On MILS I/O Sharing Targeting Avionic Systems

Kevin Müller
Airbus Group Innovations
Munich, GERMANY
Kevin.Mueller@airbus.com

Georg Sigl
Technische Universität München
Munich, GERMANY
Sigl@tum.de

Benoit Triquet
Airbus Group
Toulouse, FRANCE
Benoit.Triquet@airbus.com

Michael Paulitsch
Airbus Group Innovations
Munich, GERMANY
Michael.Paulitsch@airbus.com

21.03.2014

Published in the Proc. of the 10th European Dependable Computing Conference (EDCC 2014), Newcastle upon Tyne, UK, May 13-16, 2014, IEEE.

This paper discusses strategies for I/O sharing in Multiple Independent Levels of Security (MILS) systems mostly deployed in the special environment of avionic systems. MILS system designs are promising approaches for handling the increasing complexity of functionally integrated systems, where multiple applications run concurrently on the same hardware platform. Such integrated systems, also known as Integrated Modular Avionics (IMA) in the aviation industry, require communication to remote systems located outside of the hosting hardware platform. One possible solution is to provide each partition, the isolated runtime environment of an application, a direct interface to the communication's hardware controller. