# An Operating System for Safety-Critical Applications on Manycore Processors

Florian Kluge, Mike Gerdes, Theo Ungerer
Department of Computer Science
University of Augsburg
86159 Augsburg,Germany
{kluge,gerdes,ungerer}@informatik.uni-augsburg.de

***Abstract*—Processor technology is advancing from bus-based multicores to network-on-chip-based manycores, posing new challenges for operating system design. In this paper, we discuss why future safety-critical systems can profit from such new architectures. To make the potentials of manycore processors usable in safety-critical systems, we devise the operating system MOSSCA that is adapted to the special requirements prevailing in this domain. MOSSCA introduces abstractions that support an application developer in his work of writing safety-critical applications. Internally, MOSSCA runs in a distributed manner to achieve a high parallelism while still guaranteeing a predictable behaviour.**

## I. Introduction

For some years now, multicore processors have been entering the domain of real-time embedded systems (RTES). Initially, there were reservations against the use of multicore processors in safety-critical systems [1]. In the meantime, considerable work has been spent to overcome these reservations by appropriate hardware [2]–[5] and software design [6]–[8]. Several multicore- and real-time-capable operating systems are commercially available, e.g. the Sysgo PikeOS [9] or VxWorks [10] from Wind River. Meanwhile, the number of cores in single chips is increasing as processor technology advances. Several processors that integrate over 32 cores on one die are available [11]–[15] or are being developed [16]. On such processors, the single cores are no longer connected by a shared bus. Instead, a *Network on Chip (NoC)* is used for passing messages between the cores of such *manycore processors*. Each core possesses local memories or caches. Accesses to a remote global shared memory are possible, but incur high latencies. This fundamental change in processor architecture poses new challenges for application and OS design. In the domains of general-purpose, high performance and cloud computing considerable work is being spent on operating systems for future manycore processors [17]–[21]. eSOL announced a commercial real-time operating system for the TILE-Gx8036 manycore processor [12] for 2014 [22].

We developed a Manycore Operating System for Safety-Critical Application domains called MOSSCA [23]. With this work we investigated how the requirements of safety-critical embedded applications can be implemented in future manycore processors, and how applications can benefit from such architectures. While a performance gain seems obvious, also the implementation of special requirements of safety-criticality domains will be eased. Hitherto, much effort is being spent in guaranteeing isolation of different applications executed on the same computer. The difficulty of this aim lies in the versatile sharing of resources that is inherent in today's computers. Manycore architectures allow us to reduce the problem of resource sharing to the NoC interconnect and off-chip I/O.

In this paper, we introduce the abstractions that MOSSCA provides to application developers, namely *nodes, channels,* and *servers*. In the design and implementation of these abstractions, we especially address the special needs of the safety-criticality domain. The MOSSCA abstractions enable composability, i.e. that components developed by several teams can be integrated in one computer without violating overall system requirements. Furthermore, we present how we integrate these abstraction within the MOSSCA system architecture, and discuss the principles that underlie their implementation.

In the following section, we review the requirements of safety-critical applications. In section III we introduce the abstractions that MOSSCA provides to an application developer. The overall architecture of a MOSSCA-based system is presented in section IV. In section V we discuss the principles underlying the implementation of MOSSCA. We compare the MOSSCA approach to related work in section VI. Section VII concludes this paper.

## II. Safety-Critical Systems

A system is considered as being *safety-critical (SC)* if its failure can lead to loss of life or severe damage of property or the environment. Safety-critical systems appear in various domains, like transport, medicine, nuclear power plants etc. Typically, an SC computer executes multiple applications of different criticality levels. Any unpredictable interferences between these applications must be excluded to ensure the SC properties of such a system. One approach to achieve this would be to certify all applications to the highest criticality level of the computer. However, this approach is very complex and costly. Instead, *partitioning* [24] is used to ensure *freedom of interference*. Any application is certified to its own criticality level and then is assigned its own partition. The

underlying OS or hypervisor schedules the partitions and ensures isolation between the partitions as well as on accesses to shared resources. Communication between partitions can only be performed through an interface provided by the OS, which is usually based on messages. Timely interferences and error propagation over partition boundaries are thus prevented. In general, applications consist of multiple processes or threads. These can interact through a number of services provided by the OS operating on application-local objects. In some OS implementations, the threads execute in a shared address space and hence can also directly communicate via memory.

### A. Multicore Processors in SC Systems

While widely used in the domain of general-purpose computing, multicore processors are only slowly entering the domain of safety-critical systems. In 2009, Kinnan [1] identified several issues that are preventing a wide use of multicore processors in safety-critical RTES. Most of these issues relate to the certification of shared resources like caches, peripherals or memory controllers in a multicore processor. Wilhelm et al. [2] show how to circumvent the certification issues of shared resources through a diligent hardware design. Later work also shows that even if an architecture is very complex, smart configuration can still allow a feasible timing analysis [3].

