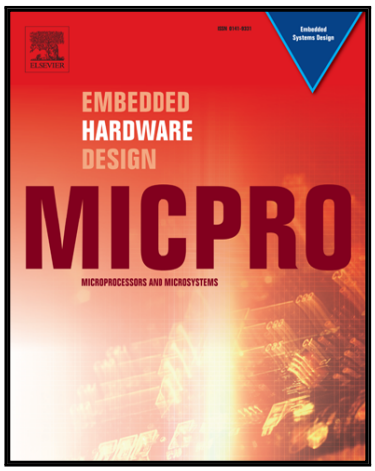

A software stack for next-generation automotive systems on many-core heterogeneous platforms

Paolo Burgio, Marko Bertogna, Nicola Capodieci,
Roberto Cavicchioli, Michal Sojka, Přemysl Houdek,
Andrea Marongiu, Paolo Gai, Claudio Scordino, Bruno Morelli

PII: S0141-9331(16)30455-0
DOI: 10.1016/j.micpro.2017.06.016
Reference: MICPRO 2586

To appear in: *Microprocessors and Microsystems*

Received date: 28 December 2016
Revised date: 3 June 2017
Accepted date: 21 June 2017

Please cite this article as: Paolo Burgio, Marko Bertogna, Nicola Capodieci, Roberto Cavicchioli, Michal Sojka, Přemysl Houdek, Andrea Marongiu, Paolo Gai, Claudio Scordino, Bruno Morelli, A software stack for next-generation automotive systems on many-core heterogeneous platforms, *Microprocessors and Microsystems* (2017), doi: 10.1016/j.micpro.2017.06.016

A software stack for next-generation automotive systems on many-core heterogeneous platforms

Paolo Burgio[a,*], Marko Bertogna, Nicola Capodieci, Roberto Cavicchioli[a], Michal Sojka, Přemysl Houdek[b], Andrea Marongiu[c], Paolo Gai, Claudio Scordino, Bruno Morelli[d]

[a] *University of Modena and Reggio Emilia, Italy*
[b] *Czech Technical University in Prague, Czech Republic*
[c] *Swiss Federal Institute of Technology in Zurich, Switzerland*
[d] *Evidence Srl, Pisa, Italy*

Abstract

The next-generation of partially and fully autonomous cars will be powered by embedded many-core platforms. Technologies for *Advanced Driver Assistance Systems (ADAS)* need to process an unprecedented amount of data within tight power budgets, making those platform the ideal candidate architecture. Integrating tens-to-hundreds of computing elements that run at lower frequencies allows obtaining impressive performance capabilities at a reduced power consumption, that meets the size, weight and power (SWaP) budget of automotive systems. Unfortunately, the inherent architectural complexity of many-core platforms makes it almost impossible to derive real-time guarantees using "traditional" state-of-the-art techniques, ultimately preventing their adoption in real industrial settings. Having impressive average performances with no guaranteed bounds on the response times of the critical computing activities is of little if no use in safety-critical applications. Project Hercules will address this issue, and provide the required technological infrastructure to exploit the tremendous potential of embedded many-cores for the next generation of automotive systems. This work gives an overview of the integrated Hercules software framework, which allows achieving an order-of-magnitude of predictable performance on top of cutting-edge Commercial-Off-The-Shelf components (COTS). The proposed software stack will let both real-time and non real-time application coexist on next-generation, power-efficient embedded platforms, with preserved timing guarantees.

Keywords: Autonomous Driving Assistance Systems, Many-core embedded

[☆] The Hercules project is funded by the EU Commission under the HORIZON 2020 framework programme (GA-688860).

[*] Corresponding author
Email addresses: paolo.burgio@unimore.it (Paolo Burgio), {first.lastname}@unimore.it (Marko Bertogna, Nicola Capodieci, Roberto Cavicchioli), {sojkam1, houdepre}@fel.cvut.cz (Michal Sojka, Přemysl Houdek), {a.marongiu}@iis.ee.ethz.ch (Andrea Marongiu), {pj, claudio, b.morelli}@evidence.eu.com (Paolo Gai, Claudio Scordino, Bruno Morelli)

systems, Predictable Execution Models, Real-Time Systems, Parallel programming models
2017 MSC: 00-01, 99-00

1. Introduction

In the next future, cars, and vehicles more in general, will be more and more "intelligent", and capable of taking independent decision on life-critical activities. In the next future, collisions and crashes will be reduced thanks to hazard detection and avoidance systems such as drivers' alert buzzes, proximity detection systems, and drowsiness [1] and health emergency detectors (e.g., heart attack [2]). Crossroads and traffic light interceptions will be safer and queues will be reduced, because cars will exchange information *on-the-fly* on their speed and route before reaching the crossing point, and take decisions accordingly [3]. Advanced features such as smart, advanced speed+distance control systems, will minimize the fuel consumption of a platoon of vehicles at close following distances, thanks to reduced aerodynamic forces acting on them [4]. We also expect that, in the next 15 years, taxi and car sharing companies, will shift their fleets to partially or fully autonomous vehicles. This is for instance the case of Uber [5, 6, 7].

Two are the technological breakthroughs that in the last decade paved the way to such an amazing future for automotive systems. On one side, the high degree of vehicle connectivity with the internet-of-things [8, 9] will provide on-board decision systems with a huge amount of information from the surrounding environment (Vehicle-to-Infrastructure) and from other vehicles (Vehicle-to-Vehicle), that can be used to take the most appropriate decision based on real-time, live data. On the other side, the tremendous increase of computational power available in the vehicle will support the hundreds of complex functionalities [10, 11] of current Advanced Driving Assistance Systems (ADAS) and –soon– of full-fledged self-driving cars.

From the technological viewpoint, building a complete self-driving car is extremely challenging, and commercializing and selling it is even harder. Luckily, despite our society and legislation are not yet ready, and the idea of fully autonomous vehicles might currently be too "futuristic" for the man in the street, for automakers (OEM and Tier-1 companies), the path to driver-less cars is clear since years. Several industrial-grade prototypes exist [12, 13, 14] and also few ones in academia [15], and technologically advanced countries such as U.S.A. and Germany already allow (partially and with limitations) testing of autonomous vehicles on their roads [16, 17]. We are entering right now an exciting period of transition and transformation, where different typologies of cars will be developed, with different **levels** of automation, as Figure 1 and Table 1 depict [1]. We expect to reach fully autonomous capabilities (autonomous Level 5) by 2030 [19].

In less distant future, the main challenge is to build ADAS that provide partial or limited set of functionalities, such as highway autopilot, valet park-

[1] Source: IHS [18]

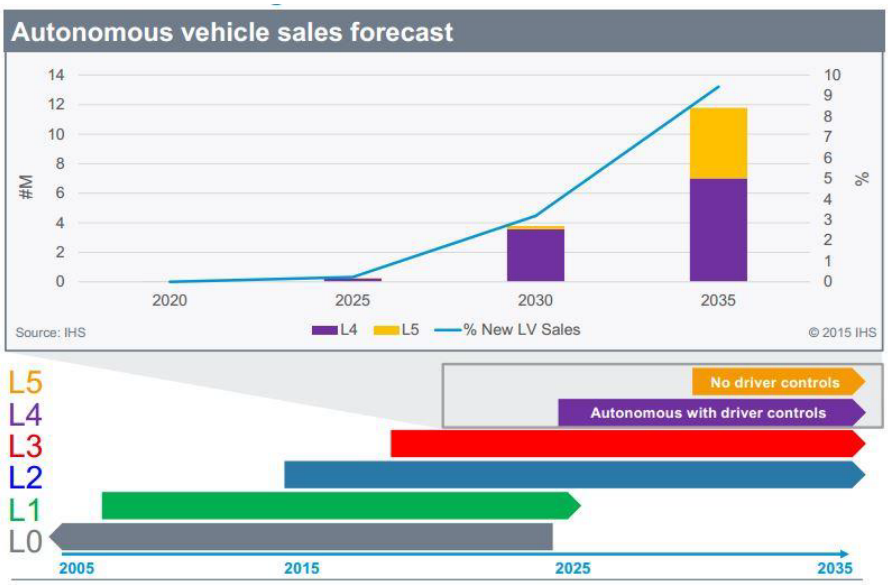

Figure 1: Autonomous vehicles sales forecast (source: IHS [18]).

