Model-Based Deployment of Mission-Critical Spacecraft Applications on Multicore Processors

J. Reinier van Kampenhout and Robert Hilbrich

Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin Germany
{j.r.van.kampenhout,robert.hilbrich}@fokus.fraunhofer.de
http://www.fokus.fraunhofer.de

Abstract. A variety of complex spacecraft applications, such as autonomous maneuvers based on image recognition, can benefit from the increased performance of multicore processors. On the other hand the redundant cores can also be used for fault-tolerance. Spacecraft missions more and more require a balanced trade-off between power, performance and reliability. Finding an optimal trade-off for each mission phase leads to new engineering challenges, especially regarding the efficient and safe deployment of software applications to hardware resources. We propose a model-based approach for the construction of software deployment schemes, and apply it to a spacecraft use case with two different mission phases to illustrate the benefits of such model-based software deployment.

Keywords: avionics, deployment, multicore, model-based engineering

1 Introduction

After being widely adopted in the server, desktop and hand held markets, multicore processors now start to appear in the domain of safety-critical and mission-critical embedded systems. At the same time an ongoing trend to integrate functionality on shared platforms becomes apparent from initiatives such as Integrated Modular Avionics (IMA) [26, 25] and AUTOSAR [2].

Similar ideas slowly start to emerge in the domain of spacecraft avionics. A holistic engineering approach to combine the two trends and exploit the full potential of multicores is required for the successful adoption of such processors. At the same time there is a significant increase in complexity that threatens the system reliability, and poses the challenge of managing this complexity to system architects, system engineers and software developers. In this work we propose a model-based systems engineering approach with software architecture deployment at its center.

In Section 2 we present the motivation for the use of multicore processors in spacecraft avionics and describe our own use case: a multicore based computing platform for space applications. Section 3 then describes the challenges that

arise by the use of multicore processors in complex space applications, and why existing engineering methods are at their limits. In Section 4 we describe our approach to model-based deployment, and present a first implementation that we apply to an example mission with two phases in Section 5.

Related Work

Recently the use of multicore processors in safety-critical domains has received much attention [11, 23, 12]. Of particular interest for these domains is the improvement of reliability by using the available parallelism [6, 17]. Real-time scheduling has been an active research domain for decades, and recently several approaches to dynamic [4] and static [7, 3] scheduling of mixed-criticality software tasks on multicores have been proposed. In the deployment process however the hardware usage and runtime scheduling are affected by the allocation and mapping of tasks [24, 9]. Therefore these should be addressed in a unified systems engineering methodology in order to arrive at an integrated result in which the key properties are balanced [21, 18]. Model-driven systems engineering offers the appropriate mechanisms to address these challenges [10]. Apart from that, *partitioning* is indispensable to guarantee the absence of interference between applications in the deployment of safety-critical systems [27, 22]. One way to achieve this which currently receives much attention is by virtualization of resources [19, 20, 13, 12].

2 Multicore Processors for Spacecraft Avionics

2.1 Space Mission Requirements

We recognize that the complexity of spacecraft applications in modern space missions has risen significantly over the last years. The widespread use of high-resolution sensors leads to the generation of large amounts of data. There is a clear trend to process these data on board in order to relieve the pressure on space-to-ground communications. In the imaging domain we see applications such as hyperspectral imaging, earth observation and synthetic-aperture radar systems. In communication, antennae arrays and broadband communication can benefit from advanced encoding algorithms. Another cluster of applications are solutions for autonomous Entry, Descent and Landing (EDL) or docking maneuvers in space. These can be based on advanced imaging processing algorithms. All these applications need a significant amount of computational power.

State-of-the-art systems often use highly specialized FPGA or DSP solutions. These are costly and limited to a relatively low functional complexity, and cannot be reconfigured easily once in flight. Such benefits are only offered by software-defined systems, which can execute complex dissimilar workloads and offer a flexibility that can be used to balance power, performance and fault tolerance. While specialized single-core processors are currently used in spacecraft avionics, their performance and energy consumption lags behind that of commercially available processors. Especially *multicore* processors provide an increase

in processing power by exploiting the parallelism that is inherent to many applications, while combining energy-efficient semiconductor technologies, moderate clock rates and advanced power management functions. Such processors provide a major increase in versatility besides power and performance benefits. In the following sections we focus on the benefits and challenges of the use of multicore processors for space applications.