The problems discussed above stem mostly from the fine-grained sharing of many hardware resources in today's multicore processors. On the single-core processors used in today's SC systems, the partitions executed on one computer share one core. Even with multicore computers, several partitions would have to share one core and various common resources. With an increasing number of cores, it would be possible to assign each partition its own core or even a set of cores exclusively. We are convinced that future SC systems can greatly benefit from manycore architectures. The only shared resources are the NoC interconnect and off-chip I/O. In our work, we aim to develop a system and OS architecture that can provides safety-criticality isolation properties on such processor and make the processor usable for future SC applications.

### B. Requirements

Although the underlying hardware architecture may vary, an operating system for SC systems in general has to provide the following properties:

1) The whole system must behave *predictably* and must therefore be *analysable*. This includes a *predictable timing behaviour* to ensure that all deadlines are kept.
2) *Partitioning* in time and space guarantees freedom of interference. Special care must be taken to make accesses to shared resources predictable.
3) *Fine-grained communication* takes only place between threads of the same application. If applications have to exchange data over partition boundaries, *special mechanisms* are provided by the OS.

Furthermore, there is ongoing research on the dynamic reconfiguration of software in safety critical systems [25]. Although this is not part of today's standards, we view the capability for reconfiguration also as an important requirement for future SC systems, leading to another requirement:

4. The SC computer must be enabled to be *reconfigured* during runtime, and the reconfiguration must be performed in a predictable manner.

In the following sections we describe how we build an operating system for a manycore processor that fulfils these requirements.

## III. MOSSCA Abstractions

For MOSSCA we assume that enough nodes are available to execute each application thread on a separate node. Thus we remove any mutual influences between partitions that stem from pure thread execution, and even interferences between threads in the same partition are reduced. Threads belonging to the same partition are grouped together by mapping them on adjacent cores. Communication between threads in one partition thus mostly takes place on the part of the NoC that is covered by this partition. It is strictly based on passing messages between the threads. The space and time partitioning thus is partly provided by the underlying hardware. This approach results in a mapping like depicted in figure 1.

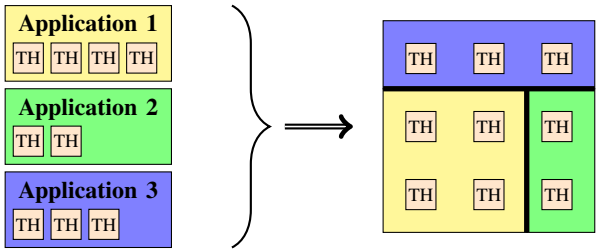


Figure 1. Mapping of threads (TH) and partitions to a manycore processor

To make these properties usable by the application developer, MOSSCA provides three abstraction concepts, namely *nodes*, *communication channels*, and *servers*. In this section, we present the considerations underlying these abstractions.

### A. Nodes

Nodes represent the primary execution resources for applications. Each MOSSCA node maps directly to a physical node of the manycore processor. The presence of functional units in the physical node is represented by *node capabilities*. Such capabilities can, e.g. stand for timer units or caches. Another kind of capability are I/O pins that are directly connected to the node. A MOSSCA node acts as container for the binary image of one application thread and executes the program defined in this image, i.e. one node executes one thread. If the thread uses only node-local resources it is executed in full isolation from the rest of the processor and thus is not subject to any interferences from other threads. An initial configuration of the physical nodes is performed by MOSSCA during the boot process, where predefined application threads are loaded to the nodes.

### B. Communication Channels

*Communication channels* represent the basic means for interaction between application threads. A communication channel in MOSSCA provides unidirectional communication between a sender *(channel source)* and a receiver node *(channel sink)*. The channel provides certain properties and constraints to its source and sink nodes. These properties are defined in the *channel policy*. The channel implementation ensures that an application cannot exceed the limits defined in the channel policy. Thus, possible overload of the NoC and the channel sink is prevented. If an application thread is trying to exceed the limits stated in the channel policies, the messages are blocked by the kernel, as the application is behaving outside its specification. Thus we ensure that faults in the application do not propagate through the system.

Channels can be used to implement more sophisticated communication patterns on top. Two-way communication is achieved by coupling two channels with opposite directions. With the help of servers (see next section), it is also possibly to implement distributed buffers or blackboards, or other forms of group communication e.g. for inter-partition communication.