ADAS level	Capabilities
L0	no autonomous functionalities (traditional non-autonomous car market)
L1	one/few autonomous functionalities like cruise control or assisted braking - *The market is currently here*
L2	at least 2 autonomous capabilities, like cruise control and lane-centering - *We are entering this level!*
L3	safety critical functions are performed to the vehicle, but only under certain environmental conditions. The driver is still required and needs to be able to take control at any time (the next generation of the market)
L4	fully autonomous, but the driver is still in charge of performing complex activities such as overtaking, or left-turn
L5	autonomous vehicle *without* human control

Table 1: ADAS levels (source: IHS [18]).

ing, emergency brakes, or the aforementioned health monitoring and platoon autopilot.

1.1. Building self-driving cars

This revolutionary change in the way we build our cars requires a technological shift also in the computing platforms, opening up a number of opportunities for innovation and research. All the main players of the automotive market are spending an increasing amount of resources in this direction. Major OEM and

Tier-1s such as BMW, Volvo, Tesla Motors, and General Motors are already developing the necessary know-how and technological background to build the next generation of automotive systems. Recently, even companies from other markets, such as Apple and Google, have entered this challenge (see the Google-Car [12]).

ADAS system engineers face a number of unprecedented challenges and requirements, which are far from being satisfied. Such a system, in fact, must:

1. manage compute-intensive sensor-fusion and image-processing;
2. run with reduced power consumption, allowing vehicles to be equipped with smaller batteries and renewable power sources;
3. quickly interact with the environment, requiring a prompt elaboration of sensor data;
4. execute all the above mentioned activities in a reliable and fault-tolerant way to take over safety-critical human activities.

Luckily, ADAS technology has moved a long way in last years, and today we are capable of meeting requirements 1, 3 and 4 employing powerful in-trunk compute servers. However, as of today, systems supporting these huge computational loads are extremely power-hungry, making them practically impossible to commercialize. The converging needs for predictable high-performance at low power call for a "real-time embedded super-computing platform", i.e., a platform capable of predictably providing real-time guarantees to applications running on top of power-efficient embedded hardware [2].

Modern Commercial-Off-The-Shelf (COTS) heterogeneous architectures based on multi- and many-core accelerators can satisfy this need for energy-efficient performance. Integrating multiple computing elements running at lower frequencies allows obtaining impressive performance capabilities at a reduced power consumption, while architectural heterogeneity enhances platform flexibility. Examples of such platforms are the NVIDIA Tegra X1 [20], a GPU-based System-on-Chip – SoC (described in Figure 3), and the Xilinx Zynq Ultrascale [21], which also embeds programmable logic. Unluckily, their tremendous potential in terms of performance/Watt comes at the price of increased architectural complexity, which ultimately makes writing efficient code extremely difficult (poor programmability). Even more importantly for the automotive domain, established methodologies and tools to provide real-time guarantees are born for single-core systems. When applied to many-cores, "traditional" techniques to achieve timing predictability make poor use of parallel/heterogeneous hardware, due to the conservative assumptions made. For this reason, the design methodologies and software stack for automotive systems must be heavily modified, and to some extent re-designed, to cope with the next generation of platforms.

Another important point is that, there is a plethora of *existing* applications and libraries, that must be supported in next-generation power-efficient ADAS. In industry, safety-critical software undergoes a long development and verification process, hence has a longer lifetime than "traditional" software (on the

[2]The capability of exactly **predicting** the timing behavior of applications is key to provide real-time guarantees, and ultimately to implement verifiable ADAS.

order of 20-25 years). It is therefore absolutely crucial to support also **legacy** code, with minor or no modifications at all.

Last but not least, the software running safety-critical tasks must be developed according to certified development process, and to strict safety standards. This certification aims at preventing possible injuries due to misbehaving or faulty software. In the case of automotive, in particular, the OSEK/VDK and the newer AUTOSAR standards already establish a set of rules and constraints for operating systems design. These standards impose the usage of a real-time operating system (RTOS) implementing a set of well-known scheduling algorithms and techniques. Usually, the size of the RTOS is kept at a bare minimum to reduce the code complexity (hence the number of possible bugs) and the costs of the verification process. By doing so, it is possible to implement the Freedom From Interference at the RTOS Level, as specified in the ISO26262 Part 6, annex-D [22].

1.2. Paper positioning: the Hercules project

These are the motivations behind the Hercules ("High-Performance Real-time Architectures for Low-Power Embedded Systems") Project [23]. The ambitious goal of Hercules is to obtain an order-of-magnitude improvement in the cost and power consumption of next generation real-time applications for safety-critical domains. This goal will be pursued by mixing a certified RTOS and a full-fledged operating system on a high-performance many-core COTS hardware.

This paper introduces the software stack envisioned in Hercules. To do so, we show some of the design choices characterizing current automotive systems, guided by industrial requirements from project partners, namely Airbus Group Innovations, Magneti Marelli S.p.A. and Pitom snc. We first describe the hardware system architectural template considered in the project (Section 2), based on existing high-performance, power-efficient COTS platforms available on the market. In Section 3 we describe the full software stack for automotive systems running on top of many-core heterogeneous platforms. In Section 4 we describe which one(s) of the existing programming models for heterogeneous many-core platforms better suits Hercules requirements. Our choices also take into account industrial needs, like the reuse of *legacy code and libraries on heterogeneous many-core devices without requiring heavy code re-factoring*. This allows maximizing the industrial impact of the Hercules framework (methodologies, tools and software), simplifying the technological transfer to existing application scenarios. Section 5 shows how virtualization techniques are employed to decouple the low-level, hardware-dependent layers of the proposed stack, and the higher application-dependent layers. The proposed hypervisor ensures the necessary spatial and timing isolation to meet Real-Time high-performance requirements of modern ADAS. The Operating Systems chosen to be part of the stack are described and motivated in Section 6, while Section 7 introduces state-of-the-art techniques for resource scheduling in many-core systems that are implemented in the proposed software stack. Finally, Section 8 draws some conclusions.

2. Target architecture

The choice of the target computing platforms is crucial and it affects every part of the technological stack of the project. Most of the existing real-time systems run on embedded architectures that were not been thought to address the predictability and analysability requirements of time-critical applications [3]. The few platforms designed for being fully timing analyzable became quickly obsoleted by later process technologies. For this reason, Hercules employs Commercial-Off-The-Shelf (COTS) components.

The advantage of using COTS are multiple: they are cheaper than custom-made solutions; they are usually more robust to hardware and timing faults; fully supported by the hardware provider; and easily available for market exploitation.

Hercules targets heterogeneous architectures, featuring a "traditional" high-performance host core (such as 64-bit ARM Cortex-A or Intel iX) and a many-core accelerator, such as GPUs [20], possibly also coupled with FPGA logic [21]. The techniques, rationales and tools developed within Hercules will be designed to be easily portable to future platforms. This is possible thanks to the software stack and highly expressive programming models chosen, which allow hiding the complexity of the hardware architecture, and ensuring application portability. Figures 2 and 3 depict these two platforms.

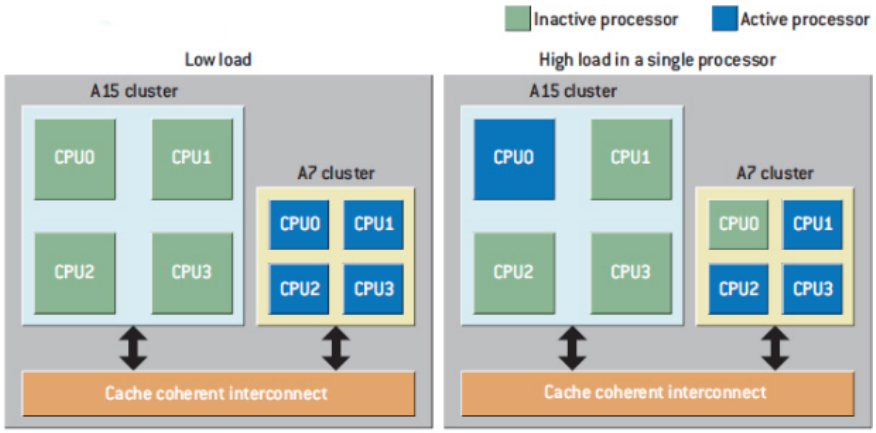

Figure 2: Scheme of the ARM big.LITTLE platform.

