Studying co-running avionic real-time applications on multi-core COTS architectures

Jingyi Bin, Sylvain Girbal, Daniel Gracia Pérez, Arnaud Grasset, Alain Mérigot

► To cite this version:

Jingyi Bin, Sylvain Girbal, Daniel Gracia Pérez, Arnaud Grasset, Alain Mérigot. Studying co-running avionic real-time applications on multi-core COTS architectures. Embedded Real Time Software and Systems (ERTS2014), Feb 2014, Toulouse, France. hal-02271379

HAL Id: hal-02271379
https://hal.science/hal-02271379

Submitted on 26 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Studying co-running avionic real-time applications on multi-core COTS architectures

Jingyi Bin*†, Sylvain Girbal†, Daniel Gracia Pérez†, Arnaud Grasset† and Alain Merigot*
*Fundamental Electronic Institute, France
†Thales Research & Technology, France

***Abstract*—For the last decades, industries from the safety-critical domain have been using Commercial Off-The-Shelf (COTS) architectures despite their inherent runtime variability. To guarantee hard real-time constraints in such systems, designers massively relied on resource over-provisioning and disabling the features responsible for runtime variability.**

The recent shift to multi-core architectures in the embedded COTS market worsened the runtime variability problem as contention on shared hardware resources brought new variability sources. Additionally, hiding this variability in additional safety margins as performed in the past will offset most if not all the multi-core performance gains.

To enable the use of multi-cores in this domain, it has become essential to finely characterize at system level the application workload, as well as the possible contention on shared hardware resources.

In this paper, we introduce measurement techniques based on a set of dedicated stressing benchmarks and architecture hardware monitors to characterize (1) the architecture, by identifying the shared hardware resources and their associated contention mechanisms. (2) the application, by identifying which shared hardware resources it is sensitive to. Such information would guide us toward identifying which applications can run smoothly together without endangering individual worst-case execution times.

I. Introduction

Industries from the safety-critical domain such as the avionic, automotive, space, healthcare or robotic industry, have been using Commercial Off-The-Shelf (COTS) architectures rather than in-house solutions to reduce both the non-recurring engineering costs (NRE) and the time-to-market (TTM) [3], while fitting their exponential needs in performance and functionalities [2], [6], [5], as illustrated by Figure 1.

In the avionic industry, the real-time software embedded in aircrafts is characterized by a few hundreds of small and independent applications running together with a few larger and very communicating applications. Embedded COTS multi-core platforms are an expected trend for next generation aircrafts, providing a low-cost opportunity to run concurrently these small independent applications.

However, as COTS providers are mainly targeting the consumer electronic market, mostly driven by best-effort performances, the safety-critical industry has to face more and more runtime variability issues [17], [22].

Concurrently, the safety-critical software is characterized by stringent hard real-time constraints and missing a single deadline may have some catastrophic consequence on the user

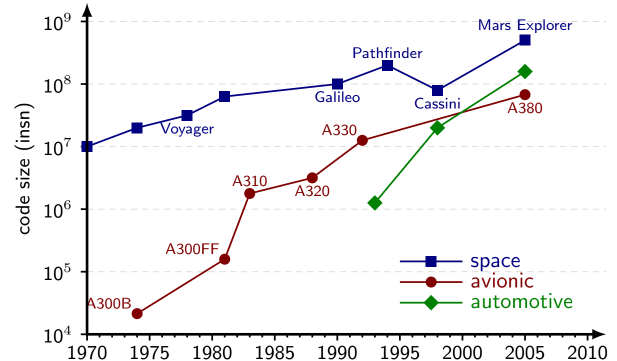


Fig. 1. Evolution of code size in space, avionic and automotive embedded systems

or the environment. Therefore, time predictability is a major concern, and the safety-critical industry is heavily relying on resource over-provisioning to mitigate such an unacceptable deadline-miss risk.

A common practice to guarantee the deadlines of a safety-critical application with single-core architecture is to determine the application Worst-Case Execution Time (WCET). This WCET computation usually relies on analysis tools based on static program analysis tools [25], [18], detailed hardware model, as well as measurement techniques through execution or simulation [11]. However, these analysis techniques and tools are not currently able to provide an exact computation of the WCET, only delivering an estimated upper bound, introducing some safety margins as depicted in Figure 2.

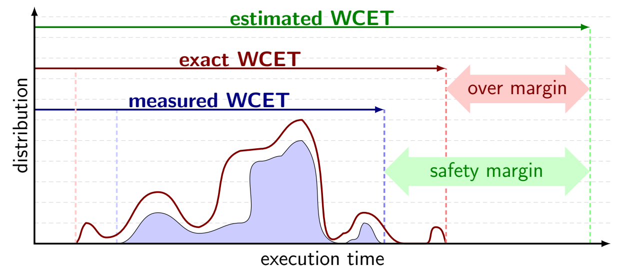


Fig. 2. Estimation of the Worst-Case Execution Time, and the over-estimation problem

A direct extension to the measurement-based analysis to multi-core COTS processors would involve several co-running applications. However this shift worsened the runtime variability problem as contention on shared hardware resources bring new variability sources even in the case of independent

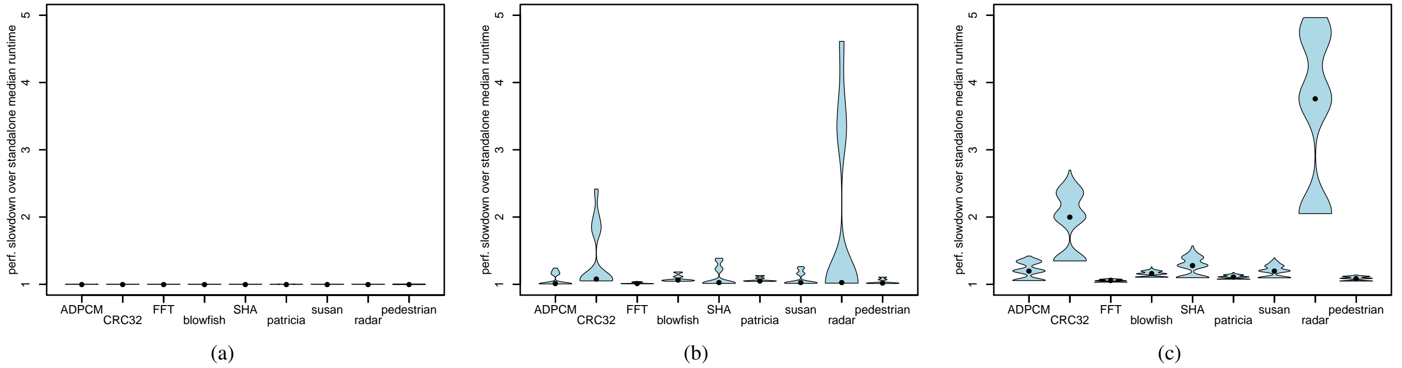


Fig. 3. Runtime variability over 600 iterations of reference applications running (a) standalone, (b) concurrently with 2 benchmarks stressing the shared memory path, and (c) concurrently with 7 benchmarks stressing this resource.

co-running applications.

