison of different timing analysis techniques that has been shown a complex task [1]. The the effort required to tailor a static timing analysis tool to the our target platform (e.g. including TLBs, unified caches and shared buses) and the space case study (e.g. to provide flow-facts) is significant. Our goal instead is showing that high-quality performance guarantees can be achieved with relatively small hardware overhead for our relatively complex processor with little impact on average performance and without increasing analysis cost w.r.t. simpler architectures.

5.1 Methodology

MBPTA application. We use MBPTA to derive WCET estimates [47]. The number of measurements of the unit of analysis (UoA) required to apply MBPTA (1) has to guarantee that the events with highest impact are effectively captured in the measurements and (2) has to allow the correct application of EVT [27]. For the experiments conducted in this paper we base on [31, 11].

The common practice for MBTA approaches is to collect end-to-end execution time measurements of the UoA when it is fed with a set of user defined input vectors. An inherent limitation of this approach is that the resulting WCET bounds are only valid for the set of paths for which observations are collected. To overcome this issue, timing analysis techniques, like Extended Path Coverage (EPC) [50], derive upper bounds of the probabilistic execution time of the complete program under analysis even when the user-provided input vectors do not exercise the worst-case path. However, despite that LEOPARD allows deriving pWCET estimates with EPC, in this paper we stick to single-path case results rather than on the software application of EPC since our focus is on the hardware platform.

In order to ensure that in each run the UoA is analyzed under the same *initial conditions*, which upper bound those during operation, we empty the caches right before UoA execution. To do this, we configure the scheduling to ensure that the UoA starts its execution right after a time partition switch and we configure the hypervisor to flush the caches on that time partition switch. When experiments are executed in bare-metal, i.e. without operating system support, we perform this process manually.

For multicore evaluation we use two setups. In the first one, LEOPARD is instructed not to provide time-composable contention bounds (referred to as non-TC mode), see Section 2.2. The second one does force time-composable contention bounds (referred to as TC mode).

5.2 Complexity of LEOPARD features

The different modifications required to achieve MBPTA compliance have been shown to involve little hardware overheads. In terms of complexity, for the FPGA implementation the maximum operating frequency on an Altera DE4 board (100MHz) has been preserved despite the latency overhead introduced by the random placement implementation in the access to the caches. For LEOPARD the *RM* placement [21] effectively minimizes this overhead and facilitates keeping the maximum operating frequency. However, frequency preservation is not guaranteed for different technology libraries and/or different processors implementations, and such evaluation is out of the scope of this paper. Modifications did not have any effect on the most critical path of the different components except for cache memories. First level caches implementing random modulo [21] only increased their delay by a XOR gate to combine address and random seed bits. In the case of the L2 cache, hash-based random placement had a larger impact on critical path due to the higher complexity of its design (few XOR gates and a bit rotator) [26]. Still, impact was not enough to decrease the maximum operating frequency.

In terms of hardware resources occupancy, the baseline design occupied 70% of the resources in the FPGA, whereas LEOPARD occupies 72%, thus showing that all modifications required to achieve MBPTA-compliance and high-speed tracing incur very low overheads.

5.3 Evaluating LEOPARD features

Average performance. Our results show no performance degradation when LEOPARD features are deactivated on the modified RTL based prototype w.r.t. the original unmodified RTL design not containing LEOPARD modifications. Hence LEOPARD does not affect the average performance of applications requiring no performance guarantees.