The **ARM Big.LITTLE** architecture [24] (2011) represents the state-of-the-art for the target "host" subsystem. It couples powerful "big" cores such as Cortex-A57 and slower yet more power-efficient "little" cores such as Cortex-A53 (both implement ARMv8 ISA). Since the two subsets of cores share the on-chip

[3] In order to simplify platform analysis, a typical solution is e.g., to exploit **only one core** of a multi-core system, leaving the other processing units disabled or – even worse for the power consumption – completely idle. This is clearly extremely inefficient.

memory banks, and caches are coherent, workloads may migrate between them almost on-the-fly. This is typically performed transparently by the OS.

Hercules will employ a *Heterogeneous Multi-Processing (HMP)* model, which allows concurrently exploiting all physical cores at once, as opposite to the "traditional" *clustered switching* model, where only one subsystem is active at each time. **Tegra** [20] is a family of NVIDIA SoCs explicitly targeting embedded

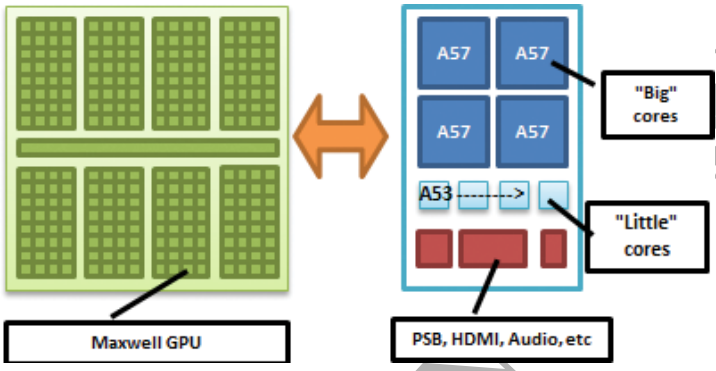

Figure 3: Scheme of the NVIDIA Tegra X1 platform.

systems such as tablets and smartphones. Figure 3 shows that. It couples a "host" subsystem based on ARM multi-cores, and a General Purpose GPU (GP-GPU) in a single package. Poorly parallelizable, control-based and I/O computations are typically executed on the host subsystem, while highly parallel workloads are offloaded to the power-efficient many-core accelerator. The latest release of the family is the Tegra X1 [20] platform that embeds an octa-core host with Big.LITTLE configuration and a Maxwell GPU with 256 CUDA cores. The platform is claimed to achieve 1 TFLOP of computing power, within only 15 Watts. Tegra X1 is not qualified according to *Functional Safety and Road Vehicles Standard (ISO 26262 [22])*. However, NVIDIA declared that the next version of the platform — called Drive PX2 [25], and based on the novel Parker architecture — will be qualified at least ASIL-B [26]. Since the two platforms are similar from the architectural point of view, the technology produced by Hercules on the Tegra X1 will be easily ported to the Drive PX2, as soon as it will be available, i.e., Q2 of 2017.

The architecture depicted in this section does not cover the full spectrum of modern heterogeneous many-core platforms, yet it is are quite representative of current market trends and products. The Hercules project targets future automotive and avionics systems, whose computational platforms are expected to provide hundreds of GFLOPs within a few Watts [4]. This is the main reason behind the selection of Tegra-like platform as one of the reference architectures of the project.

[4]Giga-FLoating point OPerations-per-Watt: a well-known metric for e the computational power of embedded computing platforms.

3. Hercules software stack

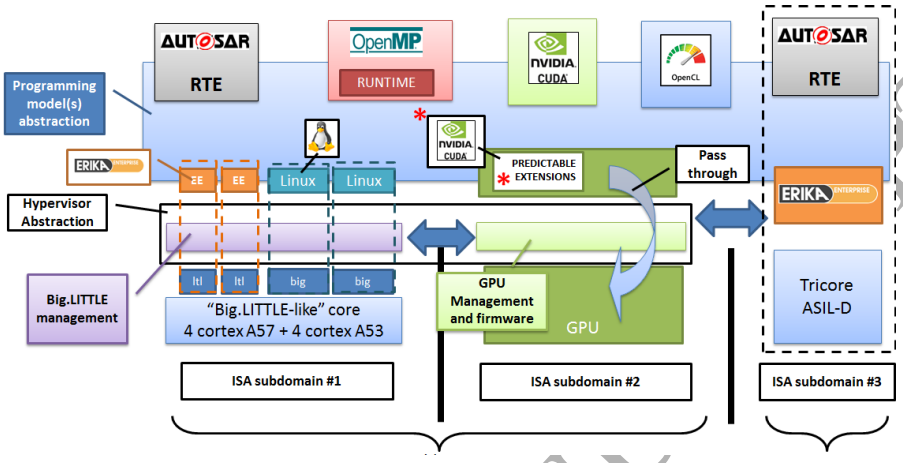

Figure 4: The Hercules Software Stack.

Figure 4 depicts the Hercules software framework. As said, the Hercules project is not tied to a specific system/SoC, yet it targets a specific architectural template, coupling a certified hard-real time platform and heterogeneous SoC with multi-core host and many-core accelerator/GPU.

Hercules aims to support on the same architecture both real-time AUTOSAR-like applications [27] running on top of ERIKA Enterprise [28], and non-real-time (but high-performance) computations performed partly in the Big.LITTLE-like subsystem and partly in the many-core accelerator. In addition to this requirement, it aims to support ISO 26262 certification [22] of some of the safety critical parts. Then, it becomes mandatory to guarantee a proper isolation between the hard real-time parts and the rest of the system, in order to obtain the freedom from interference required by the standard. For these reasons, the hard real-time subsystems have been properly "isolated":

1. For high levels of safety requirements, the project is planning the integration of an AUTOSAR subsystem running on an external ASIL-D compliant CPU, such as the Tricore AURIX [29]. The external CPU will be connected to the many-core fabric using a predictable communication infrastructure. However, this is not part of the "core" research activities of the project.
2. For lower levels of safety requirements, the real-time subsystem will be integrated in the Big.LITTLE cores. To ensure freedom from interference, we will rely on a **hypervisor** to separate and isolate the real-time subsystem (run under the ERIKA Enterprise kernel) from the rest of the system.

From the programmability viewpoint, complex many-core based systems can easily become a nightmare even for experienced programmers, and highly-

expressive programming models are the keystone for extracting the full performance/Watt available in the platform. As an addition, *predictability* is a key requirement in real-time and safety-critical applications, and it must be guaranteed at every level of the HW/SW platform. Unluckily, current programming models for parallel architectures were not designed for real-time systems. They lack of the necessary expressiveness to express, e.g., real-time task periods and deadlines, being inadequate for the goals of the project. For this reason, a key contribution of Hercules is to deeply analyze the available programming models for heterogeneous many-core systems, and discuss how they can be enhanced with **predictability extensions**, to make them suitable also for the real-time domain. We will start this discussion in the next session.

4. Programming model

There is a plethora of *de-facto* or *de-jure* standards for programming heterogeneous architectures, which we want to address to ensure not only compliance with legacy code and software libraries, but also with *existing methodologies, tools and, most of all, programmers' expertise*. In this section, we introduce the main advantages and drawbacks for each of them.

Hercules targets energy-efficient GPU-based platform. For this reason, CUDA [30] and OpenCL [31] are the first choice as reference programming models. Unfortunately, they are extremely low-level, and designed for achieving high performance, not predictability, hence they lack the necessary flexibility required by the project. As an addition – and this is especially the case of CUDA, it is not easy to "customize" and extend them with appropriate extensions and language constructs to achieve timing predictability. On the other hand, highly expressive directive-based programming models such as OpenMP [32] and OpenACC [33], specify parallelism at a much higher abstraction level, leaving a lot of room for compiler optimizations and code transformations. Last, but not least, Hercules will release software "as much as possible open-source", and this might become an issue with proprietary/closed frameworks such as CUDA. For all of these reasons, and in order to provide a clean and simple programming interface, we embrace directive-based programming front-ends such as OpenMP/OpenACC in the project. The Hercules tool-chain will employ compiler transformation to convert high-level directives to lower-level programming routines written in CUDA or OpenCL. We are designing a compiler infrastructure to transform the OpenMP program into its predictable counterpart (see Section 7), by emitting optimized code in CUDA/PTX format. As explained in the following Section 4.1, the Hercules ecosystem will also support legacy hard-real time applications written for AUTOSAR [27].