### C. Servers

MOSSCA servers provide services that are used by multiple applications or threads but need to be executed in a centralised manner. An obvious example is an I/O server that multiplexes I/O requests from several threads on one I/O device, and works as the primary interrupt handler for this device. Similarly, certain OS services that may afflict other threads need a central instance that has a global view of the system and thus is able to ensure isolation properties. Servers are also used to coordinate inter-partition communication. Finally, applications may define servers that provide library services for computations used in several threads. If these are centralised, memory is saved in the nodes that execute the application threads. Similar to communication channels, MOSSCA servers must provide a predictable behaviour to their clients. This means that they must guarantee a maximum reaction latency that is defined in the *server policy*. The server policy also implicitly controls the actual communication between server and client application over the NoC. MOSSCA servers implement non-interruptable transaction. Once a server has started processing an application request, it will finish this request without interruption by other application requests.

### D. Interface

Figure 2 illustrates a developer's view of a MOSSCA system and how the abstractions introduced above are used. The figure shows a MOSSCA system consisting of two applications and two servers. Application A consists of three threads A0, A1, A2, each executing on its own node N0, N1, N2. Application B has two threads B0, B1 executing on nodes N4, N5. Each server is also assigned a separate node, whereby the I/O server gets node N6 that possesses additional I/O pins. Using MOSSCA channels, the threads of one application can exchange data. Communication between the application partitions is handled by an inter-partition communication server (IPCS) running on node N3. The connection between application threads and servers is not mapped by (user-level) channels. From application view, only the server policy is relevant for the usage of the server.

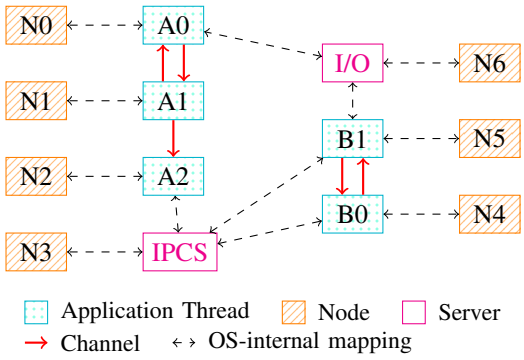


Figure 2. The developer's view on a MOSSCA system

For each abstraction concept, the MOSSCA API defines functions for de-/allocation and functions that are used during operation. The allocation functions require a check whether the resource is actually available. Concerning safety-critical applications, these checks must be performed during system integration to ensure that the deployed system will work correctly. Insofar, the required resources are allocated statically. If reconfiguration of safety-critical applications is envisaged, valid alternative configurations must also be computed offline. Only less critical applications may use the de-/allocation services more freely during runtime. However, they have to cope with possible failures of resource allocations.

### E. Discussion

With the abstractions presented above, we aim to support applications developed for a manycore processor in fulfilling the safety-criticality requirements defined in section II-B. *Predictability* and *analysability* of applications is achieved by several means. Through assigning each thread its node exclusively, pure execution (i.e. without communication) will not suffer any interferences from other threads. Accesses to shared resource like servers or the physical NoC are guarded by strict policies that are ensured by the operating system. These policies are established during system integration. They take into account requirements of the application. Additionally, they are designed such that unpredictable interferences stemming from the shared nature of servers and the NoC are excluded. These means also provide the *partitioning* in space and time. For *fine-grained communication* within partitions, applications can directly use communication channels. Communication between partitions can also be implemented through channels, or by a dedicated server if more complex communication patterns are necessary (see sections IV-B and V-E). *Reconfiguration* of a running system is supported through services for de-/allocation of resources. It can be coordinated by dedicated servers. Valid system configurations for reconfiguration of

high-critical applications must be defined already during system integration. This ensures that a reconfiguration will not fail. Furthermore, this approach keeps reconfiguration latencies low, as the complex allocation and mapping calculations are performed offline.

## IV. MOSSCA Architecture

One core idea of MOSSCA is to map safety-critical application threads to separate nodes of a manycore processor. In this mapping, each partition is represented by a cluster of nodes. Additionally, MOSSCA allocates separate nodes as servers to perform tasks that must be executed in a centralized manner. These include off-chip I/O (for all partitions) and inter-partition communication, but also application specific services. These concepts are reflected by the architecture of a MOSSCA system as depicted in figure 3.

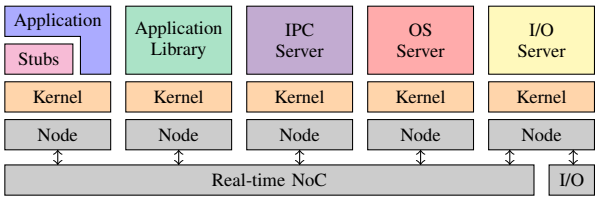


Figure 3. MOSSCA System Architecture

The hardware base is formed by *nodes*. These are connected by a *real-time interconnect* that provides predictable traversal times for messages (see e.g. [26]–[28]). Additionally, some nodes have special facilities, e.g. for off-chip I/O. OS functionalities are split into several parts to run in a distributed manner and to achieve high parallelism. An identical kernel on each node is responsible for configuration and management of the node's hard- and software. OS functionalities that require a centralised execution or need to coordinate between several nodes are provided by OS servers. Nodes possessing an off-chip connection that is used by at least one of the application threads act as I/O servers. For inter-partition communication, MOSSCA allocates dedicated communication servers (IPCS). Additionally, MOSSCA supports the implementation of application-specific servers that provide e.g. library services used by several threads. In the following sections, we discuss the properties of the single OS components and how they form an OS for safety-critical systems on manycore processors.