Despite all the improvements in the WCET estimation domain [14], [7] over the last decades, the over-estimation remained mostly constant as the predictability of the architecture decreased [25], thus making the use of WCET analysis tools difficult for real industrial programs running on multi-core COTS architectures [13], [15]. Possible interference on shared hardware resources among co-running tasks significantly increases the complexity of timing analysis, forcing it to have a full knowledge of co-running tasks at software level, and detailed resource contention models at hardware level.

In Figures 3(a, b, c), we show that hiding the worst-case variability with additional safety margins as it was performed in the past is not an option for multi-core COTS. Figure 3(a) illustrates, with distribution violin plots [12], the runtime variation for some applications running standalone on 8-core barebone platform. To mimic a single-core configuration, the other cores are idle. In such a configuration the runtime-variation remains very low (below 1%).

However, when introducing co-running benchmarks, the runtime variation of each application around the previously computed standalone median is increasing rapidly as depicted in Figure 3(b) showing runtime variation in presence of 2 co-running benchmarks stressing the shared memory path. The impact on the average runtime is not significant, however, for the worst-case, we observe an average variation of 71% and a maximum variation of 361%.

Increasing the number of co-runners as depicted in Figure 3(c) furthermore degrades the runtime variation, up to an average variation of 87% on the worst case and a maximal variation of 396%. Additionally, the average and minimal execution times start to be impacted as well, with a variation of 54% on the average runtime.

Another study [16] from EADS also exhibits that using actual WCET analysis techniques for multi-cores would force the industry to multiply the WCET by a value close to the number of cores being used, providing no performance benefits over single-cores.

As considering the impact on such a large runtime variation would lead to unsustainable WCET margins far above the performance benefits of multi-core systems, it is critical to control this variation by providing a detailed characterization on how each co-running application is behaving relatively to the shared hardware resources.

In this paper, we present measurement techniques allowing us to (1) characterize the underlying architecture, identifying the shared hardware resources responsible of most of the runtime variation, (2) identify which shared hardware resource each application is sensitive to, and (3) predict which applications could run smoothly together without endangering individual worst-case execution times.

II. Measurement techniques

In this section we present two different measurement metrics and tools: *Hardware Monitors* and *Stressing Benchmarks*. These metrics allowed us to study and characterize the behavior of co-running independent applications, as well as the behavior of the contention mechanisms of the multi-core architecture.

A. Hardware monitors

Most recent architectures include some special on-chip hardware, the hardware Performance Monitoring Counters (PMC), that provides to the system accurate information of hardware events. Once collected from special purpose registers counting the occurrence of architectural events, this information provides performance information [21] on the applications, the operating system, and the underlying hardware behavior. Therefore, it is usually used to guide the programmer into better tuning the applications to maximize their performance through various optimizations [20], [1].

This monitoring information is not only useful for performance tuning, but also to characterize workloads, allowing us to quantify hardware resource utilization at application level and providing some clues to better understand runtime variability [4]. In a multi-core context, hardware monitors are an opportunity to observe contention phenomenons at the level of the shared hardware resources.

The real source of the variation could however remain hidden. For example, in current processors we can monitor the number of cache misses that an application has, but we

cannot distinguish between cache misses inherent to application behavior and cache misses caused by the interaction with other applications accessing the same shared cache [26].

The special purpose registers available from the micro-architecture instruction-set are confined to count on-die related events. In some architectures, it would imply not being able to gather some events related to the last cache level, as well as the interconnect, and the DDR controller. As contention on these resources can actually be the bottleneck of the architecture, we cannot afford not collecting this information.

However for recent embedded architectures, the integrators are proposing some monitoring facilities at platform level. These monitoring features are most of the time dedicated to debugging through proprietary hardware probes but allow to count events at all the platform levels, including the number of DRAM refresh, page switch, and so on. This ability is a proof of the existence of some additional (most of the time undocumented) platform-level hardware counters that could be exploited through some reverse-engineering.

B. Stressing benchmarks

To better characterize concurrent accesses to shared hardware resources, we defined a large set of stressing benchmarks, each dedicated at stressing a particular potentially shared hardware resource.

On one hand, solely running a set of stressing benchmarks on the target system allows us to characterize the underlying architecture, by identifying which resources are effectively shared, as well as providing some information related to undisclosed features, such as resource-level contention mechanisms.

On the other hand, by selecting some of these stressing benchmark as co-runners for an application and studying the performance impact on the application, we are able to characterize how a black-box application behaves relatively to a particular hardware resource.

As an extension of the work proposed in [19], each resource stressing benchmark is dedicated at introducing a high-load onto one of the processor hardware resources. As a consequence, the resource-stressing benchmark is providing a good upper-bound estimate of the potential slowdown that a set of simultaneously-running applications may cause to the target application with respect to the studied resource.

Stressing benchmarks are directly written in assembly code to minimize the impact on the other resources the stressing benchmark do not aim at stressing. Even though stressing a single resource is not necessarily possible (like stressing the L3 cache without impacting the L1 and L2 caches), we tried to reduce the effect by minimizing the number of lines impacted in the L2 cache while stressing the L3.