FPU. To test the effectiveness of the worst-latency FPU unit we have designed four micro-benchmarks (DIVshort, DIVlong, SQRTshort, SQRTlong) that respectively execute short and long latency divisions and square roots. They are executed on two configurations: one with variable-latency FPU operations (labelled COTS) and another with fixed-latency FPU operations (labelled LEOPARD). Execution times when running these benchmarks in isolation (non TC mode) are shown in Figure 6. As shown, under the original setup DIVshort executes faster than DIVlong. Analogously, SQRTshort executes faster than SQRTlong. Therefore, it can be concluded that input values operated impact execution time in the original setup. Conversely, DIVshort and DIVlong have exactly the same execution time on top of the LEOPARD setup. Such execution time matches the execution time of DIVlong on top of the original setup. Results for SQRTshort and SQRTlong show exactly the same behaviour. In fact, the execution time variation between DIVshort and DIVlong corresponds exactly to the number of FDIVD instructions multiplied by 3, which is the difference between short (15 cycles) and long (18 cycles) latencies. The situation for SQRTshort and SQRTlong is analogous. Overall, this experiment validates that modifications in the FPU remove jitter due to the input data operated.

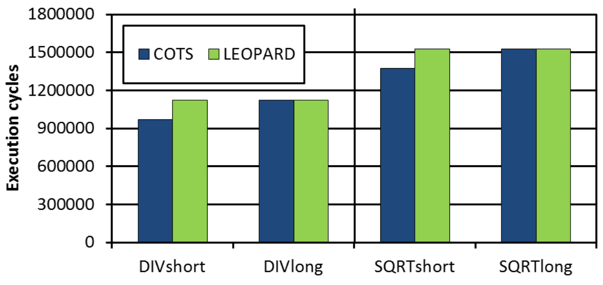


Figure 6 Tests to assess the control of the FPU jitter.

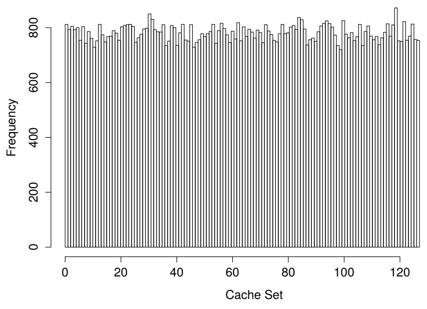


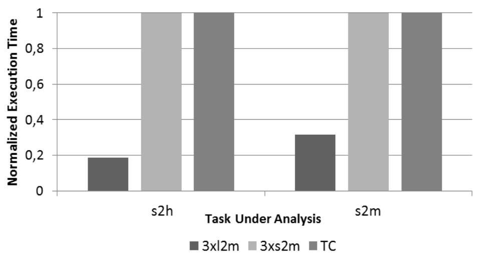
Figure 7 Randomized cache tests.

Cache. To evaluate randomized cache designs in LEOPARD, we test the ability of the cache placement function to cover different memory layouts and thus, capture different cache conflicts associated with the different arrangement of objects in memory. Figure 7 shows how an arbitrary address is mapped to different cache sets across 10,000 different runs in the LEOPARD IL1 (128 sets). Hence, the random placement function maps a given address uniformly across the different cache sets which benefits the application of MBPTA [26].

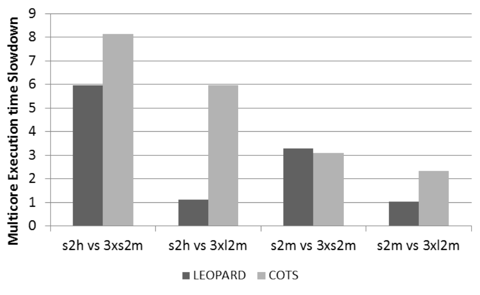
Multicore contention. To test the LEOPARD bus arbitration we assess whether (1) the measured multicore contention with LEOPARD under TC mode effectively upperbounds the highest contention scenario we can create at software level; and (2) whether the slowdown due to bus contention with LEOPARD CBA design is lower than the one of the COTS platform. To do so, we have developed several micro-benchmarks consisting in loads that always miss or hit in the L2 (respectively called *l2m* and *l2h*); and stores that always miss or hit in the l2 (respectively called *s2m* and *s2h*).