4.1. Programming heterogeneous platforms

OpenMP [32] is the *de-facto* standard for programming shared-memory systems. OpenMP was developed at the end of 90's to program regular, loop-based workload on top of symmetric multiprocessors systems (SMP) with shared-memory. More recently [34], it evolved to deal with more irregular and dynamic parallelism, switching from a loop-oriented approach to a *task*-oriented

approach. Finally, with specifications 4.5 [32] (2011), it also embraced heterogeneous computing paradigm and execution model by introducing subroutines called *target regions* to be offloaded to an accelerator device, partially relaxing the original SMP-based execution and shared-memory models. In its specifications, OpenMP is a set of APIs, pragmas and environmental variables. To implement a full parallel software stack, it relies on a run-time which provides basic functionalities for threading and resource allocation/management. The actual run-time implementation, and its set of APIs, are compiler-specific: for instance, the most known GNU port relies on the GNU GCC-OpenMP (GOMP) framework [35]. Although some efforts have been made [36], [37], OpenMP is not yet suitable for real-time computing, and the standard does not include support for real-time computing.

The preferred solution for programming NVIDIA GP-GPUs are either CUDA [30] or OpenCL [31]. CUDA provides a set of APIs for (massive) threading and hooks for data movement/placing on the GPU device. Application code runs on top of a run-time library + GPU driver which works together with the operating system to provide these services. The main drawback of CUDA – and of all offload-based programming models in general – is an increased software complexity with respect to, for instance plain C or C++ code, both in terms of lines of code, and to the fact that the programmer must manually partition the application onto computing *threads* and *groups*, and to explicitly orchestrate data movements to/from the GPU device. From the "predictability" viewpoint, unfortunately, there are no implicit real-time guarantees in the CUDA standard, and original CUDA run-time and drivers are closed and proprietary. Figure 4 shows possible "real-time CUDA extensions" which might be developed during the project. They are marked with a star. The GPU is currently seen as a non-preemptible, shared resource with run-to-completion semantics. In the project, we will explore the possibility of relaxing these assumptions by adding preemption support and concurrent programming capabilities of a single GPU device.

Open-Computing-Language (OpenCL [31]) is a joint effort by the Khronos Consortium [38] for building an open language for programming accelerator-based platforms. Similarly to CUDA, it provides non real-time APIs for threading and memory management at the application/user level, relying on a run-time+OS+drivers subsystem. Similarly to CUDA, OpenCL increases the complexity of application code, and does not provide real-time guarantees.

4.2. Automotive programming models and AUTOSAR RTE

The previous subsections introduced a set of programming models for non real-time parallel software. In the automotive domain, on the other hand, we currently see two diverging trends. On one side, the real-time, statically allocated, statically configured AUTOSAR standard [27] proposes a complete software stack including device MCAL drivers [5], Basic Software, RTOS and Run Time Environment (RTE) to implement a standard software component

[5] A Micro-controller Abstraction Layer is a software module that directly accesses on-chip controller peripheral modules and external devices that are mapped to memory, and makes the upper software layer independent of them.

model. On the other side, the infotainment world typically relies on non real-time versions of Linux, Android, and proprietary solutions coexisting in the same system.

One of the main goals of Hercules project is to harness the computational power of next-generation parallel embedded platforms, inside a framework where AUTOSAR-compliant real-time applications may concurrently run along with infotainment and non real-time software, without affecting the required timing guarantees. For this reason, the project aims to support a subset of the AUTOSAR specification by extending the operating system with a minimal RTE support, coupled with appropriate mechanisms allowing the sharing of data as well as the concurrent usage of common peripherals (see Section 3). From a research perspective, Hercules will mainly focus on offload-based languages such as OpenMP, CUDA and OpenCL. Despite supporting AUTOSAR is not the main focus of the project, its adoption enables us to support legacy automotive code, maximizing the penetration of the produced technology in industry.

In the Hercules software stack, the operating system layer is based on ERIKA Enterprise [39, 28], an open-source OSEK/VDX certified OS, and on the real-time versions of the Linux kernel. They are discussed more in detail in Section 6.

By allowing the integration of AUTOSAR components together with high-performance software stacks, the consortium aims at ensuring portability of code provided by several software suppliers with different degrees of real-time support. Special attention is also given to the possibility of running code certified under the ISO 26262 automotive functional safety standard [22]. Although the project does not aim at providing a full ASIL-D certified stack, the potential usage of the architecture proposed in safety applications will be analyzed, and recommendations for the creation of a certifiable stack will be produced.

5. Virtualization in Hercules

In the Hercules software stack, we employ an hypervisor to achieve the spatial and timing isolation between software components necessary to provide predictability and real-time guarantees. The project is currently evaluating several options, starting from existing open-source projects. The choice of the hypervisor will be guided by a set of requirements, such as:

1. possibility of running multiple operating systems (i.e., Linux and ERIKA Enterprise [28]);
2. possibility to support core assignments (*pinning*) to the single guest OS; it is important to state that we are not aiming at the coexistence of a high number of Virtual machines on the same CPU (as it happens in cloud environments), but rather the typical setup will statically allocate a single OS to one or more CPUs, to limit the virtualization overhead, still maintaining the separation needed by the safety standards;
3. possibility to share peripherals such as GPUs and communication busses;
4. possibility to be certified, which typically means choosing hypervisors which have a minimal footprint (in the order of 10k lines of code — LoCs);
5. possibility to support heterogeneous architectures, such as the many-core systems used in the project.

We employ a virtualization mechanism, composed of one hypervisor module that manages sub-domains in the system, hiding platform complexity to programmers, and to provide applications with spatial and timing isolation. This is shown at the bottom of Figure 4. In order to support the HW/SW partitioning, predictable communication will be implemented, among the multiple hardware sub-domains (the horizontal double-arrows in Figure 4).

5.1. Virtualization of the host subsystem

Virtualization is widely adopted in general purpose platforms, especially for cloud computing. One of the most widely adopted open source hypervisors is probably Xen. A recent effort to enhance Xen with real-time capabilities is the project **RT-Xen** [40] (in mainline Xen since v4.6). It implements a hierarchical real-time scheduling framework based on global EDF scheduling, within approximately 100K LoCs of C code. Unfortunately, the size of the hypervisor is crucial, as the effort for system verification, hence, certification, grows significantly with the number of LoCs. For this reason, micro-solutions such as Jailhouse or Xvisor are strongly preferred in this project.

Jailhouse [41], developed mainly by Siemens, is a Linux-based hypervisor oriented to real-time and safety applications—so called *inmates*. Jailhouse isolates the virtual machines, called *cells*, with few lines of code (13513 written in C), removing all of the unnecessary features (e.g., hooks for diagnostic tools), and schedules the virtual machines by pinning them to the computing cores. It also allows running bare-metal applications alongside to Linux.

Xvisor [42] is bigger than Jailhouse (460k lines of C code) but offers many high level features like Xen and KVM, keeping a small memory footprint (around 2MB) and a higher performance compared to them.

5.2. Porting the Jailhouse Hypervisor on Embedded platform for automotive

During the initial part of the project, we have ported the Jailhouse hypervisor on the Nvidia Tegra TX1 platform, supporting the default Linux kernel 3.10 provided by Nvidia. The source code has been made publicly available through the GitHub platform [43]. This activity has also created the opportunity for improving most of the existing documentation of the original project.

On such a platform, we have experimentally measured a jitter between 8 and 10 μs for the standard Jailhouse demo consisting of a periodic timer interrupt. The time for issuing a hypercall from a bare-metal application running on a dedicated core has been measured between 600 nsec and 2 μs. This time is about the double of the time needed for issuing a Linux system call on the platform (whose minimum duration has been measured equal to 260 nsec).

The next step will concern the porting of the ERIKA Enterprise RTOS on the platform. This activity will let creating the AMP [6] architecture envisioned in Hercules, consisting of an automotive-grade RTOS alongside the Linux OS under the supervision and enforcement of the hypervisor. We will then design and develop efficient mechanisms for communication and synchronization between the two OSs. Such a software architecture will be also ported to different platforms, including the Xilinx Zynq Ultrascale [21].

[6] Asymmetric Multi-Processing

5.3. Proxy cells for resource sharing

With regards to mainstream hypervisors like Xen, despite the recent interest in augmenting their real time capabilities with specifically designed patches (e.g. **RT-Xen** [40]), they still lack proper system-wide arbitration policies for resources other than CPU cores, resulting in unwanted interference that reduces the analysability and predictability of the whole platform. One of our recent works [44] made explicit the fact that, even if RT-Xen allows us to specify a real time domain with a global EDF virtual CPUs scheduling, systematic deadline misses occurs when tasks belonging to the non-real time domain interfere with resources that are used (even sporadically) by the real time domain. In particular, we show that if a non-real time task enqueues many big IO storage requests, the real time domain might starve for as long as a second before having its own IO request served; this was observed to happen no matter how small is the request coming from the critical partition. Mitigation of such effects is usually obtained by exploiting OS-level arbitration systems, such as *cgroups* in Linux: however, such mechanisms are known to provide an insufficient level of control granularity as cgroups rely on token bucket based mechanisms (known for their bursty nature) or proportional sharing (also proven to be based on suboptimal strategies [45]). On the top of that, the additional overhead of managing resources both at the OS level and hypervisor level poses additional problems.