We separated stressing benchmarks into different categories corresponding to the part of the architecture these benchmarks target: pipeline-level benchmarks (stressing the FPU unit, the load/store unit, the branch unit, ...), the i/o level (stressing pci-e, ethernet, or serial interfaces), the memory level (stressing on-die caches, platform caches, or the ddr-controller)

C. Representing runtime variability

In this paper, we are considering safety critical applications that requires to control their runtime variability to ensure that their worst execution time is below the hard real-time deadlines.

Runtime variability however, is not only characterized by a minimum and a maximum runtime: The statistical distribution of the runtimes also provides some useful information such as the rarity of the worst case, the distribution relatively to the median runtime, ...

To represent this distribution of runtimes, we relied to violin plots [12] such as the one presented in Figure 4.

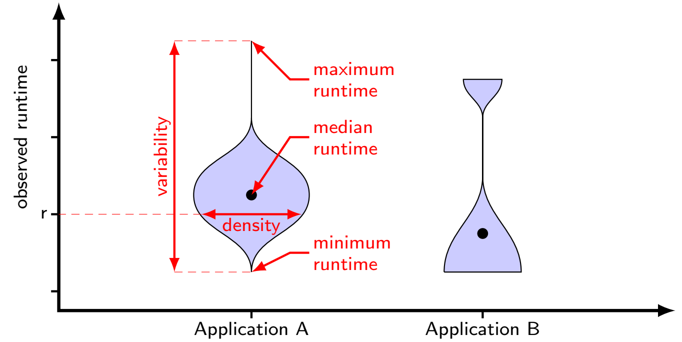


Fig. 4. Example of violin plot to represent runtime distribution of two different applications

Figure 4 is composed of two violin plots providing some information about the runtime distribution of two applications A and B. For each application the variability is characterized by the bottom-most point and the top-most point that correspond to the minimum and maximum observed running times. The black dot in the violin represents the median running time. Finally the width of the plot for a particular runtime (r in the figure) is proportional to the density of the runtime population with such a runtime.

III. Methodology

As previously described, safety-critical systems typically consist on a multitude of independent applications. To maintain / reduce NRE costs while improving usage / performance embedded COTS multi-cores are the targeted hardware solutions.

Especially, the avionic domain is relying a lot on sub-contracting and is facing an important challenge during the integration phase, dealing with the integration of both gray- and black-box components, while part of the hardware being undisclosed as using COTS architecture.

Our objective is therefore to characterize how each application is individually accessing each potential shared hardware resource, to anticipate at integration time, how these applications will impact each other when running concurrently on the multi-core.

During our study depicted by Figure 5, we started with characterizing the architecture to identify the shared hardware resources of the system. We then focused on characterizing the applications to identify which resources each application was accessing.

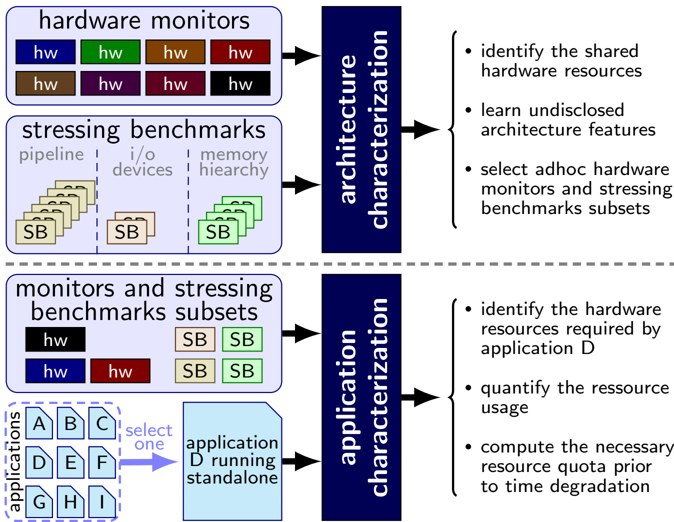


Fig. 5. Overview of the analysis process

A. *Architecture characterization*

The objective of architecture characterization is twofold: first to identify the effectively shared hardware resources of the target architecture, and second to discover undisclosed architecture features (such as the exact structure of the NOC) that can have a significant impact on runtime and contention.

To perform the architecture characterization analysis, we first run a particular resource stressing benchmark on the target multi-core processor in isolation to collect all the hardware monitor information. We then replicate this stressing benchmark up to the number of available cores to collect hardware monitor information again. Comparing these results allows us to quantify the contention mechanism of this stressed particular resource, while reproducing this study for every potentially shared resource enables us to identify each effectively shared resource as well as the maximum throughput available on each of these resources.

Also, varying the deployment pattern of stressing benchmarks on the architecture enables us to learn undisclosed architecture features, such as the structure of the interconnect, the cost of the coherency traffic, the number of shared ports on a shared cache and so on...

Finally, running all the different stressing benchmarks together may allow us to identify correlated resources and monitors, as well as constant monitored values. Such information is useful to discover useless monitors and stressing benchmarks. For example intituively in a multi-core architecture with distributed L1 and L2 caches, the monitors counting L1 hits and misses are useless when studying the impact of an application running on a core on another application running on a different core, and so are the stressing benchmarks stressing the L1 cache, as co-running application will never cause a contention on this resource; on the other side monitors counting the number of requests send by a core to the interconnect might cause contention between applications. At the end of this phase, a subset of hardware monitors and stressing benchmarks appropriate to the target architecture are identified.

If the target architecture supports different configuration modes (such as with some caches enabled or disabled, with optional contention, with some cache partitioning, etc...), then the whole architecture characterization phase could be replicated for every configuration to identify the most suitable configuration for a particular application domain.

B. *Application characterization*

The purpose of the application characterization analysis is to understand why a particular application performance, and thus its execution time, varies when the application is running with other applications in the same multi-core. As most these variation are due to conflict on shared hardware resources access, it is critical to identify which of such resources each application requires, as well as to quantify the usage of the resource by the application.

To perform the application characterization analysis, we run each application with every stressing benchmark from the subset identified by the architecture analysis, gathering hardware counters information related with shared hardware resources. Application slowdowns allow us to identify to which resources this application is sensitive as well as a quantification of the sensitiveness thanks to the observed performance variability.

Also, it enables us to evaluate the minimal share of the resource the application needs to be provided with, prior to facing significant performance degradation.

Finally, repeating the process on every application would allow us to identify applications likely to run concurrently on the system without significantly degrading individual worst-case execution times.

IV. Experimental Setup

This section first presents both the hardware and the software configurations we considered for our study; and second quantifies the associated design space size. The associated results will be presented in Sections V and VI.

A. *Hardware configuration*

The experiments presented in this paper are performed with the 8-core Freescale P4080 Development System [9]. This system, depicted in Figure 6 and Table I, is composed of eight e600mc PowerPC cores coupled with private L1 data+instruction caches and a private L2 unified cache. The eight cores communicate with two shared off-chip L3 platform caches through the proprietary CoreNet interconnect. Each L3 cache is itself connected to a dedicated DDR controller.

Core Pipeline Distributed L1 caches Distributed L2 cache	8 Power Architecture e500mc at 1.5GHz 7-stage pipeline, superscalar, out-of-order 32kb, 8-way associative, 64-byte line, PLRU 128kb, 8-way associative, 64-byte line, PLRU
Performance monitors	162 performance monitor counters 4 special registers per core
Shared L3 cache (x2)	1MB, 32-way associative, 64-byte line, PLRU
Memory controller	Two DDR memory controllers
Interconnect	CoreNet Coherency Fabric

TABLE I. Freescale P4080 specifications

Hardware monitors. Each e600mc PowerPC core provides 162 different performance monitors, and has the ability to

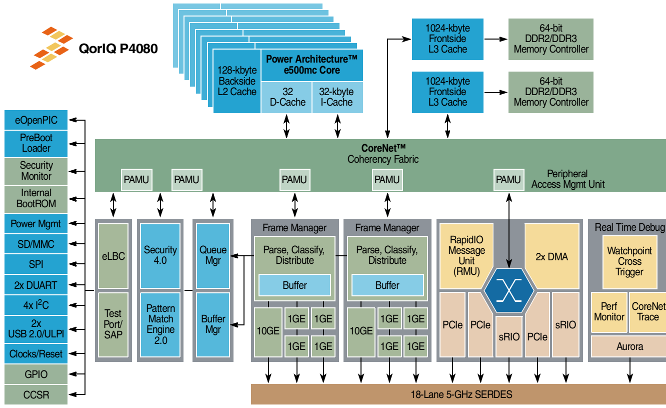


Fig. 6. Freescale P4080 block diagram

collect 4 of them at a given time thanks to 4 performance monitor registers.

Besides core-related performance monitors, the P4080 platform provides us with a set of poorly documented memory-mapped registers allowing us to monitor interconnect activity, L3 cache and DDR accesses.

Architecture tuning. Most multi-core COTS provide some amount of configurability to adapt users' requirements. For example, caches help to reduce data transfer time between cores and memory to improve overall system performance, but they also bring performance variability when they are shared among co-running applications. To improve predictability, most embedded architectures allow to simply disable caches, some provide some degree of hardware partitioning, and few allow the caches to be configured as SRAM memories.

Our P4080 platform provides us with two shared L3 caches each connected to a dedicated DDR controller. Beyond allowing us to enable one or both of the L3 cache / DDR controller pairs, the P4080 L3 cache offers some hardware partitioning support, allowing us to shift from a cache fully shared by the cores to a cache where each core has a dedicated pre-allocated memory space. While the fully shared cache will in principle provide better overall performance by better fitting asymmetric load balancing scenarii, the pre-allocated cache will provide better predictability by preventing evictions due to other cores.

In the context of safety-critical systems, the optimal configuration is characterized by low performance variability and sufficient average performance. However, it is not obvious to intuitively infer the most suitable configuration. For instance, partitioning may degrades performance below acceptable throughput, and activating the second L3 cache / DDR controller only makes sense if it does not compromise the predictability of the interconnect.

We have selected four configuration candidates described in Table II to be challenged for performance and predictability. The experiments realized to determine the optimal configuration for our safety-critical context is described in Section V-B.

configuration	level 3 cache		#ddr
	size	associativity	controllers
single controller non-partitioned	1MB, shared by 8 cores	32-way	1
single controller partitioned	128KB, per core	4-way	1
dual controller non-partitioned	2×(1MB, shared by 4 cores)	32-way	2
dual controller partitioned	256KB per core	8-way	2

TABLE II. FOUR CONFIGURATIONS OF P4080

B. Software configuration

As a proxy for various independent applications co-running on a safety critical system, we used a subset of the MiBench benchmark suite [10], a set of embedded benchmarks from various domains of the embedded market: automotive, consumer, office, networking, security and telecommunication. We selected a subset of 7 mibench benchmarks: ADPCM, CRC32, FFT, blowfish, SHA, patricia, and susan to be ported for barebone testing on the P4080 hardware platform, eliminating operating system requirements such as system calls and dynamic memory management.

We completed this suite of small benchmarks with two larger industrial-level applications developed internally at Thales including additional hard real-time constraints: 1) an airborne radar application based on the Space-Time Adaptive Processing (STAP) algorithm [24] to detect targets in the presence of both clutter and jamming. 2) a pedestrian detection application based on the Viola & Jones shape recognition algorithm [23] to detect pedestrian on security camera footage.

All the applications presented in this paper were run barebone on the platform, without any operating system to minimize variability. We eliminated preemption which has to be strictly controlled for safety-critical systems, by having each core running a unique benchmark.

Applications were loaded using a hardware debug probe managed through the CodeWarriork [8] software provided by Freescale.

C. Design space

Considering the available number of hardware monitors (~ 200), the total number of stressing benchmarks (~ 1000), and the total number of possible mappings on an 8-core architecture the the experimental space of the methodology presented in Section III can be quite large.

Let A be the number of applications to characterize, S the number of stressing benchmarks, M the number of available hardware monitors, C the number of cores, I the number of iterations of repeating each experimental scenario, and N the number of monitor each core is able to measure at once. The total number of possible experiments is:

$$\frac{1}{N}(AMC(S+1)^{(C-1)} + SMC(S+1)^{(C-1)})I$$

Considering that average execution time of a single experiment to be 100ms, and the order of magnitude for the values presented in Table III, it would requires us 10^{19} years to exhaust such a design space.