Figure 8 (a) shows the results we obtain when running benchmarks *s2h* and *s2m* (the UoA) under three scenarios. Under the former two, the UoA runs in non-TC mode, against 3 copies of *l2m* and *s2m*, respectively. In the third setup *s2h* and *s2m* run under TC mode. We observe that measurements in TC mode upperbound those of the former two scenarios. Interestingly, the second scenario is the worst case we can create in software with each request of the UoA suffering the delay of two requests due to a dirty miss eviction (56 cycles): note that *l2m* generates non-dirty misses and *s2m* dirty misses. We observe the the worst case generated by software matches the time-composable bounds generated by LEOPARD.

Figure 8 (b) shows the benefits brought by the CBA to handle variable latency requests. To that end we show the maximum slowdown due to multicore contention for different benchmarks on the baseline platform and on LEOPARD with TC-mode. For example *s2h*.vs.*3xs2m* represents the case where the UoA is the *s2h* benchmark when it runs against 3 cores executing the *s2m* benchmark. As shown, in all cases except one, LEOPARD significantly reduces the bus contention. For instance, for the *s2h*.vs.3*xl2m* case, contention is reduced from 5.96X to 1.12X. However, for the *s2h*.vs.3*xl2m* case, LEOPARD contention is slightly worse. The reason is that despite CBA is effective to control the interference, it also has a side effect on the UoA preventing it to access the bus when the budget is exhausted. However, we have observed that this only occurs in very extreme cases like in this one, where all instructions from the UoA perform two memory operations, one to write the current data and one to evict data stored in the L2. In this case the budget of the UoA is exhausted in every bus access and the CBA scheme provides no benefit.



(a) Contention upperbound



(b) Slowdown

Figure 8 Results of the experiments carried out to assess LEOPARD's on-chip bus design.

Tracing. Our results show that the maximum achievable *I-point* frequency in terms of cycles between *I-points* is 24 cycles if just 1 core is traced, 51 for 2 cores, 75 for 3 cores and 100 cycles for 4 cores. For the values in which more than one processor is traced, the value corresponds to traces from each available processor. This test has been accomplished by using an infinite loop in which *I-points* arrive at a constant frequency. This might not be exactly representative of a real application since there might be different phases in a real application and if there is enough space between phases to empty the buffers it might be possible to trace more frequent *I-points* for short periods of time. But if the application has loops with very high number of iterations which contain *I-points*, then those results are representative as the *I-points* will arrive at a constant frequency. As expected, with the increasing number of cores the rate at which *I-points* can be traced reduces linearly. This can be explained with the fact that the amount of data that needs to be read from the external DRAM increases linearly with the increasing number of cores.

Since the tracing scheme with external DRAM depends on shared resources (Ethernet link speed, AHB bus, and the host) there is a limitation on the frequency of *I-points* that can be traced real-time. Our evaluations showed that the main bottleneck in the trace bandwidth are GRMON reads and partly the AHB BUS architecture. Increasing the Ethernet buffer size from 4kB to 64kB allowed to increase the read speed from GRMON significantly which resulted in increased bandwidth. But since the FIFOs from the processors compete with the Ethernet reads on the same bus, the read speed during tracing is slower compared to reading the DRAM while there are no traces written to the DRAM. This reduces the potential maximum bandwidth that can be achieved with the available Ethernet link speed. For example, while it is possible to reach on average 550Mbit/s read speed while reading 64MB of data from the external DRAM through GRMON, the read speed reduces to 470Mbit/s when processor(s) create very frequent *I-points*. Also a more dedicated software for large data reads on the host side can improve the bandwidth.

5.4 Space Case-study

The space case study we have used consists of a payload application with a high criticality application, a control loop applying deformations on mirrors, and a low criticality application, responsible of processing images coming from 3 detectors and that requires performance. To fulfill mixed-criticality isolation requirements, applications from this case study run on top of PikeOS hypervisor [44]. The goal of the application is to get better images on the detectors by applying deformations on the mirrors. The output of the image processing is used as an input to compute the displacement of the mirrors; however the control loop of the mirrors

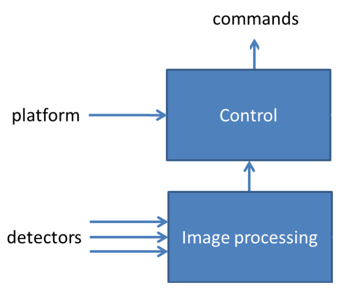


Figure 9 Sketch of the Space App.