The idea of the Hercules for addressing such challenges is to exploit the innovative design enabled by the Jailhouse hypervisor of assigning devices in an exclusive way either to the root cell or to one of the bare metal applications running on inmates. Inmates are known to provide extremely low latencies, hence minimal drivers might run as bare metal applications (or on top of small footprint RTOS such as Erika) to have pass through access to such resources. It is trivial to understand that such solution (encouraged by Jailhouse "philosophy") represents an unacceptable restriction when we want such inmate assigned resources to be used also by other cells. We tackle this with a resource sharing mechanism we are developing, based on the concept of *Jailhouse proxy cells*. This is enabled by defining a region of shared memory that acts as a mailbox messaging system among cells. Our solution is shown in Figure 5.

In particular we implemented it on the NVIDIA TK1 developer board [7]. The Tegra® K1 System on Chip (SoC), which includes a quad-core ARM® A-15 CPU and an NVIDIA Kepler GPU with 192 CUDA cores. The developer kit comes equipped with 2 GB of RAM and many different I/O peripherals such as UART serial port, HDMI, GPIOs, USB, etc. The lower part of the figure shows which memory mapped devices of the board are made exclusively available to the different Jailhouse cells. Following a simple color code, the white peripherals (USB, Display and PCIe) are bound to the *root* cell, which runs Debian, while the rightmost one (UART) is accessible only by the *inmate* cell. The darker (SHMEM) region is the shared memory that we defined in order to implement the mailbox messaging system: here three memory location are defined (*a,b* and *c*) that are readable and writable by both cells.

[7] The Tegra X1 was not available when we carried this exploration. However, the two boards are equivalent from an architectural viewpoints.

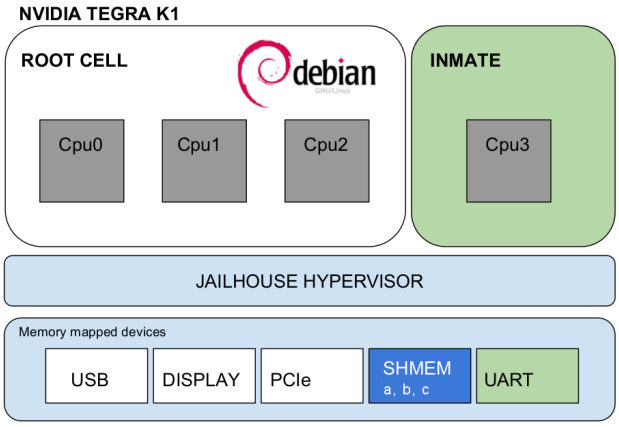

Figure 5: UART virtualization by the means of Jailhouse proxy cell. Implementation on NVIDIA TK1 board.

We set the different cell configuration mapping the different devices and memory to different physical addresses of the RAM. In this setup we wanted to make the *root* cell able to communicate using the serial port, which is not directly accessible because it is in exclusive use of the *inmate* cell. To achieve this, the two cells communicate through the shared memory using a simple protocol: the requester writes on memory location a the size of the data that has to be transmitted and the actual data on memory location b, which is of fixed size *data-size*; the proxy cell reads the request (time-triggered) and starts reading the data from memory location b, notifying at location c when it finishes. If the size of data is greater than *data-size*, than the requester writes the second chunk of data at location b and the procedure repeats. The proxy cell can eventually publish the data received on the UART port at the end of the transmission.

This simple scenario is just a proof-of-concept, but it is trivial to extend the proxy cell concept to whatever device that needs low latency capabilities or real-time constraints, provided that it is not possible to offer higher capabilities than the ones offered by the hardware resource or by the OS capabilities of the requester cell.

5.4. GPU management and virtualization

As GP-GPUs became mainstream, there was a big interest of virtualizing graphic cards, e.g., for cloud computing. Unfortunately, "hiding" one or multiple GPUs under a hypervisor introduces a serious performance penalty for crossing its software layers, which ultimately might compromise the advantage of many-core acceleration. For this reason, a common solution is to provide a so-called *pass-through* mechanism for the CUDA drivers, which are allowed to bypass the virtualization layers and directly access the device. It is depicted in Figure 6. This mechanism is represented by the arrow in Figure 4 that directly accesses the GPU, and it's currently supported on a limited set of GPUs, and for specific

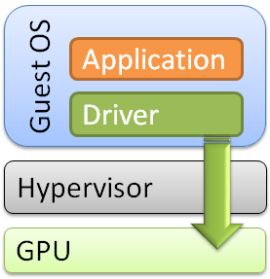

Figure 6: GPU pass-through.

drivers[8]. A number of hypervisors and virtualization schemes exist for GPUs. Interested reader might refer to [47] as a good survey.

6. Host Operating Systems: ERIKA and Linux

For the host part, we decided to start from the application requirements collected by the partners, in order to build an innovative infrastructure providing good performance while supporting legacy code.

The critical tasks of automotive vehicles need to be executed by a RTOS compliant to well-known safety standards (e.g., OSEK/VDX, AUTOSAR). Such RTOS is typically offered by companies specialized in the automotive domain (e.g., Vector, ElektroBit, ETAS) under commercial royalty-based licenses.

The growing interest for using a general-purpose operating system (OS) for the execution of real-time tasks has involved the automotive market too. Such interest aims at both lowering the production costs and providing a higher number of functionalities. The Linux OS, for example, has recently integrated the real-time CPU scheduler SCHED_DEADLINE [48] originally developed by Evidence Srl in the context of the ACTORS FP7 project. This scheduler is based on the Earliest Deadline First (EDF) algorithm and provides Resource Reservations among the running tasks: each task is guaranteed to meet its timing constraints regardless of the behavior of the other tasks executing in the system. In parallel, the Linux Foundation started financing the PREEMPT_RT [49] and the Automotive Grade [50] projects.

Unfortunately, the code size and the lack of full determinism of general purpose OSs do not make them eligible for safety-critical tasks in the automotive domain. Nevertheless, a number of on-going efforts (e.g., AUTOSAR Adaptive) aim at using such OSs for non-critical activities within the vehicle.

6.1. Why Erika Enterprise and Linux

The choice of the ERIKA Enterprise RTOS let the Hercules stack supporting legacy AUTOSAR applications on top of an open-source implementation.

[8]For instance, NVIDIA published [46] a list of applications which are certified for this technology (called NVIDIA Grid).

ERIKA Enterprise [28] is currently the only open-source OSEK/VDX certified operating system, implementing a subset of the extensions specified by the AUTOSAR OS standard. This opens the possibility to run legacy automotive applications with minimal or no changes. For this reason, we are currently implementing Big.LITTLE support in ERIKA Enterprise.

The Linux OS is the best candidate for running on the host core of the envisioned hardware architecture illustrated in Section 4. There are multiple reasons behind this choice: the excellent throughput within reasonable response times (that can be further reduced through additional real-time patches [9]), the high number of SoCs and peripherals already supported, the built-in support for big.LITTLE architectures that will be further improved by next-coming patches by ARM and Linaro [10], the extreme customizability both at compile- and at run-time. More in details, the project will leverage the advanced SCHED_DEADLINE [48] scheduler, recently integrated into the official Linux kernel. This component will be further improved by integrating predictable power-management algorithms (e.g., synchronization with the `cpufreq`'s schedutil governor) and better scheduling strategies (e.g., CPU reclaiming [51]), to realize an energy-efficient real-time Linux-based run-time for the host processors.

7. Scheduling of shared resources

7.1. Memory accesses and the PRedictable Execution Model

In order to achieve predictable execution on heterogeneous many-core platforms, it is necessary to control the way how individual CPU cores and on-chip peripherals access the shared resources such as on-chip interconnects and main memory. Of these, the main memory is the slowest one and likely to be the bottleneck. Hence, we focus our attention to predictable sharing of it.