2.2 The MUSE Platform

As an answer to the increasing demand for computing power in space applications, we developed a satellite payload computer that builds on COTS multicore technology in the context of the MUSE project [5]. We employed the Freescale QorIQ P4080 processor, which features eight PowerPC e500 cores operating at 1.5 GHz. An additional advantage is the use of power saving silicon-on-insulator technology, which is less susceptible to single-event upsets that are due to the background radiation in space applications. Especially when using COTS components, the high reliability and availability demands for space computing can only be guaranteed when sufficient redundancy is provided on the system level.

The MUSE platform consists of two P4080 nodes on separate boards, connected by an additional I/O board that also contains the logic and connectors to other instruments. Thus a dual redundant symmetric 1-out-of-2 setup with several high-speed I/O channels is achieved (see Figure 1).

Each processing board furthermore contains DDR3 memory, radiation tolerant flash memory, and an FPGA that together with its symmetrical counterpart controls the worker/monitor configuration of the boards. The two FPGA's are connected to the processors over the Enhanced Local Bus (ELB) and implement watchdog functionality to monitor the status of the processor, as well as a synchronizing voter to generate reliable control outputs. Because the FPGA's are radiation tolerant and their logic is triple-mode redundant, the implementation as a whole is radiation-hardened. All large memory areas such as the DRAM, L3 and L2 caches are protected by error correction logic. Thus a design with multiple complementary mechanisms for fault-tolerance is achieved, with the ultimate safety net of a node switch where the monitor becomes worker if a fault in the latter can not be handled locally.

We implemented the software for the mission that was previously discussed in Section 2.1, namely that of autonomous EDL, which builds on an advanced image processing algorithm called *MoonDetect* [5]. Using a pre-loaded map of the moon's surface, the landing zone is recognized with help of one or more high-resolution cameras. The application was implemented on top of a standard embedded operating system and uses the redundant cores of the multi-core processor not only to maximize computing power, but also to increase fault tolerance by the redundant execution of critical functions.

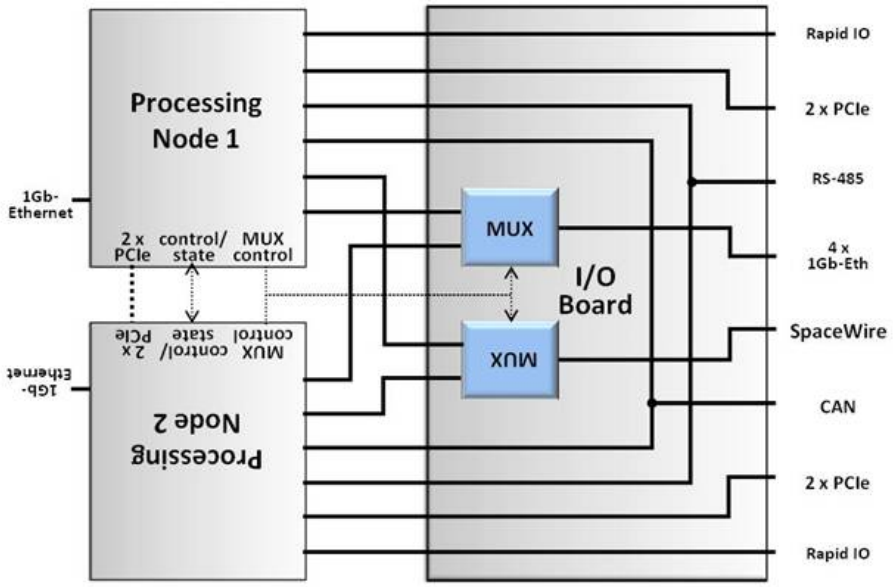


Fig. 1. The architecture of the MUSE platform with one I/O board and two redundant processing nodes.

3 Emerging Engineering Challenges

3.1 Dynamic Mission Requirements

The increasing functional requirements that we described in Section 2.1 translate to novel and more complex resource requirements for space missions. Mission requirements especially become more dynamic, there are three main drivers for this.

Firstly, missions become more and more complex and thus include multiple applications with different criticality levels that each have unique requirements regarding power, performance and reliability. The engineering process must ensure that the tasks of each application are mapped and scheduled so that there is no unintended interference during simultaneous execution. We will elaborate on this in Section 3.2.

Secondly, deep-space and planet exploration missions usually consist of different phases. To execute multiple functions with different criticality levels in each phase of the mission, the deployment of software-based platforms must be *dynamically reconfigured*.

Thirdly, support for online software updates is desirable for both maintenance and configuration changes caused by a revision of the mission targets. This again requires dynamic reconfiguration, we focus on this subject in Section 3.3.