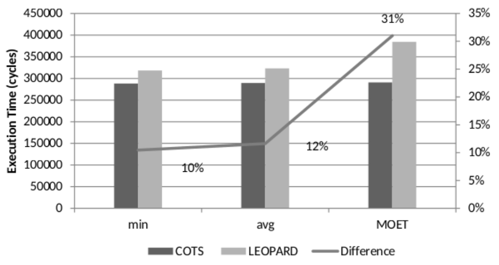


Figure 10 Single-core measured execution times.

can work for some cycles without the input value (with degraded performance). A general view of the application is given Figure 9.

- Image processing partition. The image processing application consists is 3 different tasks each of them pinned to a core. The input data of the image processing partition is preloaded in the memory of the platform to avoid I/O interference during the experiments. This simplifies the observations. The image processing application runs at a higher frequency than the control application. It generates 10 values that are merged by the control partition. The control loop needs to compute the voltage to apply to the mirror motors in order to compensate thermal effects on the main mirror. The image processing has no real-time deadline and thus does not need to be periodic.
- Control partition. The control partition is the destination of the queuing port shared with the image processing partition and corresponds to the UoA of this case study. The control partition uses 10 values to compute the new matrix of voltages to apply to the mirrors. In the case study, there is no mirror to command but these output values are logged to verify the functional behaviour of the application. In the absence of values in the queue or if less than 10 values have been computed by the image processing partition, the control partition reuses old values. Thus, it is tolerant to errors coming from the image processing partition. The control partition runs in isolation in one core.

We first compare single-core performance of the COTS platform and the LEOPARD processor that must not be confused with average performance, for which LEOPARD provides no degradation. In the measurements for the COTS platform, see Figure 10, the impact of SoJ (e.g. FPU and cache jitter) are not factored in the measurements, while the measurements obtained on top of the LEOPARD design expose this jitter to the measurements. For the space case study used in this paper, LEOPARD single-core measurements show that the impact of SoJ on the control application is 12% on average, 10% for the minimum observed execution time and 31% for the maximum observed execution time (MOET).

Figure 11 compares the execution time results of the control application (the highly critical) when is run in three different configurations: (1) isolation, (2) with the image processing application, referred as mixed criticality or MC scenario, and (3) time-composable estimates that are derived in the worst contention scenario. We observe that when the high critical control application runs with the 3-core low critical image processing application, its execution time increases w.r.t. its execution time in isolation (first column). Still measurements collected in TC mode effectively upperbound them (second column).

In Figure 12, where all values are normalized to the MOET of the COTS processor when the control application runs in isolation, we observe that the MOET for LEOPARD with the 3-core image processing application (LEOPARD MC) is 1.87. This value is smaller than that

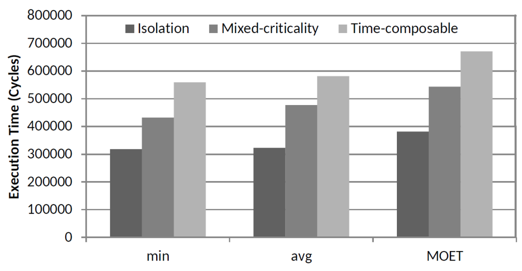


Figure 11 Multicore measured execution times.

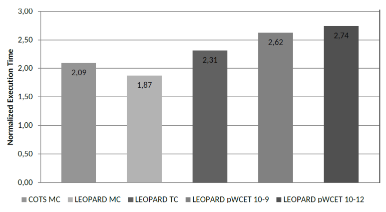


Figure 12 Mixed-criticality execution times and pWCET estimates.

for the same experiment on the COTS platform (COTS MC) that is 2.09. This confirms that the credit-based arbitration allows increasing the performance of the UoA in the presence of inter-core (contention) interference.