### A. Kernel

The kernel manages all node-local hardware devices. Concerning e.g. the interface to the NoC interconnect, this includes a fine-grained configuration of the send/receive bandwidth within the bounds that are given by the OS. The kernel ensures that the local application does not exceed these constraints. The kernel is also responsible for the software configuration of the node, i.e. loading the code of the application thread that should run on the node, including OS, I/O and other servers. To achieve these aims, the kernel can interact with an OS server.

### B. Servers

Servers in MOSSCA provide services that cannot or should not be executed on every node. Services that cannot be executed on each node typically concern management of or access to shared resources. Furthermore, if multiple application threads use the same library routine, it can be beneficial to execute this routine on only one node acting as *Application Library Server*. Thus, memory on the other nodes is saved. Depending on their functionality, servers may be used by multiple partitions.

The OS server manages all on-chip resources, and especially the mapping and scheduling of application threads to nodes. Which thread is executed on which node can either be decided by an OS server dynamically, or already during system integration to give stronger guarantees. A MOSSCA system can have several OS servers running in parallel, each managing a part of the processor. Thus, if e.g. each partition has its own OS server, interferences between partitions are kept at bay. With the help of the kernel, the OS servers manage the NoC interconnect and set up send/receive policies for all nodes. One OS server instance also coordinates the boot process of the chip and configures all other nodes by deciding which code to execute on which node. If the need for online-reconfiguration of the system arises, this task is also managed by an OS server. Applications cannot directly access an OS server. Instead, the kernel interacts with an OS server to implement certain system services.

Through the integration of many cores on one chip, only dedicated nodes will be able to access I/O hardware directly. These nodes act as I/O server for the connected device. They handle all I/O requests from the other nodes for this device. The I/O server takes care of time multiplexing the applications' output towards an I/O device.

For communication between partitions, MOSSCA can provide an *inter-partition communication server (IPCS)*. While channels provide a reasonable abstraction for fine-grained communication, more coarse-grained approaches that need further kinds of control can be implemented in such a server. We present further reasons for communication servers in section V-E.

### C. Stub Interfaces

Applications can use a generic server interface to communicate with all servers except the OS server. However, server requests have to follow a certain protocol that is defined by the server developer. To ease the use of any server, we encourage server developers to provide an additional server stub component that provides a server-specific interface and internally maps to MOSSCA's generic server interface.

### D. Discussion

The aim of the presented architecture is to introduce a structure into MOSSCA that supports the adherence of the OS implementation to the safety-criticality requirements (see sect. II-B). Basically, the architecture is a continuation of

the abstract concepts of *nodes* and *servers*. Thus, the safety-criticality support properties of these abstraction are found again in the system architecture. The mapping of each application thread and each server to a separate node continues the *partitioning* concept and thus ensures *predictability* and *analysability*. With the kernel, the MOSSCA architecture introduces a new, OS-internal abstraction. On the one hand, the kernel provides the basis for executing application threads and servers. On the other hand, based on the underlying NoC it maps the third MOSSCA abstraction, *communication channels*, into the system architecture. Furthermore, the kernel provides the basis for reconfiguration of the system in concert with an OS server. This is achieved by basing all application threads and servers on the general kernel interface. Thus, code running on top of this interface can be exchanged, even during runtime.

## V. MOSSCA Implementation

We have implemented MOSSCA on a functional simulator for a manycore processor [29]. The simulator adopts simple cores that enable a tight *worst-case execution time (WCET)* analysis. Each core on the processor possesses local scratchpad memories for code and data, thus abolishing complex cache hierarchies. Even though this may seem restrictive, we are also able to map our OS architecture to processors like the Intel SCC [13] that only has core-local caches. Using prefetching techniques (see e.g. [30]), these caches can be used like local memories provided that code and data sections are small enough. We assume further that each core possesses at least a range-based memory protection unit and provides a basic interrupt system. Some cores additionally have a connection to off-chip facilities like I/O or external memory. Communication between the cores is based on messages that are sent through a NoC. The NoC provides a predictable timing behaviour. Communication on the NoC can be either scheduled statically for hard real-time guarantees or dynamically for best effort messages (see e.g. [26]–[28]).

In this section, we present the principles that underlie our MOSSCA implementation. We show how we map the API services that are defined for the MOSSCA abstractions to architectural components. For each component, we discuss its internal structure.