3.2 Multi-Function Integration Within a Mission Phase

The requirements for the spacecraft missions that we previously described have a complexity that requires an implementation based on multiple applications. In our example, an image recognition algorithm is executed in parallel with a camera driver and a control task that monitors the system status. As these applications have different requirements, we are dealing with *mixed criticality* software for which the use of time and space partitioning is indispensable, as described in [29]. This publication also mentions a reduced integration effort, hardware savings, fault containment and increased security between applications as additional advantages of a partitioning approach. Partitioning in time assures that the execution of a software application in one partition does not affect timely behaviour of a software application in another partition.

Partitioning in space on the other hand protects the data and private devices of partitions by exclusively allocating memory and other hardware resources such as busses and I/O interfaces. Thus partitioning isolates applications from each other. This functionality is usually provided by the operating system [27]. Correct partitioning can however not be achieved with mere isolation, as safety requirements such as redundancy also impose constraints on the deployment. A deployment only based on safety requirements however would lead to inefficient resource usage, because simultaneous access to shared resources will be blocked by the isolation mechanism. Therefore we propose a method for optimizing the deployment according to the changing resource requirements.

3.3 Dynamic Reconfiguration between Multiple Mission Phases

A spacecraft mission comprises multiple phases which are executed over time during the mission. Each phase has different requirements so that the resource utilization trade-off between power, performance and reliability may need to be re-balanced at run-time. Dynamic reconfiguration allows to exploit the flexibility of multicore processors to switch from one phase to another. These transitions require a change and perhaps transformation of partitions, which is not accounted for in classical time and space partitioning. Thus an extension of these concepts is required in order to deal with data integrity and security, and to minimize and control the disruption of service. The phase-based ("mode-based") partitioning that we proposed in [16] is a first step in this direction. A reconfiguration mechanism furthermore simplifies maintenance and allows online software updates.

We conclude that the versatility of multicore processors can be exploited with existing methods such as partitioning, but enhancements are necessary to deal with the extended spatial dimension and to better support varying requirements of multiple mission phases as well as online updates.

4 Model-based Software Deployment

Software deployment refers to the assignment of hardware resources, such as CPU time, memory, and I/O access to software tasks. This assignment process

results in a *deployment scheme*. We distinguish between spatial and temporal deployment. The former leads to the mapping of a task to a set of hardware resources, such as a processor or electronic control unit. The latter concerns the execution schedule of each hardware resource in time. A deployment is correct if the proper type and amount of resources is assigned to all tasks at the right time. The operating system must ensure that the temporal deployment is properly executed at runtime.

There are two prerequisites for successful deployment: software must be *distributable* and *isolatable*. The trend towards function-oriented development ensures the dependencies between software and hardware are dissolving. This is enabled by standardized interfaces such as described in the ARINC 653 Apex [1] and integrated modular avionics (IMA) [26, 25] for the avionics domain, and in AUTOSAR [2] for the automotive domain. The ability to distribute tasks on its own however is not sufficient for building cost-effective mission-critical embedded systems, this capability must be complemented with time and space isolation. There are commercially available hypervisors and tools that provide such isolation on modern multicore architectures.

The correctness of a software deployment scheme has a direct impact on the extra-functional requirements of embedded software, such as real-time behaviour and reliability. Furthermore the deployment scheme determines the amount of resources that are required, and thus its efficiency has a significant influence on the hardware costs for a system. Because the trend towards integrating software from different vendors on one electronic control unit increases, a deployment scheme is the central asset and synchronization point for different development teams during the system integration phase. Therefore we address these issues by exploring an automated deployment approach based on models.

4.1 Engineering Methodology

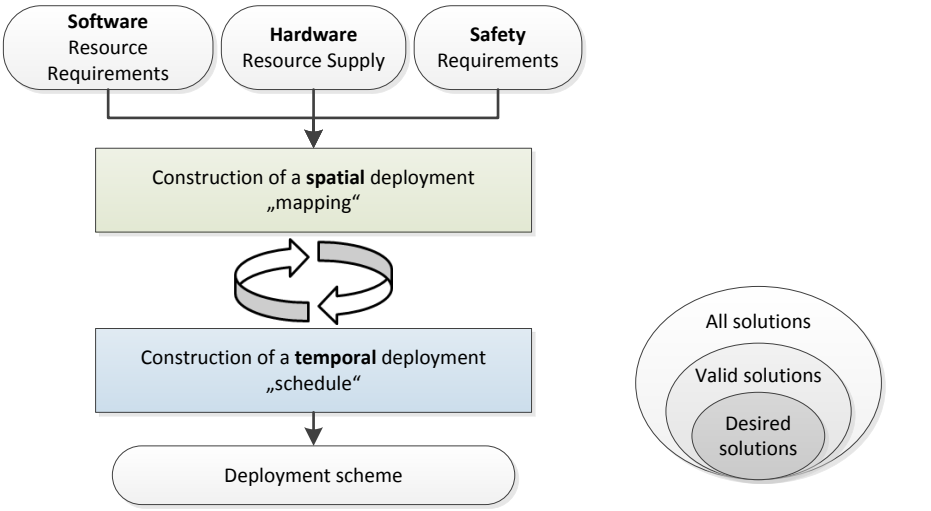
Traditional engineering methodologies for software-based systems require a high number of development iterations and prototypes, and the correctness of the system is often primarily assured by observing the system behaviour after it has been built [21]. This becomes more and more difficult as the complexity of the system and software architecture increases, and analytical approaches to assure correct timing behaviour do not suffice for modern multicore processors [28].