Applications	A	9
Stressing benchmarks	S	1000
Hardware monitors	M	200
Cores	C	8
Iterations	I	100
Performance monitor registers	N	4

TABLE III. ORDER OF MAGNITUDE OF THE DESIGN SPACE

To deal with such a design space we setup an automatic framework. As depicted in Figure 5 we perform an overall architecture characterization prior to performing the application characterization.

Beyond the characterization aspect presented in Section V, the architecture characterization phase also allows us to filter out unnecessary hardware monitors and stressing benchmarks by identifying those that are not related to performance variability. For instance, hardware monitors and stressing benchmarks related to the L1 data cache were filtered out by our framework, as the L1 data cache is not a shared hardware resource and therefore not responsible for performance variability in a co-running context.

Beyond this first step towards the reduction of the design space size, the Section V and the Section VI will show how more detailed characterization of both the architectures and the applications helps to furthermore reduce this space.

V. ARCHITECTURE CHARACTERIZATION RESULTS

This section regroups all the architecture characterization related results. We present first how our methodology allowed us to identify undocumented architecture features, and second the experiments that enabled us to select the most suitable hardware configuration for safety-critical real-time applications. Finally, we detail the experiments that allowed us to quantify each hardware resource availability.

A. Identifying undocumented architecture features

Embedded COTS architectures come with detailed ISA and block diagram, but many aspects of the micro-architecture remain undisclosed such as the exact SoC network topology, contention, arbitration and prefetcher mechanisms. If such information is not necessary to guarantee correct functional behavior, it could have a significant impact on the timing behavior, that is as much important for safety-critical real-time systems.

The architecture characterization phase is an opportunity to learn about these undisclosed features and mechanisms. More particularly, our experimental P4080 platform features a complex CoreNet interconnect to connect all the cores to two L3 platform caches, each connected to a DDR controller. The topology of this interconnect has a significant impact on how the memory traffic of one core will interfere with the traffic of other cores.

The architecture characterization allows us to quantify the competition on the CoreNet interconnect resource, as well as the impact of both the interconnect and the mapping on the application runtime variability.

To characterize the interconnect, we used the default platform configuration appearing as first row of Table II. By running and monitoring various stressing benchmarks within this configuration, we managed to figure out that the eight cores are organized as two clusters of four cores.

The experiment illustrating this cluster effect is depicted in Figure 7. It shows the runtime variability of running three instances of a particular stressing benchmark of the 8-core architectures, mapping and monitoring the first instance on core #1, while varying the mapping of the other two instances on the other available cores.

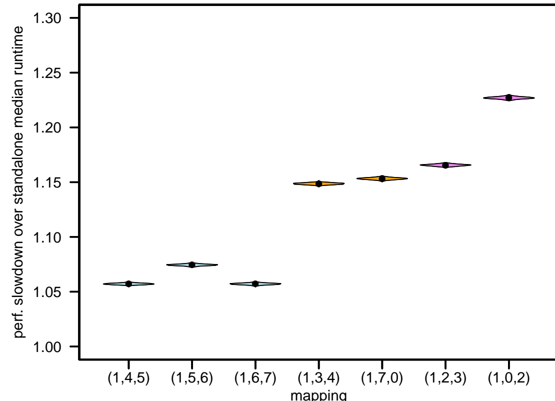


Fig. 7. Runtime variability while mapping three instances of a stressing benchmark on different cores.

Three different distributions can be identified in the figure: The first one, corresponding to the first three violin plots, corresponds to mapping the monitored instance alone while running the two other instances on the other cluster. It exhibits only a small performance degradation with a maximum of $+7.6\%$ and relatively small variance between the three mappings. The second distribution, illustrated by the next two violin plots, corresponds to mapping the monitored benchmark on the same cluster as one of the two other instances and shows a bigger performance degradation with a maximum of $+15.6\%$ and low variability. Finally, the third distribution, illustrated by the last two violin plots corresponds to running all three instances on the same cluster and exhibits the largest performance degradation and an important variability (from $+16.3\%$ to $+23.0\%$).

As a conclusion, due to the cluster effect, performance variability of applications is placement dependent for the P4080 platform. However within each 4-core cluster the performance does not depend on the placement, enabling us to reduce the number of mapping to be tested, and therefore allowing us to reduce the overall design space.

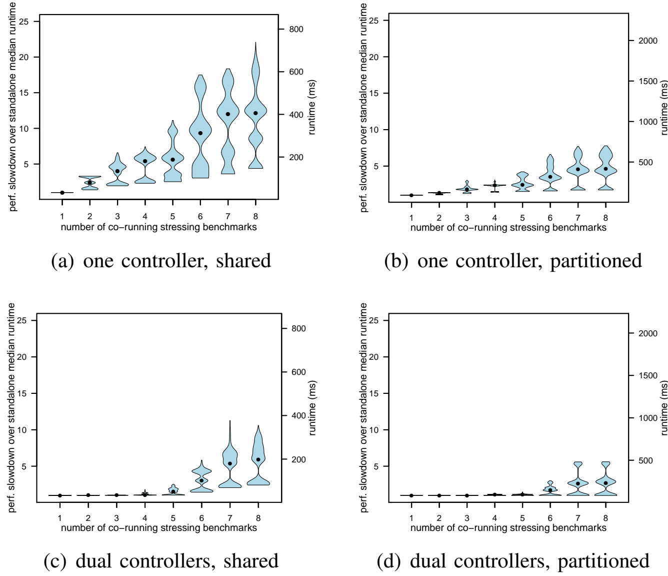
B. Identifying the optimal hardware configuration

In Section IV-A we identified several hardware configurations presented in Table II. Selecting the most appropriate configuration for safety-critical applications is not straightforward as two criteria have to be maximized: 1) predictability, ensuring low performance variability, and 2) sufficient minimal performances assuring the worst case execution time will be below the application deadlines of the hard real-time system.

To evaluate these different hardware configurations, we designed a set of stressing benchmarks dedicated at stressing

the different shared hardware resources along the memory path including the CoreNet interconnect, the L3 caches and the DDR controllers.

While monitoring the stressing benchmark running on core #1, we varied the number of stressing benchmarks running on the remaining cores. Figure 8 shows the distribution of the observed runtime of core #1 while varying the number of co-runners, during 200 iterations of these experiments.