### A. Basic Principles

*1) Server API and OS Interface:* Applications can send requests to servers using the generic API services defined for server usage. While the API services define a function interface, the actual request is performed through messages sent over the NoC. It would also be possible to expose the messaging nature of server requests explicitly to applications, thus reaching a higher grade of transparency. However, we decided against this approach to keep the MOSSCA API uniform. To make any concrete server implementation easier to use, it is possible to implement user-level libraries to wrap the generic server API calls into more device-specific functions.

Internally, any server must implement an interface for control and management through the OS. Over this interface, service requests from application nodes are handed to the server and replies are sent to the applications. The interface is also used by the OS to mediate usage permissions requested by applications. In return, the OS provides a framework to ease the implementation of servers. The framework includes managed buffers for messages to and from the server.

*2) Messages:* All communication in MOSSCA is based on messages that are explicitly sent resp. received. Concerning application messages sent through channels, these messages stay in the receiver node's hardware buffer until they are explicitly read by software. While being sensible for user-level messages, this approach does not suffice for OS-internal control. If, e.g. the OS needs to stop a misbehaving application, it needs to directly address the kernel on the appropriate node. In turn, the kernel should react fast and interrupt application execution. To achieve such aims, MOSSCA employs so-called *privileged messages* to address the kernel of a node directly. Instead of simply being put into a buffer upon arrival, privileged messages additionally trigger an interrupt and thus can be processed with low latency by the kernel. In principle, such privileged messages can impact the predictability of application execution. They must be carefully used during proper execution. However, if used in exceptional situations, system predictability may already be harmed. In such a case, the system will actually benefit from such messages as they allow to restore a valid system state with low latencies. In MOSSCA, only OS servers are able to send such messages.

### B. Kernel

The kernel is MOSSCA's component that is in closest touch with the applications. It provides the full MOSSCA API through its system call interface. An additional messaging interface is used for coordination with an OS server.

*1) System Call Interface:* The kernel's system call interface provides the MOSSCA API. From kernel view, this API can be divided in two groups: (1) services that need interaction with an OS server, and (2) services that can be performed locally or in concert with a non-OS server. Locally performed services are use of channels (send/recv). Sending and receiving through/from a channel requires only interaction with the node's local hardware. For sending over a channel, the kernel plays an important role in ensuring that the defined usage bounds are kept by the calling application. Insofar, the kernel of a sender node ensures that the receiver node will not experience an overload situation. Requests to regular servers (non-OS servers) are handled in a similar manner. Like for sending over channels, the requester's kernel controls the traffic the server will experience. API services that concern execution control, e.g. activation and termination of threads, need coordination through an OS server and possibly interaction with the kernel on some other node. API services for resource de-/allocation need coordination through an OS server and possibly interaction with the kernel on some other node. As we have already discussed in V-A1, we integrate

the server interface into a function interface to keep the API uniform. Concerning the OS server, there is another reason for this approach: some services provided by the OS server need interaction with the kernel of the calling node. This can be achieved easier if the kernel has direct control of the service requests sent to the OS server and thus can increase performance of the implementation.

*2) Messaging Interface:* The messaging interface of the kernel is used for configuration and control through the OS server. This interface provides means to load application code on the node and to control application execution. In addition to the API services, the interface also allows to halt and continue application execution. These services can only be invoked by an OS server. The usage policies of channels and servers used by the local application are also set through the kernel's messaging interface.

*3) Structure and Mechanisms:* The MOSSCA kernel consists of two components, namely *communication control* and *execution control*. Communication control is responsible for any kind of communication originating in the application, i.e. it handles *channel* and *server* abstractions used by the application. For allocation of these abstraction and setting of usage policies, the kernel interacts with the OS server. Fine-grained control of usage of these resources is solely performed by the kernel. *Execution control* takes care of loading an application image and running the application thread. If requested by the OS server, it can also interrupt the running application thread.

For API services requiring interaction with an OS server, the kernel performs as much work as possible locally. Only work that needs interaction with other nodes or a global knowledge is marshaled into messages. These are sent to an OS server, where they are processed. All mechanisms implemented in the kernel deliver a predictable timing behaviour.

If several application threads have to be executed on one node, a user-level scheduler [31] can be employed by the application developer. However, the scheduler can be somewhat simplified (compared to a full-featured OS scheduler) as it does not need to cross addressing space boundaries. Thus, the application has full control of the execution resource. System analysis is not complicated by an OS scheduler and interrupts.

### C. OS Server

Within one MOSSCA systems, several OS servers can run in parallel, each managing a different part of the processor. Applications cannot address an OS server directly. Instead, they use the OS server indirectly through specific kernel system calls. The kernel communicates with the OS server by messages. Therefore, the OS server possesses only a messaging interface.

*1) Messaging Interface:* The messaging interface of an OS server is subdivided in two parts: The first part is responsible for communication with kernels that need to coordinate with the OS server. Over this interface part, execution control of threads is performed and the resource allocation/free requests are transmitted. The second part of an OS server's messaging interface is responsible for coordination between multiple OS servers and with other servers. This is important if, e.g. new channels are allocated during runtime.

*2) Structure and Mechanisms:* Like in the kernel, mechanisms in an OS server can be divided in communication and execution control. However, the OS server performs these tasks on a more global level. The communication control part of the OS server manages channel and server connections. If such a new connection is requested, the OS server checks whether the required resources are available. For server usage requests, this work is performed in concert with the affected server. The execution control part of the OS server manages the execution resources of a part of the processor. Additionally, it performs some monitoring of the system to ensure proper operation. If it detects anomalies in application behaviour, it can interrupt execution of these applications. All OS server mechanisms are implemented as non-interruptable transaction to deliver a predictable timing behaviour to applications.

### D. I/O Server

Any I/O server time-multiplexes I/O requests from application threads to the I/O device it manages. It provides means for bandwidth and latency management which allow to give certain timing guarantees to applications. The concrete implementation of the management depends on the specific device. MOSSCA provides general helpers to alleviate this implementation. The I/O Server is also the primary interrupt handler for its device. If necessary, it may forward interrupt requests to specialised nodes.

### E. Inter-Partition Communication Server

The communication channel abstraction of MOSSCA can principally ensure a predictable communication between any two application nodes. Therefore, it could also be used to implement communication between two partitions. Nevertheless, we are convinced that inserting an inter-partition communication server as mediator can greatly improve system predictability and composability. We reason as follows: the NoC interface of any node has to buffer received messages until they are processed by the software running on the node. Naturally, the space available for buffering is limited. An overflowing receive buffer may seriously impact the behaviour of the application thread running on the correspondent node. For communication within a partition, we do not see this as a big problem, as the application developer should be able to tune and synchronise his own application threads appropriately. However, he might have only few knowledge about the behaviour of other applications he needs to interact with. Thus, if a message from another partition arrives at some node, the node might not be able to handle it immediately. Hence, this message will block buffer space and thus can influence the receive behaviour experienced by intra-partition messages arriving later. We solve this problem through a dedicated Inter-Partition Communication Server. This server stores inter-partition messages until the receiver is actually ready to receive them. Thus, no inter-partition message can interfere with intra-partition messages. At the moment, the

IPCS implements communication buffers and blackboards for unbuffered communication. Applications can access an IPCS through the regular API services defined for servers, or a user-level library wrapper.

In the current implementation, all MOSSCA servers process requests in FIFO ordering. This allows to bound the response time for any request, depending on number of clients that use a server. However, in this approach the bound incurs a high pessimism due to the assumption that any request has to wait for requests of all other clients to be finished. The implementation will be optimised in the future such that knowledge about the timing behaviour of applications can be used to derive tighter WCET bounds for server requests.

### F. Discussion

The MOSSCA implementation achieves *predictability* and *analysability* in the following way: For any system service, as much work as possible is performed on the node the service request originates. Thus, if a service needs to use a server, the interferences with service requests from other nodes are reduced. Further reduction of interferences can be achieved by replicating servers, as we do with OS servers. MOSSCA services used by critical applications are implemented such that they can be performed in bounded time. Concerning allocation of resources during runtime, this requires that some tests are performed already offline. Hence, MOSSCA can guarantee bounded worst-case response times for any OS service, and also for any real-time application built upon these services. Additionally, the IPCS reduces interferences from communication between partitions. Thus, it also strengthens the *partitioning* properties of MOSSCA. Apart from that, the MOSSCA implementation continues the partitioning concepts introduced in the MOSSCA abstractions and architecture. *Fine-grained communication* within partitions is further guaranteed through the implementation of the communication channel abstraction in the kernel. *Coarse-grained communication* between partitions is provided by a dedicated inter-partition communication server. The IPCS averts the problems that might otherwise be introduced into the system if only communication channels were used for inter-partition communication. The implementation also provides the basis for *reconfiguration* of a MOSSCA system. On the one hand, resources can be allocated dynamically through the OS servers. On the other hand, OS servers also have full control of application execution an can exchange the application threads running on certain nodes.