In our research we explore an alternative engineering methodology, which is rooted in the "Correctness by Construction" (CbyC) principle. CbyC is based on the observation that the correctness of a complex system should be argued "*in terms of the manner in which it had been produced, rather than just observing operational behaviour*" [8, 14].

We applied this approach to the construction of software deployment schemes. In order to assure correctness, we use models to capture deployment requirements as well as architectural properties and capabilities in a clear and unambiguous way. The explicit distinction between problem statement, solution construction strategy and solution proves to be beneficial to achieve optimized, re-usable architectures.

Our approach for the construction of a deployment scheme is depicted in Fig. 2a. Due to the complexity of the solution space, the spatial and temporal deployment are addressed consecutively. Therefore our methodology comprises two steps that distinctly focus on the spatial ("mapping") and temporal deployment ("scheduling"). Within each step, different types of deployment requirements and engineering constraints are addressed. Several iterations between these two steps may be required before valid a deployment scheme is obtained.

However, in order to account for the knowledge of experts, we distinguish between *valid* and *desired* solutions (see Fig. 2b). Valid solutions are all those that are correct with regard to the specified requirements. This set might be very large. The desired solutions on the other hand are valid, but also optimal with respect to certain optimization criteria provided by a domain expert. Thus we achieve solutions that are not only formally correct, but also incorporate valuable knowledge and experiences from those experts.



(a) Engineering Methodology (b) Valid and Desired Solutions

Fig. 2. Engineering methodology and types of solutions

This methodology was implemented in a prototype tool suite (see Section 5). It allows us to automatically construct a correct and optimized deployment scheme for our use-case within minutes.

4.2 Construction of Mappings

Where spatial deployment could be performed manually for a handful of single-core processors, it is not viable for complex systems with hundreds of cores spread over dozens of processors. This is especially true when we consider the need for

a safe dynamic reconfiguration to optimize the resource utilization. Besides, the mapping of safety-critical systems does not only depend on the schedulability, but also on the satisfaction of other safety-related criteria, such as redundancy, dissimilarity and independence.

Applications that belong to two redundant partitions for instance are usually not allowed to be mapped on the same core or processor, because that would violate safety requirements. Some safety critical applications contain partitions that feature a dissimilar implementation of the same functionality. Depending on the criticality level, these partitions may need to be mapped onto dissimilar processors and configured to use dissimilar communication channels to account for undetected design errors.

Our prototype tool generates all valid mappings of software components onto processor cores. It matches the resources offered by the underlying hardware architectures to the resource requirements of the applications.

Currently the following *matching criteria* are implemented:

- type of processors, cores and I/O interfaces;
- capacity of processors, cores and I/O interfaces;
- safety relations between applications (redundancy or dissimilarity);
- communication intensity between applications, in order to express proximity.

The first three criteria are used to specify *valid* spatial deployments, where the correct type, amount and the independence of resources for critical software components has to be guaranteed.

Metrics and optimization strategies for a *desired* spatial deployment on the other hand are based on criteria such as minimizing average communication distances and equal load distribution.

Note that our tool only generates solutions with significant difference. Deployment schemes which only differ in the location of identical tasks of the same parallel application are treated as being equal. This reduces the complexity of the solution space for applications with parallel tasks.

4.3 Construction of Schedules

Our model-based approach for the construction of static operating system schedules for multicores is based on the following assumptions. To achieve predictable and deterministic real-time behaviour for a system that relies only on periodical tasks with known periods, the following information (or at least an estimation) must be available at design time:

1. timing characteristics such as the worst case execution time (WCET), period and jitter, of all tasks within an application;
2. dependencies between tasks;
3. the access patterns of external resources of all applications.