(a) one controller, shared (b) one controller, partitioned

(c) dual controllers, shared (d) dual controllers, partitioned

Fig. 8. Runtime variability of one of the stressing benchmarks while varying the number of co-running instances.

The y-axis on the left shows the observed speed down compared to when running the stressing benchmark standalone for each particular configuration. The y-axis on the right correspond to the overall runtime.

Performance variability of the different configurations can be obtained by comparing the height of the various distribution violin plots relatively to the left y-axis, tallest plots being the one with the largest variability. Worst performance of the various configurations can be observed with the top-most point of each violin plot. The associated runtime appears on the right y-axis.

The configurations exhibiting the lowest variability are the configurations with a partitioned L3 cache appearing in Figures 8(b) and (d). On the other hand, enabling a second L3 cache and associated DDR controller in configurations in Figures 8(c) and (d) also allow the system to reduce performance variability by providing some load balancing, while increasing the overall performances.

To put it simply, on one hand, activating the second L3 cache and associated controller brings both more predictability and more performance. On the other hand, shared L3 caches are providing more performance while partitioned L3 caches are offering more predictability.

To compare these two last configurations, we collected worst execution times of Figures 8(c) and (d) into Table IV. Even by offering a larger variability, the shared configuration provide better overall performance leading to lower worst

	1	2	3	4	5	6	7	8
dual-controller shared	34	34	35	61	84	196	378	355
dual-controller partitioned	86	86	87	102	126	259	481	482

TABLE IV. WORST EXECUTION TIMES (IN MS) FOR THE MONITORED CORE WHILE VARING THE NUMBER OF RUNNING BENCHMARKS.

case runtimes than the partitioned configuration. Therefore the worst case upper bound has the opportunity to be lower for the shared configuration, making the dual-controller shared configuration the most efficient setup for our safety critical system.

Even though we shown that the configuration with two controllers and shared L3 caches could be the most efficient for safety critical systems as allowing a lower upper bound for worst execution time, avionic applications usually favor partitioning as it allow to minimize interferences, here eliminating eviction due to other co-running tasks. For this reason we focused on the configuration with two controllers but with partitioned L3 caches for the remaining of the paper.

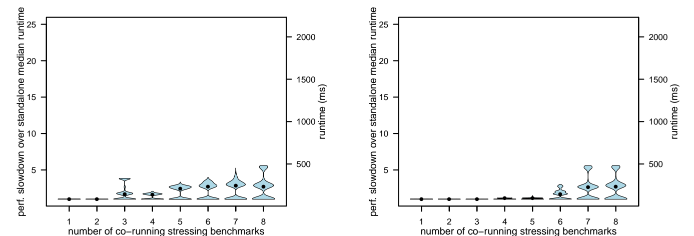
Using this configuration is also in cope with the cluster organization identified in Section V-A, and will benefit from the same design space reduction options as applications sharing the same cluster / L3 cache / memory controller will compete on the same shared hardware resources, whereas application placed in different clusters will not.

C. Selecting the adequate mapping

Selecting an optimal mapping with the considered dual-controller partitioned hardware setup is important as the amount of resource competition between tasks will be tied to their core placement, large competition for tasks running on the same cluster, and low to no competition for tasks running on different clusters.

This optimal mapping largely depends on the application to be considered. In a system only running critical blackbox tasks with the same level of criticality, a fair load balancing of the tasks between the cluster should be privileged.

In a mixed critical system running a few very high-critical applications together with some lower-critical ones, designers may want to keep one cluster for the high-critical tasks, and the other cluster for the low critical tasks. This way, the possible impact of low-critical tasks on high-critical ones is minimized by reducing the sources of resouce competition between them.



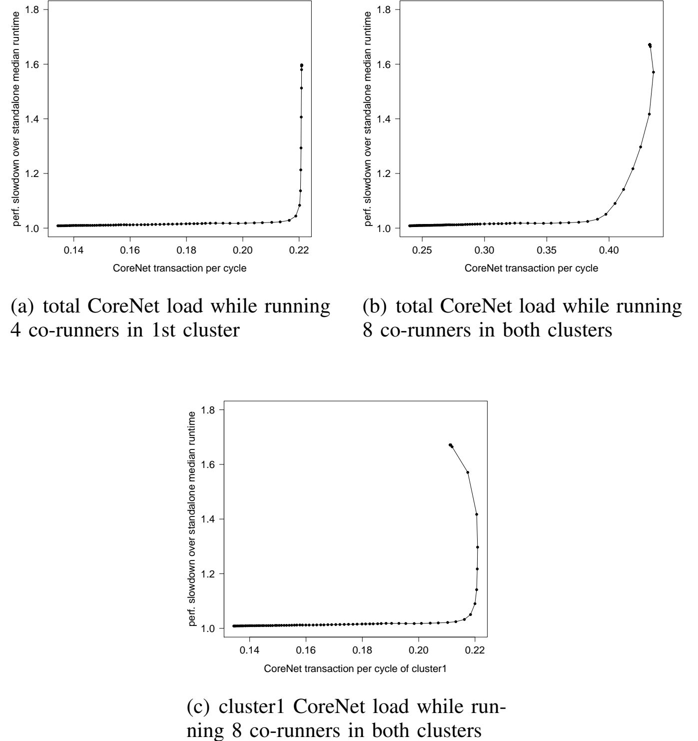
(a) fair load balancing between tasks with the same criticality level

(b) load balancing minimizing tasks running with the high-critical one

Fig. 9. Comparing the runtime variability of different balancing techniques.

Figure 9 presents these both setups. We placed the benchmark with highest criticality level in the first cluster, and then augmented the number of co-running benchmarks. In Figure 9(a) new benchmarks were placed to ensure fair load balancing, in Figure Figure 9(b) new benchmarks, considered of lower criticality were placed first on the other cluster.

While 9(a) corresponding to the fair load balancing exhibit similar and more regular variabilities, for 9(b) the performance variability starts to be impaired only when the number of co-runners do not fit anymore into the second cluster. As a consequence, mixed-critical system should better not try to use all the avialable ressources to ensure that the low-critical traffic does not impact the high-critical one.