The approach proposed in Hercules is inspired by the so-called PRedictable Execution Model (PREM) [52, 53], where the predictability of memory accesses from single software components (*tasks*) is increased using prefetching techniques and scheduling prefetch bursts from different cores to not interfere with each other.

Under PREM, tasks are split into pairs of *memory* and *computational* phases. Figure 7 shows the distribution of memory accesses both in PREM and non-PREM models. In a first memory phase, tasks retrieve and copy data from the main memory into the local cache of the core they are executing on, whereas, in the following, computational phase, they elaborate non-preemptively previously-cached data. This execution model allows the variability of memory-contention latencies to be greatly reduced, by explicitly controlling memory accesses during memory phases. As such, it allows the overall task execution times to become much more predictable. Addressing single-core systems, a PREM-compliant co-scheduler is proposed granting main-memory access only when the task being executed on the processor is in the computational phase, without incurring memory conflicts.

[9] E.g., *PREEMPT_RT*, https://wiki.linuxfoundation.org/realtime/

[10] Energy-Aware Scheduling (EAS), https://www.linaro.org/blog/core-dump/energy-aware-scheduling-eas-project/

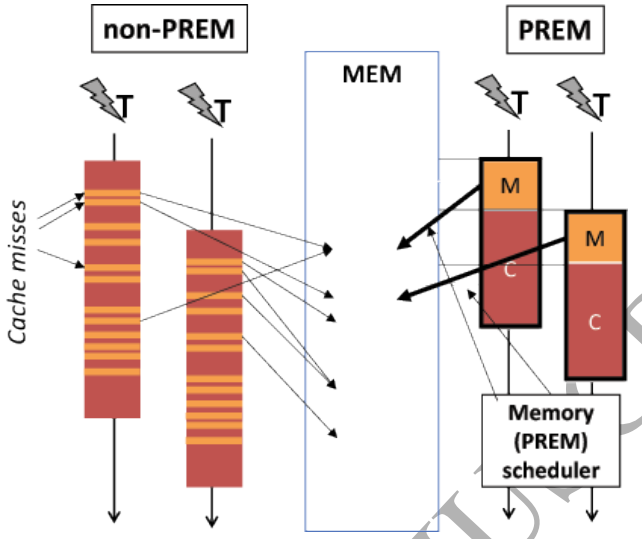

Figure 7: PRedictable Execution Model in a parallel environment.

To enforce this scheduling policy, the co-scheduler relies on the presence of a *Real-Time Bridge*, which arbitrates the access to memory in a time-sharing fashion. This is however **not** the case of Hercules, whose approach is *completely on the software point of view*, i.e., it does not need additional hardware other than the one which is usually already shipped embedded in a board (e.g., one or more DMA engines). To apply the PREM approach to the SoCs selected for the project, it is not sufficient to deal with software running on the CPUs but one has to take into account other on-chip peripherals accessing the main memory, such as GPUs, Ethernet, Video Input (VI) etc. The following sections describe how this is done in Hercules.

7.1.1. Memory accesses from host CPUs

On the CPU side, scheduling of PREM-based prefetch bursts leads to predictable execution only when the application can actually be converted to the PREM-compatible way of execution. This is however not a trivial task, because, for instance, several ADAS applications already leverage a host-accelerator model, where data transfers among the two subsystems are made explicit, e.g., with buffers (OpenCL or CUDA) or code annotations (OpenMP/OpenACC pragmas). As an addition, data prefetching is widely employed as a performance booster for embedded applications, and code written with such techniques naturally tends itself to PREM. On the other hand, there are certainly cases where either the conversion is not possible/straightforward, or the application might not follow the PREM rules due to bugs. The Hercules software stack handles these cases by providing a *throttling* mechanism for preventing uncontrolled memory access. The mechanism is inspired by the *MemGuard* tool [54], that was proven to increase system predictability by providing timing and spatial isolation among software components in real-time systems. The difference be-

tween the work in [54], and the Hercules approach, is that we will implement throttling within the hypervisor on Tegra's ARM sub-domain, rather than the Linux kernel running on x86 processor.

The MemGuard-like throttling mechanism uses performance counters to monitor cache misses made by virtual machines (VM). Whenever the VM exceeds the cache miss budget allocated to it, an interrupt is generated and the hypervisor pauses the VM execution to protect other VMs from unwanted interference.

To implement the MemGuard-like throttling on the ARM platform, it is necessary to select the proper performance event to be counted by the performance counter. Performance events are defined in ARMv8 reference manual [55], but quite a lot of details is left implementation defined. It is therefore crucial to properly analyze the semantics of performance events on the given ARM implementation – NVIDIA Tegra X1 in our case.

The following events are candidates for using in the throttling mechanism: L2D_CACHE_REFILL, L2D_CACHE_WB[11] and BUS_ACCESS. From our experiments, we conclude that L2D_CACHE_REFILL can be used for counting of memory reads, L2D_CACHE_WB for memory writes and BUS_ACCESS for both reads and writes. BUS_ACCESS is the best candidate for use in throttling. One has to be aware that, as follows the our experiments, there are four BUS_ACCESS events per a single memory access.

7.1.2. Memory accesses from on-chip peripherals

Contention among CPUs is not the only source of non-predictability. Most on-chip peripherals can access the main memory as well and interfere with the CPUs. A conceptual diagram of how these controllers might interact is shown in Figure 8. A part of the problem is that, it is not easy to control when

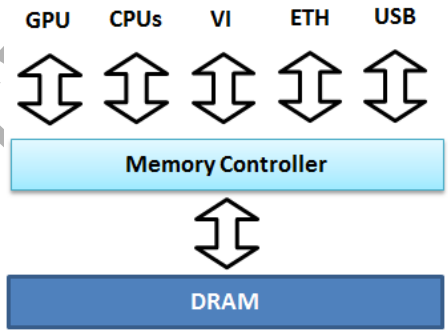

Figure 8: Conceptual diagram of NVIDIA Tegra X1 memory controller and its on-chip clients.

the memory is accessed. For example, the Ethernet controller stores incoming packets to the main memory when they arrive and the CPU has little control over this process. To eliminate this source of unpredictability, we studied the

[11] WB stands for write-back

details of the Memory Controller (MC) on the Tegra X1. Figure 9 describes the memory hierarchy of the platform, and highlights possible source of contention. This is part of the analysis carried on in [56]. The MC can be reconfigured to limit the memory bandwidth allocated to non-CPU clients and use this to limit the interference from the selected memory clients.

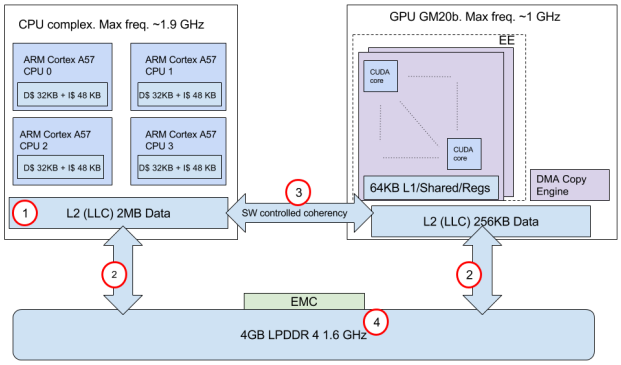

Figure 9: A simplified overview of the Tegra X1 memory hierarchy, with notable memory contention points: 1) L2 cache shared by the four cores; 2) contention of memory bus from different cores; 3) coherency protocol on Last-Level Cache (LLC), and 4) access arbitration and traffic shaping by memory controller.

Figure 10 shows the results of our experiments with throttling the GPU. The horizontal axis shows the level of throttling and the vertical axis the memory bandwidth consumed by a simple memory-bound GPU kernel. One thing the graph shows is that the memory bandwidth available to the GPU, which is inversely proportional to the kernel execution time, depends on the memory activity of the CPU(s), i.e. the memory accesses from the CPU have priority over GPU accesses[12]. By changing the memory controller configuration, it is possible to limit the GPU memory bandwidth so that the GPU kernel execution time is independent of CPU activity (throttle value of 8 in Fig. 10).

7.1.3. Summarizing memory scheduling

We are currently implementing a system where the throttling mechanisms described in the previous sections is used together with the goal of limiting the interference caused by uncoordinated accesses to the shared memory. In other words, we will co-schedule the applications and hardware resources via the run-time reconfiguration of the throttling mechanisms implemented in the hypervisor module. The results will be not only the increased predictability, but also resistance of the system to potentially misbehaving application.