A key aspect of static scheduling is that all conflicts that may appear at runtime and lead to unpredictable timing are resolved at design time. Our approach

aims to construct and optimize the schedule beforehand to avoid cumbersome timing analysis afterwards.

For this we developed a prototype scheduling tool [15]. It automatically generates a valid static schedule that satisfies the timing characteristics of a given set of applications on a given hardware architecture. As the underlying problem is NP-hard, there are many solutions and the user can adjust the generated schedules for specific needs and purposes. Our scheduler ensures that no constraints are violated during the adjustment process.

Based on constraint-solving strategies specifically tuned for the problem, our scheduler constructs a schedule that satisfies the constraints defined in the input model. This model and the problem specifications have been developed to suit the needs of the safety-critical aerospace domain. All execution times for software tasks should be based on a worst-case analysis.

The resulting schedule is input for a time-triggered OS dispatcher that executes the tasks at run-time. External resources can be separately modelled and incorporated in the schedule construction process. After being approved by the certification authority, the schedule is used in the final configuration of the operating system.

5 Use-Case: Spacecraft Software Deployment on the MUSE Platform

In this section we describe our use-case, a real mission consisting of two phases to show our approach to model-based software deployment on the fault-tolerant MUSE hardware platform.

5.1 Mission Description

We implemented one of the previously discussed missions, namely that of autonomous EDL based on images provided by a Camera Driver (CamDr). An optimal performance can be achieved if MoonDetect (MD), our image recognition application, is executed on eight cores in parallel, as depicted on the left hand side of Figure 3. Here, one instance of MoonDetect is executed, its result are saved with triple redundancy and checked for plausibility by independent tasks (Plausibility Check, PC). The ELB driver (ELBDr) collects these results and transfers them shifted in time after which they are voted in the FPGA, our radiation hardened component.

Furthermore we consider the supervisor system application, which consists of two redundant tasks in a worker/monitor configuration (SVWorker/SVmonitor). This application has a higher criticality than MoonDetect, and thus we are dealing with mixed-criticality as discussed in Section 3.2.

The main goal in this phase is to recognize the landing area from a great distance, for which high performance is necessary because of the large area that must be covered. At a resolution of 800x600 pixels we achieved a frame rate of 23 fps when MoonDetect was executed on eight cores.

In the second phase the spacecraft descends for landing, and reliability becomes more important because a fault might lead to loss of the spacecraft. The frame rate can however be reduced because the target area is already recognized and the algorithm is only used for course alterations that are performed every few seconds or so. Thus in this phase an architecture such as depicted on the right hand side of Figure 3 is more suitable. Here two cameras are used, there are two (possibly dissimilar) instances of MoonDetect on four cores each, and the data path of each instance still has triple-mode redundancy. We expect that in this case the overhead will reduce the frame rate to less than half than that of the eight-core solution, and indeed we measured 11 fps at 800x600 in the actual setup.

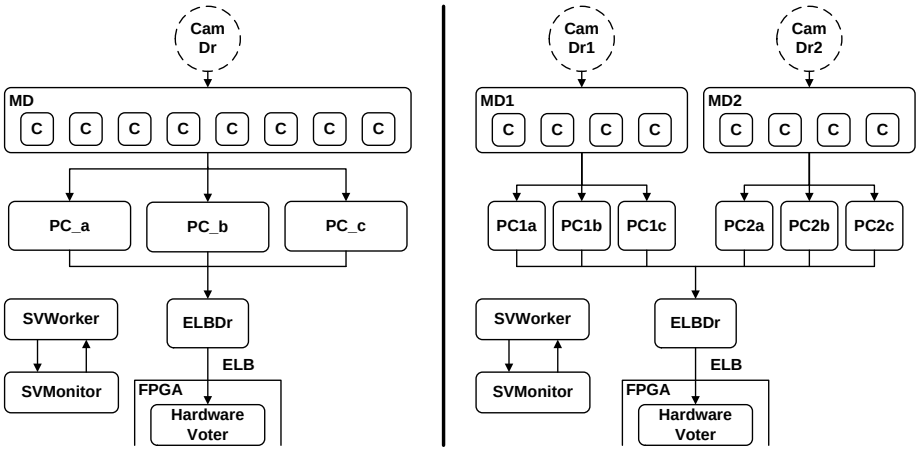


Fig. 3. Two mission phases: on the left an eight-core solution with triple redundant datapath is shown, the solution on the right features two redundant instances of the algorithm that use four cores each.