(a) total CoreNet load while running 4 co-runners in 1st cluster

(b) total CoreNet load while running 8 co-runners in both clusters

(c) cluster1 CoreNet load while running 8 co-runners in both clusters

Fig. 10. Performance slowdown variability versus CoreNet load to identify CoreNet maximum bandwidth

D. Quantify resource availability

With a hardware setup selected, the last step of the hardware characterization is to identify the available amount of each hardware resource, quantifying maximum throughput, available number of concurrent accesses and so on.

The most important resource to be characterized (and the least documented) is the CoreNet interconnect connecting the core clusters to their dedicated L3 cache and DDR controller, as well as to the other off-chip resources.

Evaluation of the CoreNet interconnect. Very few details on the topology of the CoreNet interconnect are available, and the only information available in the P4080 reference manual is the 0.8 Tbps coherent read bandwidth.

To better characterize this interconnect, we need to figure out its maximum throughput corresponding to the amount of traffic needed to saturate the interconnect. Remaining strictly below this saturation value would allow us to minimize the variability due to the interconnect, while getting closer to the saturation value would mean a significant increase of the runtime.

We also need to figure out how this available bandwidth is distributed among the clusters. Is the total bandwidth shared by all the cores, or is this bandwidth partitioned per-cluster?

To collect this information, we set up a dedicated stressing benchmark performing misses in the L1 and L2 caches to maximize possible CoreNet usage, and tuned this stressing benchmark to be able to select the CoreNet stressing level (the number of CoreNet accesses per CPU cycle). The results of the related experiments are presented in Figure 10.

Figure 10(a) depicts the correlation between the performance variability and the CoreNet load while running four instances of our stressing benchmark on the first cluster. The figure clearly shows a brutal degradation of performance when reaching 0.219 CoreNet transactions per CPU cycle. This inflection point corresponds to when the interconnect becomes saturated just before reaching the maximum bandwidth.

Figure 10(b) describes the same correlation when running eight instances of the stressing benchmark. The performance degradation now occurs when the number of CoreNet transactions reach 0.406 transaction per CPU cycles. This nearly doubled value means that the full available CoreNet bandwidth is evenly distributed to both CPU clusters, and that the maximum available bandwidth for a single application is 0.221 transaction per CPU cycle.

Finally, in Figure 10(c) we ran again the experiments of Figure 10(b), while only monitoring the CoreNet accesses of the first cluster. These results have to be compared to the results of Figure 10(a), also monitoring a single cluster activity.

The final shape of the curve in Figure 10(c) confirms that the total bandwidth of the CoreNet network (0.436 transactions per cycle) is not sufficient to support the maximum bandwidth of both clusters (2×0.221 transactions per cycle).

In a real-time context, to enforce that the activity of a cluster does not impact the activity of the other one, we therefore need to make sure that this maximum bandwidth is not reached.

Evaluation of the DDR. Another important shared hardware resource to consider are the DDR controllers and the associated DDR memory.

To similarly evaluate the maximum throughput and saturation value of each DDR controller, we shifted back to the hardware setup with one unique L3 cache and associated DDR controller. This allowed us define a new stressing benchmark aiming at stressing the DDR controller with the load of 8 different running cores performing misses in the L3 cache.

Figure 11 shows the correlation between the performance variability of the benchmark monitored in core #1 and the number of accesses to the DDR controller. The figure exhibit

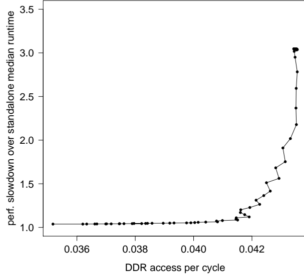


Fig. 11. Runtime variability versus DDR controller accesses to identify each DDR controller maximum bandwidth

again an inflection point when saturating the DDR controller when reaching 0.042 accesses to the controller per CPU cycle.

VI. Application Characterization Results

In the previous section we quantified the different available shared hardware resources. In this section, we will focus on identifying the resource requirements of the applications. The application runtime variability while stressing a particular resource will allow to determine the share of the hardware resource that each application requires.

We will start by figuring out the optimal number of iterations required to fully capture the runtime variability of each application, and then quantify the resource usage of each application. Such an information about resource usage could later be used to determine which applications could run smoothly together.

A. Optimal number of iterations to capture variability

To capture runtime variability of a particular application, each experiment involving this application has to be run several times in successive iterations. A large enough number of iterations will be able to capture the whole runtime variability of the application, while a not sufficiently large number will miss the runtime with the rarest distribution. As missing the worst execution time is not an option, we need to figure out what is this optimal number of iterations allowing to fully capture the runtime variability.

To empirically determine this optimal execution iteration number of a particular application, we setup an experiment performing successive executions of this concurrently with a resource-stressing environment. Every 100 iterations, we collected the runtime distribution since the beginning of the execution.

Figure 12 shows such runtime distribution results for the Adpcm benchmark. The shape of a violin plot corresponds to the captured behavior of the application. Therefore, comparing the violin plot shapes enables us to figure out if the optimal iteration value has been reached. For these experiments, stopped the iteration counter when we obtained three consecutive identical violin plots. For Figure 12 the optimal number of iteration is therefore 1000.

Identifying the optimal number of iterations to capture the runtime variability of each application allows us to reduce the

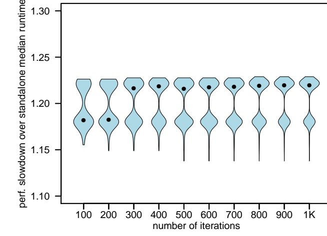


Fig. 12. Runtime variability collected with different number of iterations for application Adpcm.

overall design space. We applied the same methodology to identify the optimal number of iterations for every experiment. We found that the largest number of required iterations was 1000, and the average number was 200.

B. Capturing resource utilization

Section V allowed us to quantify the maximum number of resource available in the architecture. We now want to capture the resource requirements of each application from the software setup described in Section IV-B.

To perform this measurement, we ran each application standalone, collecting the same hardware monitors as the ones which allowed us to quantify the CoreNet and the DDR controller maximum load in Section V-D.

Table V provides the resource usage information for each application, this usage being computed as a ratio to the previously quantified saturation value: 0.219 requests per CPU cycle for the CoreNet interconnect, and 0.042 requests per CPU cycle for the DDR controller.