The MOET in the highest contention scenario (LEOPARD TC mode) is 2.31x showing that is close to the actual observed values (1.87). These TC values are used as input to MBPTA that results in tight pWCET estimates for exceedance probabilities (per activation of the image processing algorithm) at 10^{-9} and 10^{-12} (considered relevant in previous case studies [47]). Reported pWCET estimate results are in the range 2.6x-2.7x, which are reasonable bounds for a 4-core architecture.

6 Related Work

The timing requirements across different domains (or different systems in the same domain) change and so do the processor designs that have been proposed to support them. In this section we review several research efforts aiming at achieving time-predictability with different processor implementations. Note that we have opted to provide system-level descriptions of related works rather than focusing on related work for each technique, which we have presented in previous sections.

Patmos [42] is a statically scheduled, dual-issue RISC processor specifically designed to facilitate deriving tight WCET estimates with static timing analysis. As both, static timing analysis (STA) and MBTA, are used by industry to derive WCET estimates, having processor designs capturing STA requirements is of interest for both academia and industry. Unlike Patmos that favors WCET over average-performance, i.e. the latter is considered a secondary goal, LEOPARD is designed to incur minimum impact on average performance. LEOPARD also aims at injecting minimum changes on the baseline platform rather than to make a design specifically oriented to WCET.

CompSOC [19] platform offers a virtual platform (resource partition) per application so that applications cannot cause any interference on other co-running applications. LEOPARD, instead, focuses on applications domains for which having bounded interference (rather than no interference) suffices to satisfy timing isolation requirements [37]. In LEOPARD, instead of preventing any interaction among applications – so that one application is forbidden to generate even a single-cycle delay on others – applications are allowed to interfere each other as long as that interference can be bounded.

The FlexPRET [51] processor balances WCET and average performance by designing an specific simultaneous multi-threaded (SMT) architecture that simultaneously executes both 'hard' real-time tasks and 'soft' real-time tasks. Hardware changes (e.g. avoiding data hazards via specific forwarding paths) and a smart instruction scheduler make the execution of soft real-time tasks transparent, so not affecting hard real-time tasks execution. This is similar to parMERASA core architecture [35] that also aims at transparent execution of low-critical SMT threads by for instance preventing non-preemtable multi-cycle operations. FlexPRET also makes other important hardware changes like adding new timing instructions to the baseline RISC-V ISA, while LEOPARD sticks to SPARC v8+ ISA. Further, in PRET caches are replaced by scratchpads to provide more predictability. Overall, different approaches are pursued by FlexPRET that introduce non-negligible real-time specific hardware components to support WCET analysis. Conversely, LEOPARD focuses on MBTA requirements and aims at introducing minimum changes on the baseline architecture.

Other recent approaches have focused on understanding the analysability properties of complex COTS multicore processors. A summary of some of these works can be found in [15] and [37]. Some works suggest a separate analysis approach [41, 12]: they propose a separate analysis for contention and, frequently, rely on splitting tasks into sub-tasks or phases so that worst-case alignment in (typically) TDMA-based arbiters can be reasonably computed. In [24] authors derive a model to handle multicore contention based on resource stressing kernels and performance monitoring counters for deriving both, the access latencies to resources and the impact that tasks can suffer in the access to hardware shared resources. For static timing analysis, the work in [34] proposes an approach to factor in contention in single-core execution times by considering the interference (number of parallel contender requests) the requests of the task under analysis need to be arbitrated against, instead of considering the worst latency. Also, an enforcing mechanism (safety net) is put in place to ensure that tasks do not try to use more than its assigned budget in terms of request count.