The co-scheduling is based on several sources of information – off-line analysis made by the compiler (with the hints from programmers) and on-line information based on light-weight tracing of run-time activity (e.g. performance counters).

[12]Unfortunately, it is not possible to change memory client priority on Tegra X1

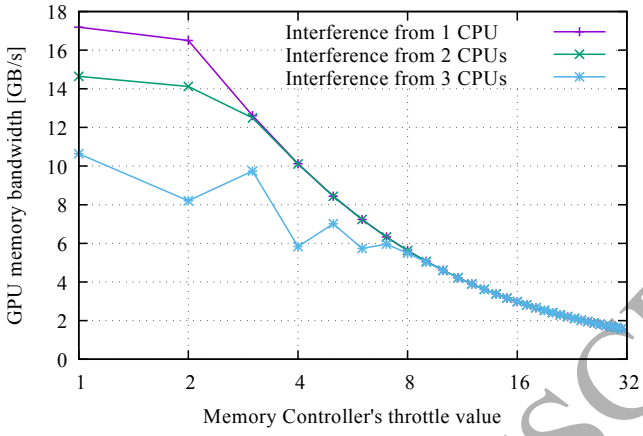

Figure 10: Throttling of GPU client at the memory controller level.

Exploiting all available information about the application together with using hardware features of the modern platforms, allows us to improve the predictability of the overall system.

8. Conclusions

This paper describes the goals and organization of the Hercules H2020 project [23], a first attempt of building a complete software stack for automotive systems based on commercial-off-the-shelf components, that is also able to interface with an ASIL-D certified subsystem for running legacy, hard-real time workload.

As a summary, we hide the complexity of the underlying platforms by means of virtualization, and provide support for two kind of operating systems. On one side, a statically configured instance of ERIKA Enterprise, pinned to one of the "LITTLE" cores, allows running static real-time applications typical of the automotive market. On the other side, Linux with RT extensions (typically running on all the remaining CPUs of the "Big" subsystem) is devoted to more computationally intensive dynamic workloads. The small footprint of the adopted hypervisor (few lines of code) opens the possibility of a functional safety certification path following the ISO 26262 specification [22].

The integration of hypervisor, operating system and run-time libraries enables the Hercules framework to provide predictable real-time guarantees for next-generation safety-critical applications, supported by a lightweight pragma-based application programming interface. Widely-adopted programming models for heterogenous architectures, such as OpenMP and OpenACC, will be extended with real-time semantic constructs. The ultimate goal of the project is to enable parallel, non real-time and hard/soft real-time workloads to run side-by-side on the same platform, while preserving the required timing guarantees of safety-critical applications with different performance requirements.

References

[1] Laukkonen, J, Drowsiness Detection: Waking Up Fatigued Drivers , LifeWire blog.
URL https://www.lifewire.com/drivers-alter-systems-534806

[2] Golgowski, N, Self-Driving Car Takes Man To Hospital After He Suffers Pulmonary Embolism, The Huffington Post.
URL http://www.huffingtonpost.com/entry/tesla-drives-man-to-hospital_us_57a8aee8e4b0b770b1a38886

[3] D. Carlino, S. D. Boyles, P. Stone, Auction-based autonomous intersection management, in: 16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013), IEEE, 2013, pp. 529–534.

[4] D. K. Murthy, A. Masrur, Braking in close following platoons: The law of the weakest, in: 2016 Euromicro Conference on Digital System Design (DSD), 2016, pp. 613–620. doi:10.1109/DSD.2016.78.

[5] Fred Lambert, Uber and Mercedes agreement.
URL https://electrek.co/2016/03/18/uber-order-mercedes-100000-autonomous-cars/

[6] Anand S. Rao and Mehrad Ahari, PwC Impact of Car Sharing, Automated Driver Assistance, Autonomous Cars on Insurance.
URL https://www.pwc.com/ca/en/insurance/publications/pwc-impact-of-driverless-cars-2015-12-en.pdf

[7] Limer, E, Uber's first self-driving car hits the street, Road And Track blog.
URL http://www.roadandtrack.com/new-cars/car-technology/a29229/uber-first-self-driving-car/

[8] Meola, A, Automotive Industry Trends: IoT Connected Smart Cars and Vehicles, The Business Insider.
URL http://uk.businessinsider.com/internet-of-things-connected-smart-cars-2016-10?r=US&IR=T

[9] Davidson, L, How connected cars are driving the Internet of Things, The Telegraph.
URL http://www.telegraph.co.uk/finance/newsbysector/industry/engineering/11372205/How-connected-cars-are-driving-the-Internet-of-Things.html

[10] 2025 AD, Japans Olympic dream: driverless cars on the road for 2020, 2025 AD blog.
URL https://www.2025ad.com/in-the-news/blog/japan-driverless-cars-in-2020/

[11] The Boston Consulting Group, The Autonomous Vehicle: The Car of the Future.
URL http://www.bcg.com/expertise/industries/automotive/autonomous-vehicle-car-future.aspx

[12] Google, inc., Google Self-Driving Car Project.
URL https://www.google.com/selfdrivingcar/

[13] Tesla, Tesla's self driving cars.
URL https://www.tesla.com/blog/all-tesla-cars-being-produced-now-have-full-self-driving-hardware

[14] O' Kane, S and Goode, L, George Hotz is giving away the code behind his self-driving car project, The Verge webzine.
URL http://www.theverge.com/2016/11/30/13779336/comma-ai-autopilot-canceled-autonomous-car-software-free

[15] A. Broggi, Automatic vehicle guidance: the experience of the argo autonomous vehicle (1999).

[16] The Guardian, Germany ready to test self-driving cars on the road and remove legal barriers , The Guardian.
URL https://www.theguardian.com/world/2016/apr/12/germany-self-driving-cars-angela-merkel

[17] O' Kane, S, California gives Nvidia the go-ahead to test self-driving cars on public roads, The Verge webzine.
URL http://www.theverge.com/2016/12/9/13902704/california-dmv-permit-nvidia-autonomous-car-testing

[18] IHS, ADAS Current and Future Perspectives.
URL https://www.ihs.com/pdf/IHS-ADAS-Current-and-Future-Perspectives_227834110913052332.pdf

[19] Muoio, D, Toyota exec: 'We are not even close' to fully self-driving cars, The Business Insider.
URL http://uk.businessinsider.com/toyota-gill-pratt-unveils-self-driving-plans-concept-car-at-ces-2017-1?r=US&IR=T

[20] NVIDIA, The Tegra X1 Platform (2015).
URL http://www.nvidia.com/object/tegra-x1-processor.html

[21] Xilinx, Inc., , The Xlinx Ultrascale Architecture.
URL http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[22] I. O. for Standardization / Technical Committee 22 (ISO/TC 22), ISO/DIS 26262-1 - Road vehicles Functional safety, Tech. rep., International Organization for Standardization / Technical Committee 22 (ISO/TC 22), Geneva, Switzerland (Jul. 2009).

[23] The Hercules Consortium, Hercules – High-Performance Real-time Architectures for Low-Power Embedded Systems.
URL http://hercules2020.eu/

[24] ARM Ltd., The Big.LITTLE architecture.
URL https://www.arm.com/products/processors/technologies/biglittleprocessing.php

[25] NVIDIA, Drive PX2 – The AI Car Computer for Self-Driving Vehicles.
URL http://www.nvidia.com/object/drive-px.html

[26] Gavin Kistner, Developing Next Generation Human Machine Interfaces (HMI).
URL http://on-demand.gputechconf.com/siggraph/2013/presentation/SG3110-Developing-Generation-Human-Machine-Interfaces.pdf

[27] The AUTOSAR Consortium, AUTomotive Open System ARchitecture.
URL http://www.autosar.org/index.php

[28] Evidence srl, ERIKA Enterprise RTOS.
URL http://erika.tuxfamily.org

[29] Infineon Technologies AG, Tricore AURIX Family.
URL http://www.infineon.com/cms/en/product/channel.html?channel=db3a30433727a44301372b2eefbb48d9&ic=0101033

[30] NVIDIA Corporation, NVIDIA CUDA Compute Unified Device Architecture Programming Guide, NVIDIA Corporation, 2007.