Application	average CoreNet	peak CoreNet	average DDR	peak DDR
ADPCM	0.86%	9.04%	4.52%	47.13%
CRC32	0.93%	1.30%	4.77%	6.90%
FFT	0.19%	13.38%	0.38%	31.43%
blowfish	0.14%	0.72%	0.74%	3.74%
SHA	0.25%	2.15%	1.33%	11.19%
patricia	0.07%	0.19%	0.35%	0.97%
susan	0.40%	2.96%	2.01%	15.44%
airborne radar	2.23%	3.06%	11.68%	16.19%
pedestrian detection	0.10%	4.29%	0.48%	22.86%

TABLE V. CoreNet and DDR loads of standalone application

Table V lists both the average and peak number of resource usage for the considered applications. The maximum average usage remains quite low: 2.23% of the available CoreNet resources, and 11.68% of the DDR resources. The peak usage however is significant with Adpcm using as much as 47% of the available DDR bandwidth.

C. Predicting co-running behavior

With the resource usage of each benchmark and the total amount of available resources quantified, we can use this information to predict the behavior of co-running applications.

Looking at the peak usage of the DDR resource for the ADPCM application if Table V, up to two instances of ADPCM should run fine on the same cluster with low performance impact compared to the standalone version; 3 and 4 instances should start to exhibit significant speeddown.

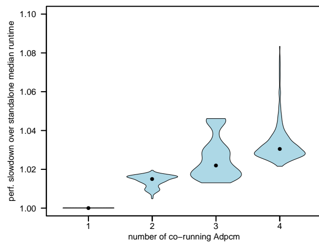


Fig. 13. Performance slowdown with difference number of co-running ADPCM.

Figure 13 depicts the performance variability while running an increasing number of ADPCM instances on a single cluster. All speeddown are normalized to the standalone execution time of ADPCM. Running two concurrent instances of the ADPCM benchmark produces a slight maximum performance degradation of +2%. This is in cope with the fact that the DDR controller is just below saturation. When running three concurrent iterations the maximum increases to +5%, and to +8% when co-running four different instances.

Even though the impact on runtime behavior is not that high, the behavior is correctly captured. The reason why the maximum performance degradation is only 8% is due to the fact that the average DDR controller usage of ADPCM is only 9%, far away from the peak usage of 47%.

To better capture the impact level on the architecture as future work, resource usage distribution could be used instead of peak usage for co-running performance prediction.

VII. Conclusion and Next Steps

In this paper, we presented a methodology and its associated automatic framework allowing us to characterize both the hardware and the safety-critical software relying on hardware monitors available in multi-core architectures and stressing benchmarks.

From the hardware point of view, we successfully quantified the shared hardware resource availability, an identified some undisclosed hardware features. From the software point of view, we were able to capture the median and peak resource utilization of the applications, allowing us to perform a first prediction on co-running application behavior.

References

[1] R. Azimi, M. Stumm, and R. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th international conference on Supercomputing*, ICS '05, pages 101–110. ACM, 2005.

[2] E. Bailey. Study report on anionics systems for the time frame 2007, 2011 and 2020. *European Organisation for the Safety of Air Navigation (EOSA)*, EUROCONTROL, Nov 2004.

[3] T. G. Baker. Lessons learned integrating COTS into systems. In *Proceedings of the First International Conference on COTS-Based Software Systems*, ICCBSS '02, pages 21–30, 2002.

[4] E. Duesterwald and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *International Conference on Parallel Architectures and Compilation Techniques*, page 220, 2003.

[5] D. Dvorak and M. Lyu. NASA study on flight software complexity. *Jet Propulsion*, page 264, May 2009.

[6] C. Ebert and C. Jones. Embedded software: Facts, figures and future. *Computer*, 42(4):42–52, April 2009.

[7] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. Program analysis and compilation, theory and practice. pages 12–52. 2007.

[8] Freescale. CodeWarrior Development Tools. http://www.freescale.com/webapp/sps/site/homepage.jsp?code=CW_HOME0. [Online].

[9] Freescale. P4080 Product Summary Page. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080. [Online].

[10] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001*, WWC '01, pages 3–14, 2001.

[11] R. Heckmann and C. Ferdinand. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. In *Proceedings of the conference on Design, Automation and Test in Europe*, DATE'05, pages 618–619, 2005.

[12] J. L. Hintze and R. D. Nelson. Violin Plots: A Box Plot-Density Trace Synergism. *The American Statistician*, 52(2):181–184, 1998.

[13] R. Kirner and P. Puschner. Obstacles in worst-case execution time analysis. In *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 333–339, 2008.

[14] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner. Using measurements as a complement to static worst-case execution time analysis. In *Intelligent Systems at the Service of Mankind*, volume 2. Dec. 2005.

[15] E. Mezzetti and T. Vardanega. On the industrial fitness of wcet analysis. In *Proceedings of the 11th International Workshop on Worst Case Execution Time Analysis (WCET2011)*. 2011.

[16] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. *European Dependable Computing Conference*, pages 42–52, 2012.

[17] PREDATOR. Design for predictability and efficiency. http://www.predator-project.eu/.

[18] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.

[19] P. Radojkovic, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *TACO*, 8(4):34, 2012.

[20] V. Salapura, R. Bickford, M. Blumrich, A. Bright, D. Chen, P. Coteus, A. Gara, M. Giampapa, M. Gschwind, M. Gupta, S. Hall, R. Haring, P. Heidelberger, D. Hoenicke, G. Kopcsay, M. Ohmacht, R. Rand, T. Takken, and P. Vranas. Power and performance optimization at the system level. In *Proceedings of the 2nd conference on Computing frontiers*, CF '05, pages 125–132, 2005.

[21] B. Sprunt. The basics of performance-monitoring hardware. *Micro, IEEE*, 22(4):64–71, 2002.

[22] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.

[23] P. Viola and M. Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.

[24] M. Wicks, M. Rangaswamy, R. Adve, and T. Hale. Space-time adaptive processing: a knowledge-based perspective for airborne radar. *Signal Processing Magazine, IEEE*, 23(1):51–65, 2006.

[25] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, T. Mitra, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, I. Puaut, R. Heckmann, F. Mueller, P. Puschner, J. Staschulat, and P. Stenström. The worst case execution time problem, overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, pages 36–53, May 2008.

[26] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, page 339, 2007.