Another software approach for time predictability is the $WCET(m)$ [29]. It creates, via OS support, resource partitions: private cache partitions for the last level cache; memory bandwidth partitioning by controlling and enforcing a maximum number of access counts considering worst-case memory latency L_{max}; and bank partitioning to further reduce memory latency. As a result, each core receives a resource partition and the WCET of each task depends on the number of active cores, m. Despite each task receives a hardware partition of resources, those resources can be jittery such as multi-level caches and FPU, and handling this intra-partition jitter complicates deriving WCET estimates as summarized in Section 2. LEOPARD hardware designs simplify providing evidence that measurements taken for a given task (under a resource partition) are representative and capture the impact of the jittery resources. LEOPARD support for multicore, e.g. the on-chip bus not covered in [29], contention has been shown to improve the baseline ones and further is compatible with the WCET(m) approach.

Overall, a commonality in these software models is that they assume that execution times are representative. For example, these works consider the task under analysis suffers no jitter either in time or access counts due to memory mapping (cache layout), or the particular values operated in the floating-point unit. However, as we have shown in this paper, both factors can create jitter. This assumption of representative measurements also holds in the typical worst-case analysis approach [39]. LEOPARD, by deploying deterministic and probabilistic jitter bounding, helps all these software approaches to increase the representativeness on the observations without requiring additional testing efforts.

7 Conclusions

We have shown the design of hardware support for easing MBPTA and its implementation resulting in an-RTL based prototype. While hardware concepts are simple and their implementation has low or moderate complexity in a simulator, their real integration with other hardware elements such as virtual/physical address management and cache flushing has been challenging. Moreover, during their integration and test we have discovered performance limitations that have been addressed proposing, implementing and integrating more efficient designs. Also, tracing capabilities needed by the timing analysis tools were not powerful enough in the baseline FPGA prototype. The net results is that we have successfully implemented all features required to make the FPGA prototype be MBPTA compliant, i.e. controlling on-core and on-chip sources of jitter. We have further designed and integrated a new high-speed tracing feature able to dump traces through the Ethernet port at high speed and without affecting normal execution of programs. Overall, LEOPARD benefits hardware designers to better understand how MBTA compliance can be achieved with low overhead and no impact on average performance while allowing to reach a growing market with time predictability needs. We also expect LEOPARD to motivate embedded system practitioners to push for those small changes to be implemented in other designs, increasing the effectiveness of existing software approaches to control jitter.

References

1 Jaume Abella, Damien Hardy, Isabelle Puaut, Eduardo Quiñones, and Francisco J. Cazorla. On the comparison of deterministic and probabilistic WCET estimation techniques. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, 2014. doi:10.1109/ECRTS.2014.16.

2 Jaume Abella, Carles Hernández, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems, SIES 2015, Siegen, Germany, June 8-10, 2015*, 2015. doi:10.1109/SIES.2015.7185039.

3 Irune Agirre, Mikel Azkarate-askasua, Carles Hernández, Jaume Abella, Jon Perez, Tullio Vardanega, and Francisco J. Cazorla. IEC-61508 SIL 3 compliant pseudo-random number generators for probabilistic timing analysis. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, 2015. doi:10.1109/DSD.2015.26.

4 Benny Akesson, Andreas Hansson, and Kees Goossens. Composable resource sharing based on latency-rate servers. In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2009, 27-29 August 2009, Patras, Greece*, 2009. doi:10.1109/DSD.2009.167.

5 Benny Akesson, Liesbeth Steffens, and Kees Goossens. Efficient service allocation in hardware using credit-controlled static-priority arbitration. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009, Beijing, China, 24-26 August 2009*, 2009. doi:10.1109/RTCSA.2009.13.

6 Peter Alfke. *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators.* Xilinx, 1996.

7 ARM. Amba bus specification. URL: http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

8 Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery D. Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. PROARTIS: probabilistically analyzable real-time systems. *ACM Trans. Embedded Comput. Syst.*, 12(2s), 2013. doi:10.1145/2465787.2465796.

9 Certification Authorities Software Team. *CAST-32A Multi-core Processors*, 2016.

10 Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor – GR740-UM-DS-D1 – Data Sheet and Users Manual, 2015.*

11 Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, 2012. `doi:10.1109/ECRTS.2012.31`.