[31] Kronos Group, The OpenCL 1.1 Specifications (2010).
URL http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[32] OpenMP Application Program Interface v4 (2011).
URL http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[33] OpenAcc Specifications 2.5 (2015). [link].
URL http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf

[34] OpenMP Application Program Interface v.3.1 (2009).
URL http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[35] FSF - The GNU Project, GOMP - An OpenMP implementation for GCC.
URL http://gcc.gnu.org/projects/gomp/

[36] L. M. Pinho, V. Nélis, P. M. Yomsi, E. Quiñones, M. Bertogna, P. Burgio, A. Marongiu, C. Scordino, P. Gai, M. Ramponi, M. Mardiak, P-SOCRATES: A parallel software framework for time-critical many-core systems, Microprocessors and Microsystems - Embedded Hardware Design 39 (8) (2015) 1190–1203. doi:10.1016/j.micpro.2015.06.004.
URL http://dx.doi.org/10.1016/j.micpro.2015.06.004

[37] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, C. Lu, A real-time scheduling service for parallel tasks, in: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th, 2013, pp. 261–272. doi:10.1109/RTAS.2013.6531098.

[38] The Khronos Group. [link].
URL http://www.khronos.org/

[39] P. Gai, E. Bini, G. Lipari, M. D. Natale, L. Abeni, Architecture for a portable open source real time kernel environment, in: In Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial, 2000, pp. 866–875.

[40] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, I. Lee, Real-time multi-core virtual machine scheduling in xen, in: Proceedings of the 14th International Conference on Embedded Software, EMSOFT '14, ACM, New York, NY, USA, 2014, pp. 27:1–27:10. doi:10.1145/2656045.2656066.
URL http://doi.acm.org/10.1145/2656045.2656066

[41] Siemens, The Jailhouse Hypervisor.
URL https://github.com/siemens/jailhouse

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

[43] Evidence Srl, Jailhouse for TX1.
URL https://github.com/evidence/linux-jailhouse-tx1

[44] I. Sañudo, R. Cavicchioli, N. Capodieci, P. Valente, M. Bertogna, A survey on shared disk i/o management in virtualized environments under real time constraints, ACM SIGBED Review, accepted, to appear.

[45] P. Valente, M. Andreolini, Improving application responsiveness with the BFQ disk I/O scheduler, in: Proceedings of the 5th Annual International Systems and Storage Conference, ACM, 2012, p. 6.

[46] NVIDIA, NVIDIA GRID Remote Workstation Certifications.
URL http://www.nvidia.com/content/cloud-computing/pdf/GRID-certification-microsite-v2.pdf

[47] Y. Suzuki, S. Kato, H. Yamada, K. Kono, GPUvm: why not virtualizing GPUs at the hypervisor?, in: 2014 USENIX Annual Technical Conference, 2014, pp. 109–120.

[48] J. Lelli, C. Scordino, L. Abeni, D. Faggioli, Deadline scheduling in the linux kernel, Software: Practice and Experience 46 (6) (2016) 821–839, spe.2335. doi:10.1002/spe.2335.
URL http://dx.doi.org/10.1002/spe.2335

[49] Linux Foundation, The Real Time Linux collaborative project.
URL https://wiki.linuxfoundation.org/realtime

[50] Linux Foundation, Automotive Grade Linux.
URL https://www.automotivelinux.org/

[51] C. Scordino, G. Lipari, A resource reservation algorithm for power-aware scheduling of periodic and aperiodic real-time tasks, IEEE Transactions on Computers 55 (12) (2006) 1509–1522. doi:10.1109/TC.2006.190.

[52] P. Burgio, A. Marongiu, P. Valente, M. Bertogna, A memory-centric approach to enable timing-predictability within embedded many-core accelerators, in: Real-Time and Embedded Systems and Technologies (RTEST), 2015 CSI Symposium on, 2015, pp. 1–8. doi:10.1109/RTEST.2015.7369851.

[53] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, R. Kegley, A predictable execution model for COTS-based embedded systems, in: Proceedings of the 17th IEEE International Real-Time and Embedded Technology and Applications Symposium, RTAS '11, 2011, pp. 269–279.

[54] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms, in: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th, IEEE, 2013, pp. 55–64.

[55] ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile, a.k Edition (9 2016).
URL http://www.arm.com

[56] R. Cavicchioli, N. Capodieci, M. Bertogna, Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms,, in: Proceedings of 22nd IEEE International Conference on Emerging Technologies and Factory Automation (IEEE ETFA), 2017, p. to appear.

Paolo Burgio got a M.S. degree in Computer Engineering from the University of Bologna in 2007, and a Ph.D in Electronics Engineering jointly between the University of Bologna and the University of Southern-Brittany, in 2013. He now holds a post-doc position at University of Modena and Reggio Emilia. His main research interests are embedded many-core architectures and programming models, heterogeneous architectures, HLS, virtual platforms, and real-time systems. He is currently doing research on predictable many-core architectures for next-generation real-time systems.

Marko Bertogna is Associate at the University of Modena, Italy. He previously was Assistant Professor at the Scuola Superiore Sant'Anna of Pisa, Italy, where he also received (cum laude) a Ph.D. in Computer Engineering. He has authored over 60 papers in international conferences and journals in the field of real-time and multiprocessor systems, receiving six Best Paper Awards and one Best Dissertation Award. He served in the Program Committees of the major international conferences on real-time systems.

Nicola Capodieci is a post-doc at the university of Modena. He got his M.Sc and Ph.D in Computer Engineering at UNIMORE, however, he did most of his research activity in different institutions such as Univ. of Groningen (NL) for his M.Sc thesis, Univ. of New Brunswick (CAN) as a research internship and Edinburgh Napier Uni. (UK) for his Ph.D Thesis. His main research interests are Bio-inspired AI, Multi-Agent and Self-Organizing Systems, but also languages, architectures and everything else related to GPUs.

Roberto got his Master Degree and Ph.D. in Computer Science. His research has been focused in Numerical Optimization and Parallelization of algorithms on multicore systems and GPU. In particular the target was mainly to speed up computations for Image Reconstruction algorithms by exploiting High Performance Computing systems, both traditional (MPI and OpenMp) and hybrid with accelerators (NVidia GPU).

Michal Sojka is assistant professor at Czech Technical University in Prague. His research interests include design and verification of real-time systems and communication protocols, real-time and microkernel-based operating systems and hypervisors, real-time middleware platforms, model-driven engineering, safety and security in embedded systems and robotics.

Přemyslobtained an M.S. degree in Computer Engineering from the Czech Technical University (CTU) in Prague, Czech Republic, in 2016, where he is currently working toward the Ph.D. degree in Robotics and Control Engineering. He is Research Fellow with Industrial Informatics group. Přemysl is interested in embedded devices, operating systems, real-time and safety. His other interests include algorithms, AI and communication. Přemysl is currently working on a hypervisor for multi-core systems with isolation properties.

Andrea Marongiu received the PhD degree in electronic engineering from the University of Bologna, Italy, in 2010. He currently is a postdoc researcher at ETHZ, Zurich. He also holds a postdoc position at University of Bologna. His research interests concern parallel programming model and architecture design in the single-chip multiprocessors domain, with special emphasis on compilation for heterogeneous architectures, efficient usage of on-chip memory hierarchies and SoC virtualization.

Dr. Paolo Gai CEO, graduated (cum laude) in Computer Engineering at University of Pisa in 2000 with a graduation thesis developed at the ReTiS Laboratory of the Scuola Superiore SantAnna on the development of the modular real-time kernel SHaRK. He obtained the PhD from Scuola Superiore Sant'Anna in 2004. Since 2000, he founded the ERIKA Enterprise project, an open-source RTOS which recently reached the OSEK/VDX certification, and which is currently used by various industries and universities. Since 2002 he is CEO and founder of Evidence Srl, a SME working on operating systems and code generation for Linux- and ERIKA- based industrial products in the automotive and white goods market. His research interests include development of hard real-time architectures for embedded control systems, multi-processor systems, object-oriented programming, real-time operating systems, scheduling algorithms and multimedia applications.

Claudio Scordino obtained Master Degree and PhD in Computer Engineering from the University of Pisa in 2003. In 2007 he obtained the PhD from the same university, with a thesis about power-aware real-time scheduling. His research activities include operating systems, real-time scheduling, energy saving and embedded devices. He collaborates with the Linux kernel community since 2008, having several patches integrated in the official Linux kernel. He currently works as Project Manager at Evidence Srl.

Dr. Bruno Morelli received Master Degree in Computer Engineering at University of Pisa in 2007. He is expert of the whole Linux ecosystem, from the development of Linux kernel drivers to the integration and modification of libraries and development tools. He has a good knowledge of C/C++ languages, bash scripting, ARM and x86 assembler. He works as Linux Software Engineer at Evidence Srl since 2007.