Our example mission consists of two phases. Both contain mission-critical applications and the phases must be dynamically switched over time. The mission thus contains the challenges we initially described in Section 3. In the remainder of this section we will describe our approach to automatically deploy these two phases on the MUSE hardware. As the software on both processors is identical in this mission, we will show the workflow for one processor only. Our tools however can handle multiple (distributed) processors with dissimilar workloads as well.

5.2 Spatial Deployment - "Mapping"

As described in Section 4.2, the generation of correct mappings requires modelling the resources provided by the hardware architecture and the demands of software components. In our use-case, we employ an eight core processor (see Section 2.2).

Although all cores on the P4080 are physically connected to all I/O interfaces, we imposed additional access constraints on I/O interfaces for reliability purposes. Therefore Ethernet connections are allowed on core 1 and 2 only, and ELB connections only on core 3. The resulting hardware model is contained in Listing 1.1. The computational capacity and I/O access capabilities are specified for each core within a processor.

```
Hardware {
   Processor QorIQ_P4080 {
    Core C1 {
        Capacity = 100;
        Provides IO access = Ethernet;
    };

    [...]

    Core C3 {
        Capacity = 100;
        Provides IO access = EnhancedLocalBus;
    };

    [...]
 }
}
```

Listing 1.1. Modelling the QorIQ P4080 hardware architecture

The software architecture is modelled similarly (see Listing 1.2). Applications are modelled per task, thus each "*Application*" is mapped onto a single core. An "*Application*" requires a certain amount of computational capacity, as well as access to specific I/O interfaces.

```
Software {
   Application CamDr {
      Core Utilization = 2;
      Requires IO access = Ethernet;
   };

   Application ELBDr {
      Core Utilization = 2;
      Requires IO access = EnhancedLocalBus;
   };

   Application MD_a {
      Core Utilization = 90;
   };

   Application MD_b {
      Core Utilization = 90;
   };

   [...]

   Application PC_a {
      Core Utilization = 1;
   };

   [...]
}
```

Listing 1.2. Modelling the software architecture

In addition to the matching of resource supply and demand, the correctness of a mapping depends on reliability criteria. Using *independent* resources is a significant prerequisite for robustness and reliability.

In our use-case we limit our focus to independence on the core level, which means that two redundant applications are not allowed to be deployed on the same core. These redundancy requirements are specified in Listing 1.3.

```
Relations {
    MD_a, MD_b, MD_c, MD_d, MD_e, MD_f, MD_g, MD_h redundant;
    SVWorker, SVMonitor, CamDr, ELBDr redundant;
    PC_a, PC_b, PC_c redundant;
}
```

Listing 1.3. Modelling reliability relations for spatial deployment (phase 1)

With this modelling approach we generated mappings for both phases. The results for each phase are shown in Table 1. The generation process took about 1 second on a standard laptop.

Table 1. Result of the spatial deployment for phase 1 (upper half) and 2 (lower half)

Phase	Cores							
	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
1	MD_a	CamDr	ELBDr	MD_d	MD_e	MD_f	MD_g	MD_h
	PC_a	MD_b	MD_c	SVMonitor				
	SVworker	PC_b	PC_c					
2	MD_1a	MD_1b	MD_1c	MD_1d	MD_2a	MD_2b	MD_2c	MD_2d
	PC1a	PC1b	PC1c	SVMonitor	SVWorker	PC2a	PC2b	PC2c
	CamDr1	CamDr2	ELBDr					

5.3 Temporal Deployment - "Static Scheduling"

In order to initiate the scheduling process, several input parameters are required. Global parameters include the hyperperiod duration and switching time between applications (OS overhead), which we fixed at 0.5 milliseconds. Then the processor is modelled as follows:

```
def_processor
    id('1'),
    name('P4080'),
    cores(8).
```

Listing 1.4. The processor model

For the modelling of applications timing information is needed, the execution time of MoonDetect can be deduced from the frame rate. Although we measured that the camera and ELB drivers, the supervisor and the plausibility checks

have execution times in the order of microseconds, we will model them all with 1 millisecond to be able to depict them on the same scale. An application model looks as follows:

```
def_application
    id(CamDr),
    processor('1'),
    cores_possible(3),
    period(500),
    duration(10).
```

Listing 1.5. The model of the CamDriver application

We see that this process is only allowed to be executed on core C_3, which is the result of the mapping process. Finally the dependencies that are needed for the scheduling must be specified. They are implied by Figure 3. Such dependencies may be specified in the following way.

```
def_after
    id1(CamDr),
    id2(moondetect),
    distance(0,10).
```

Listing 1.6. Specifying a dependency

Figure 4 shows the schedules of one execution period of each phase. The first phase is shown in the upper eight horizontal bars, time is on the horizontal axis. We see that MoonDetect, as expected, requires the bulk of the processing power. Shorter tasks are executed in parallel to each other whenever possible.