## VI. Related Work

Most works on operating systems targeting explicitly many-core architectures have been performed in the domains of general-purpose, high performance, and cloud computing. Nevertheless, they have introduced concepts that we have taken up in the design of MOSSCA and extended for the use in SC systems. The Corey operating system [32] targets cache-coherent shared memory multicore processors. Like in MOSSCA, Corey tries to give the application thread full control of its primary execution resource, i.e. the core. A similar approach is found in the works on ROS [20] and Akaros [33], which introduce the *ManyCoreProcess* as a central abstraction. The general concept of MOSSCA servers to distribute OS and other functionalities on a manycore is based on the *factored operating system (fos)* [18] that targets cloud computing scenarios. The multikernel operating system Barrelfish [17], [34] views the manycore processor as a distributed system. One core idea of Barrelfish is to make any communication explicit and thus amenable for analysis. This approach has also been used for a long time in microkernel research (e.g. [35], [36]). MOSSCA additionally defines policies that constrain communication and thus help to ensure freedom of interference. Yang et al. [37] have developed a preemptive kernel to improve the kernel latency of Barrelfish for real-time operation. *Space-time partitioning (STP)* as a means to integrate general-purpose applications with diverse requirements within one manycore processor is introduced in the Tessellation OS [19], [38]. The STP concept is designed to provide certain QoS guarantees, but does not heed the requirements posed by a safety-critical systems. The Osprey operating system [21] aims to provide a predictable timing behaviour for computing clouds by means of *resource containers* that give guarantees about resource availability to the application.

## VII. Summary & Future Work

Fine-grained sharing of hardware resources is impacting the usability of today's multicore processors in safety-critical real-time systems. We have discussed, how these problems might be alleviated in future manycore processors as long as an operating system provides a sound basis for applications. Borrowing ideas from existing manycore operating systems, we have devised a manycore operating system for safety-critical applications. MOSSCA introduces abstractions that assist an application developer to ensure safety-criticality properties in the application. Its architecture and implementation continue these properties inside the operating system by distributing OS functionalities to several servers running on separate nodes. The central issue of a safety-critical system is the predictability of its behaviour, which today is achieved through partitioning the system. Partitioning in MOSSCA is partially achieved by mapping different application threads to different nodes of a manycore processor. Remaining sources of possible interferences are shared resources like the NoC and shared servers (including off-chip I/O). While interferences cannot be excluded due to the shared nature of these resources, MOSSCA helps to bound these interferences by providing usage policies for NoC communication and servers, and ensuring that the constraints defined in these policies are kept by the applications.

In the future, we plan to extend MOSSCA by mechanisms for fault-tolerance and dynamic reconfiguration to make it more robust while still ensuring a predictable system behaviour. Furthermore, MOSSCA will be ported to a real hardware platform. This work will allow us to derive concrete performance measurements and to perform an actual WCET

analysis of the OS. The MOSSCA approach is adapted to provide runtime environments (RTEs) for different industrial application domains. Most widely developed is an AUTOSAR RTE [39], which we will use for an automotive case study.

## ACKNOWLEDGEMENTS

Part of this research has been supported by the EC FP7 project parMERASA under Grant Agreement No. 287519.

## REFERENCES

[1] L. M. Kinnan, "Use of multicore processors in avionics systems and its potential impact on implementation and certification," in *28th IEEE/AIAA Digital Avionics Systems Conference, 2009 (DASC '09)*, Oct. 2009, pp. 1.E.4–1 –1.E.4–6.

[2] 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*, vol. 28, no. 7, pp. 966 –978, Jul. 2009.

[3] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm, "Predictability considerations in the design of multi-core embedded systems," in *Proceedings of Embedded Real Time Software and Systems*, May 2010, pp. 36–42.

[4] T. Ungerer *et al.*, "MERASA: Multicore Execution of HRT Applications Supporting Analyzability," *IEEE Micro*, vol. 30, pp. 66–75, 2010.

[5] D. N. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in *Design Automation Conference (DAC)*, June 2011, pp. 274 – 279.

[6] B. D'Ausbourg, M. Boyer, E. Noulard, and C. Pagetti, "Deterministic Execution on Many-Core Platforms: application to the SCC," in *4th Symposium of the Many-core Applications Research Community (MARC)*, Dec. 2011.

[7] F. Boniol, H. Cassé, E. Noulard, and C. Pagetti, "Deterministic execution model on cots hardware," in *Architecture of Computing Systems (ARCS 2012)*, ser. Lecture Notes in Computer Science, vol. 7179. Springer Berlin / Heidelberg, 2012, pp. 98–110.

[8] J. Nowotsch and M. Paulitsch, "Leveraging Multi-Core Computing Architectures in Avionics," in *Ninth European Dependable Computing Conference (EDCC 2012)*, Sibiu, Romania, May 2012.

[9] *PikeOS – Safe and Secure Virtualization, Product Data Sheet*, SYSGO AG, 2012.

[10] "Wind River VxWorks Platforms, Product Overview," Wind River Systems, Inc., 2013.

[11] D.-R. Fan *et al.*, "Godson-T: An Efficient Many-Core Architecture for Parallel Program Executions," *Journal of Computer Science and Technology*, vol. 24, pp. 1061–1073, 2009.

[12] *TILE-Gx8036 Processor Specification Brief*, Tilera Corporation, 2011. [Online]. Available: www.tilera.com/tile-gx8000

[13] J. Howard *et al.*, "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS," in *IEEE International Solid-State Circuits Conference, ISSCC 2010, Digest of Technical Papers, San Francisco, CA, USA*, Feb. 2010, pp. 108–109.