12 Dakshina Dasari, Vincent Nélis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, 52(3), 2016. `doi:10.1007/s11241-015-9229-9`.

13 Enrique Díaz, Jaume Abella, Enrico Mezzetti, Irune Agirre, Mikel Azkarate-Askasua, Tullio Vardanega, and Francisco J. Cazorla. Mitigating software-instrumentation cache effects in measurement-based timing analysis. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, 2016. `doi:10.4230/OASIcs.WCET.2016.1`.

14 Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous non-intrusive hybrid WCET estimation using waypoint graphs. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, 2016. `doi:10.4230/OASIcs.WCET.2016.4`.

15 Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Christine Rochange, Tullio Vardanega, and Francisco J. Cazorla. Contention in multicore hardware shared resources: Understanding of the state of the art. In *14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, 2014. `doi:10.4230/OASIcs.WCET.2014.31`.

16 E. Francis. Autonomous cars: no longer just science fiction. *Automotive Industries*, 2014.

17 Cobham Gaisler. *Leon3 Processor*. `http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53`.

18 Sylvain Girbal, Miquel Moretó, Arnaud Grasset, Jaume Abella, Eduardo Quiñones, Francisco J. Cazorla, and Sami Yehia. On the convergence of mainstream and mission-critical markets. In *The 50th Annual Design Automation Conference 2013, DAC'13, Austin, TX, USA, May 29 – June 07, 2013*, 2013. `doi:10.1145/2463209.2488962`.

19 Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Design Autom. Electr. Syst.*, 14(1), 2009. `doi:10.1145/1455229.1455231`.

20 Carles Hernandez, Jaume Abella, Francisco J. Cazorla, Jan Andersson, and Andrea Gianarro. Towards making a LEON3 multicore compatible with probabilistic timing analysis. In *Proceedings of the 20th Data Systems In Aerospace Conference, DASIA, 2015, Barcelona, Spain*, 2015.

21 Carles Hernández, Jaume Abella, Andrea Gianarro, Jan Andersson, and Francisco J. Cazorla. Random modulo: a new processor cache design for real-time critical systems. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, 2016. `doi:10.1145/2897937.2898076`.

22 HiPEAC. hiPEAC vision, 2017. `https://www.hipeac.net/publications/vision/`.

23 Infineon. AURIX – TriCore datasheet. highly integrated and performance optimized 32-bit microcontrollers for automotive and industrial applications, 2012.

24 Javier Jalle, Mikel Fernandez, Jaume Abella, Jan Andersson, Mathieu Patte, Luca Fossati, Marco Zulianello, and Francisco Cazorla. Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP). In *8th ERTS*, 2016.

25 Javier Jalle, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. Bus designs for time-probabilistic multicore processors. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, 2014. `doi:10.7873/DATE.2014.063`.

26 Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. A cache design for probabilistically analysable real-time systems. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, 2013. doi:10.7873/DATE.2013.116.

27 Samuel Kotz and Saralees Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.

28 Stephen Law and Iain Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, 2016. doi:10.1109/ECRTS.2016.21.

29 Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. Wcet(m) estimation in multi-core systems using single core equivalence. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*, 2015. doi:10.1109/ECRTS.2015.23.

30 Enrico Mezzetti and Tullio Vardanega. A rapid cache-aware procedure positioning optimization to favor incremental development. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, 2013. doi:10.1109/RTAS.2013.6531084.

31 Suzana Milutinovic, Jaume Abella, and Francisco J. Cazorla. Modelling probabilistic cache representativeness in the presence of arbitrary access patterns. In *19th IEEE International Symposium on Real-Time Distributed Computing, ISORC 2016, York, United Kingdom, May 17-20, 2016*, 2016. doi:10.1109/ISORC.2016.28.

32 Kevin Mueller, Georg Sigl, Benoit Triquet, and Michael Paulitsch. On MILS I/O sharing targeting avionic systems. In *2014 Tenth European Dependable Computing Conference, Newcastle, United Kingdom, May 13-16, 2014*, 2014. doi:10.1109/EDCC.2014.35.