In the second phase depicted in the lower eight bars we see that two instances of MoonDetect are executed in parallel and subsequently have a longer execution time, and that there are six instances of the plausibility check task instead of three. Because of the parallel instances of MoonDetect the frame rate deteriorates. In the first phase it is 20 fps (or a hyperperiod of 50 ms), in the second we achieve 10.4 fps (hyperperiod is 96 ms).

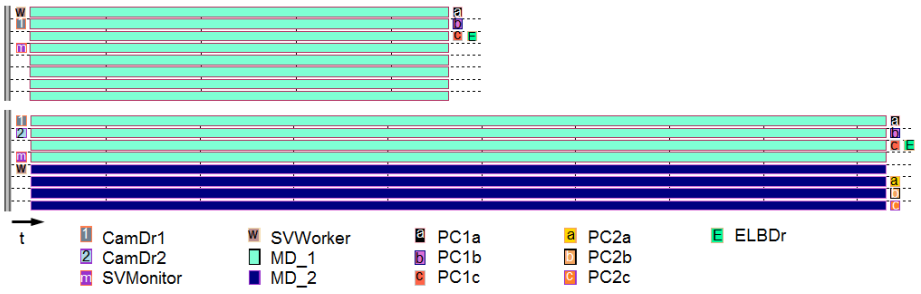


Fig. 4. The schedule of one period of phase 1 (upper eight bars) and 2 (lower eight bars). The horizontal axis represents time, on the bottom the task identifiers are indexed.

Between the two phases dynamic reconfiguration is necessary as both the mapping and scheduling change because tasks are started and migrated. Startup and migration times in our system are negligible with these execution times, and thus a phase switch can take place without noticeable disruption of service. A switch could however also be modeled as separate phase in which the startup and migration times are precisely indicated.

While our example is somewhat simplified and the final deployment scheme could have been produced manually, the benefit of using a model-based approach becomes especially apparent when more tasks and more resources have to be incorporated. With the advent of manycore processors comes the likelihood that the system complexity will significantly rise in the near future. At this point, manual processing of problems with such orders of magnitudes is costly, error prone and inefficient. Our automated approach completes in mere seconds, its result is *correct by construction* and a new iteration can be started with one click. Furthermore, optimization criteria can be easily modified so that existing models can be efficiently re-used.

6 Conclusions

In this paper we discussed the advantages of using multicore processors in spacecraft avionics, and recognized that many missions can benefit from the increased computing power. We introduced our MUSE hardware platform that comprises two eight-core processors and focused on the engineering challenges that come with increasingly diverse mission requirements. Multicore processors seem a future-proof platform for space applications as they offer the flexibility to support these requirements. The push towards multi-function integration leads to new challenges that are created by applications with different criticality levels.

Safe integration requires the operating system to offer time and space partitioning extended to support multicores, in concert with correct deployment. Comprehensive mission requirements result in multiple mission phases and corresponding mode-switches that must be accounted for. Such dynamic reconfiguration furthermore allows online software updates.

Exploiting the potential of multicores in spacecraft avionics requires an engineering method that efficiently builds optimized system configurations for each phase of a mission. The challenge to profit from the available flexibility becomes apparent in the software deployment. A correct deployment of software components relies on spatial and temporal resource assignment. We presented a model-based engineering method that automates the construction of deployments. It facilitates the efficient design of complex software architectures and has a result that is “Correct by Construction”.

Finally we presented an example mission consisting of two phases which contain mixed-criticality applications. We deployed these phases onto the MUSE hardware platform and showed the advantages of our engineering approach, namely that, once a phase is modelled, a correct deployment scheme can be generated almost instantly. This allows quick feedback and easy balancing of

parameters in the development process. In concert with the operating system, isolated execution of mission-critical applications along with non-critical applications and can be handled. Multiple deployment schemes allow multi-phase missions provided that mode-switches are well-defined.