[14] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2012, pp. 983–987.

[15] "Kalray MPPA 256 processor," KALRAY Corporation, 2013, retrieved Dec. 2013. [Online]. Available: http://www.kalray.eu/products/mppa-manycore/mppa-256/

[16] Intel Corporation, "News Fact Sheet: Intel Many Integrated Core (Intel MIC) Architecture ISC'11 Demos and Performance Description," Jun. 2011, retrieved Dec. 2013. [Online]. Available: http://newsroom.intel.com/servlet/JiveServlet/download/2152-4-5220/ISC_Intel_MIC_factsheet.pdf

[17] A. Baumann *et al.*, "The Multikernel: A new OS architecture for scalable multicore systems," in *22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, Oct. 2009, pp. 29–44.

[18] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 76–85, Apr. 2009.

[19] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiat, "Tessellation: Space-Time Partitioning in a Manycore Client OS," in *1st Workshop on Hot Topics in Parallelism (Hot-Par '09)*, Mar. 2009.

[20] K. Klues, B. Rhoden, A. Waterman, D. Zhu, and E. Brewer, "Processes and Resource Management in a Scalable Many-core OS," in *HotPar10*, Berkeley, CA, Jun. 2010.

[21] J. Sacha, J. Napper, S. Mullender, and J. McKie, "Osprey: Operating System for Predictable Clouds," in *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W), 2012*, Jun. 2012, pp. 1 –6.

[22] eSOL Co., Ltd., "Real-time OS for many-core processors," retrieved 22.11.2013. [Online]. Available: http://www.esol.com/embedded/emcos.html

[23] F. Kluge, B. Triquet, C. Rochange, and T. Ungerer, "Operating Systems for Manycore Processors from the Perspective of Safety-Critical Systems," in *Proceedings of 8th annual workshop on Operating Systems for Embedded Real-Time applications (OSPERT)*, Pisa, Italy, Jul. 2012.

[24] P. J. Prisaznuk, "Integrated modular avionics," in *IEEE National Aerospace and Electronics Conference (NAECON)*, 1992, pp. 39–45 vol.1.

[25] C. Pagetti *et al.*, "Reconfigurable IMA platform: from safety assessment to test scenarios on the SCARLETT demonstrator," in *Embedded Real Time Software (ERTS'12)*, 2012.

[26] T. Bjerregaard and J. Sparso, "A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip," in *Proceedings of Design, Automation and Test in Europe (DATE)*, Mar. 2005, pp. 1226–1231 Vol. 2.

[27] K. Goossens, J. Dielissen, and A. Radulescu, "Aethereal network on chip: concepts, architectures, and implementations," *Design & Test of Computers, IEEE*, vol. 22, no. 5, pp. 414–421, 2005.

[28] R. Stefan, A. Molnos, and K. Goossens, "dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-up," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–1, 2012.

[29] S. Metzlaff, J. Mische, and T. Ungerer, "A Real-Time Capable Many-Core Model," in *Work-in-Progress Session of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011)*, Vienna, Austria, Nov. 2011.

[30] B. Cilku and P. Puschner, "Towards a time-predictable hierarchical memory architecture - prefetching options to be explored," in *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, 2010, pp. 219–225.

[31] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: effective kernel support for the user-level management of parallelism," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 53–79, Feb. 1992.

[32] S. Boyd-Wickizer *et al.*, "Corey: An Operating System for Many Cores," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 43–57.

[33] B. Rhoden, K. Klues, D. Zhu, and E. Brewer, "Improving per-node efficiency in the datacenter with new os abstractions," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 25:1–25:8.

[34] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the Barrelfish manycore operating system," in *Workshop on Managed Many-Core Systems, Boston, MA, USA, June 24, 2008*. ACM, Jun. 2008.

[35] J. Liedtke, "Improving IPC by Kernel Design," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, pp. 175–188, Dec. 1993.

[36] K. Elphinstone and G. Heiser, "From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?" in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 133–150.

[37] J. Yang, X. Long, X. Shen, L. Wang, S. Feng, and S. Zheng, "Towards rtos: A preemptive kernel basing on barrelfish," in *10th International Conference on Advanced Parallel Processing Technologies, Stockholm, Sweden, Revised Selected Papers*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 8299, pp. 47–61.

[38] J. A. Colmenares *et al.*, "Resource Management in the Tessellation Manycore OS," in *Second USENIX Conference on Hot Topics in Parallelism (HotPAR 2010)*, Jun. 2010.

[39] F. Kluge, M. Gerdes, and T. Ungerer, "AUTOSAR OS on a message-passing multicore processor," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES), 2012*, Karlsruhe, Germany, 2012, pp. 287–290.