33 Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, 2014. doi:10.1109/ECRTS.2014.20.

34 Jan Nowotsch, Michael Paulitsch, Arne Henrichsen, Werner Pongratz, and Andreas Schacht. Monitoring and WCET analysis in COTS multi-core-soc-based mixed-criticality systems. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, 2014. doi:10.7873/DATE.2014.080.

35 Marco Paolieri, Jörg Mische, Stefan Metzlaff, Mike Gerdes, Eduardo Quiñones, Sascha Uhrig, Theo Ungerer, and Francisco J. Cazorla. A hard real-time capable multi-core SMT processor. *ACM Trans. Embedded Comput. Syst.*, 12(3):79:1–79:26, 2013. doi:10.1145/2442116.2442129.

36 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, 2009. doi:10.1145/1555754.1555764.

37 Michael Paulitsch, Oscar Medina Duarte, Hassen Karray, Kevin Mueller, Daniel Münch, and Jan Nowotsch. Mixed-criticality embedded systems – A balance ensuring partitioning and performance. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, 2015. doi:10.1109/DSD.2015.100.

38 PROXIMA. Probabilistic real-time control of mixed-criticality multicore and manycore systems, oct 2014. URL: http://www.proxima-project.eu/.

39 Sophie Quinton, Torsten T. Bone, Julien Hennig, Moritz Neukirchner, Mircea Negrean, and Rolf Ernst. Typical worst case response-time analysis and its use in automotive network design. In *The 51st Annual Design Automation Conference 2014, DAC'14, San Francisco, CA, USA, June 1-5, 2014*, 2014. doi:10.1145/2593069.2602977.

40 Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. Special publication 800-22rev1a, US National Institute of Standards and Technology (NIST), 2010.

41 Simon Schliecker, Mircea Negrean, Gabriela Nicolescu, Pierre G. Paulin, and Rolf Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2008, Atlanta, GA, USA, October 19-24, 2008*, 2008. `doi:10.1145/1450135.1450172`.

42 Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, and Christian W. Probst. Towards a time-predictable dual-issue microprocessor: The patmos approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop PPES 2011, March 18, 2011, Grenoble, France.*, 2011. `doi:10.4230/OASIcs.PPES.2011.11`.

43 Mladen Slijepcevic, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Design and implementation of a fair credit-based bandwidth sharing scheme for buses. In *DATE conference*, 2017.

44 Sysgo. *PikeOS Safe and Secure Virtualization*, 2010.

45 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, 2016. `doi:10.1109/RTAS.2016.7461361`.

46 Augusto Vega, Chung-Ching Lin, Karthik Swaminathan, Alper Buyuktosunoglu, Sharathchandra Pankanti, and Pradip Bose. Resilient, uav-embedded real-time computing. In *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015*, 2015. `doi:10.1109/ICCD.2015.7357189`.

47 Franck Wartel, Leonidas Kosmidis, Code Lo, Benoit Triquet, Eduardo Quiñones, Jaume Abella, Adriana Gogonel, Andrea Baldovin, Enrico Mezzetti, Liliana Cucu, Tullio Vardanega, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013, Porto, Portugal, June 19-21, 2013*, 2013. `doi:10.1109/SIES.2013.6601497`.

48 Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter P. Puschner. Measurement-based timing analysis. In *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, 2008. `doi:10.1007/978-3-540-88479-8_30`.

49 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008. `doi:10.1145/1347375.1347389`.

50 Marco Ziccardi, Enrico Mezzetti, Tullio Vardanega, Jaume Abella, and Francisco Javier Cazorla. EPC: extended path coverage for measurement-based probabilistic timing analysis. In *2015 IEEE Real-Time Systems Symposium, RTSS 2015, San Antonio, Texas, USA, December 1-4, 2015*, 2015. `doi:10.1109/RTSS.2015.39`.

51 Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. Flexpret: A processor platform for mixed-criticality systems. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, 2014. `doi:10.1109/RTAS.2014.6925994`.