References

1. ARINC. ARINC Specification 653P1-2: Avionics Application Software Standard Interface Part 1 - Required Services. Technical report, Aeronautical Radio Inc., Maryland, USA, Dec. 2005.
2. AUTOSAR. Layered Software Architecture. `http://autosar.org/download/R4.0/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf`, Nov 2010.
3. J. Baro, F. Boniol, M. Cordovilla, E. Noulard, and C. Pagetti. Off-line (Optimal) multiprocessor scheduling of dependent periodic tasks. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1815–1820, New York, NY, USA, 2012. ACM.
4. S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling Real-Time Mixed-Criticality Jobs. *Computers, IEEE Transactions on*, 61(8):1140 –1152, aug. 2012.
5. P. M. Behr, I. Haulsen, J. R. van Kampenhout, and S. Pletner. Multi-Core Technology for Fault Tolerant High-Performance Spacecraft Computer Systems. In *DASIA*, 2012.
6. S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10 – 16, nov. 2005.
7. V. Brocaly, M. Masmanoy, I. Ripolly, A. Crespoy, P. Balbastrey, and J.-J. Metge. Xoncrete: a scheduling tool for partitioned real-time systems. In *Proceedings of the Embedded Real Time Software and Systems Conference (ERTS2 2010)*, May 2010.
8. R. Chapman. Correctness by construction: a manifesto for high integrity software. In *Proceedings of the 10th Australian workshop on Safety critical systems and software - Volume 55*, SCS '05, pages 43–46, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.
9. C.-L. Chou and R. Marculescu. User-Aware Dynamic Task Allocation in Networks-on-Chip. In *2008 Design, Automation and Test in Europe*, pages 1232–1237. IEEE, März 2008.
10. G. Deng, D. C. Schmidt, and A. Gokhale. Addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems via aspect-oriented & model-driven software development. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 811–814, New York, NY, USA, 2006. ACM.
11. R. Fuchsen. How to address certification for multi-core based IMA platforms: Current status and potential solutions. In *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, pages 5.E.3–1 –5.E.3–11, oct. 2010.
12. T. Gaska, B. Werner, and D. Flagg. Applying virtualization to avionics systems #x2014; The integration challenges. In *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, pages 5.E.1–1 –5.E.1–19, 2010.
13. G. Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems - IIES '08*, pages 11–16, New York, New York, USA, 2008. ACM Press.

14. R. Hilbrich. How to Safely Integrate Multiple Applications on Embedded Many-Core Systems by Applying the "Correctness by Construction" Principle. *Advances in Software Engineering*, 2012(354274):14, 2012.
15. R. Hilbrich and H.-J. Goltz. Model-based generation of static schedules for safety critical multi-core systems in the avionics domain. In *Proceeding of the 4th international workshop on Multicore software engineering*, IWMSE '11, pages 9–16, New York, NY, USA, 2011. ACM.
16. R. Hilbrich and J. R. van Kampenhout. Partitioning and Task Transfer on NoC-based Many-Core Processors in the Avionics Domain. In *4. Workshop: Entwicklung zuverlssiger Software-Systeme (Stuttgart, Deutschland) and Journal Softwaretechniktrends*, 2011.
17. A. Jacobs, G. Cieslewski, and A. George. Adaptive Software-based Fault Tolerance for Space Multicore Processing. In *Workshop for Multicore Processors For Space - Opportunities and Challenges, IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, 2009.
18. M. Jamshidi. *System of systems engineering: innovations for the 21st century.* Wiley series in systems engineering and management. Wiley, 2009.
19. R. Kaiser. Alternatives for scheduling virtual machines in real-time embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, IIES '08, pages 5–10, New York, NY, USA, 2008. ACM.
20. R. Kaiser. *Virtualisierung von Mehrprozessorsystemen mit Echtzeitanwendungen.* Dissertation, Universitt Koblenz-Landau, June 2009.
21. A. Kossiakoff, W. Sweet, S. Seymour, and S. Biemer. *Systems Engineering Principles and Practice.* Wiley Series in Systems Engineering and Management. Wiley, 2011.
22. P. N. Leroux and K. Johnson. Using Resource Partitioning to Build Secure , Survivable Embedded Systems. Technical report, QNX Software Systems, 2009.
23. J. Nowotsch and M. Paulitsch. Leveraging Multi-Core Computing Architectures in Avionics. In *2012 European Dependable Computing Conference*, 2012.
24. C. Pinello, L. Carloni, and A. Sangiovanni-Vincentelli. Fault-Tolerant Distributed Deployment of Embedded Control Software. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(5):906 –919, may 2008.
25. P. J. Prisaznuk. ARINC 653 role in Integrated Modular Avionics (IMA). In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1.E.5–1 – 1.E.5–10. IEEE, 2008.
26. RTCA. Integrated Modular Architecture – Development Guidance and Certification Considerations, 2005.
27. J. Rushby. Partitioning for Safety and Security: Requirements, Mechanisms, and Assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999.
28. R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966 –978, july 2009.
29. J. Windsor and K. Hjortnaes. Time and Space Partitioning in Spacecraft Avionics. In *Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference on*, pages 13 –20, july 2009.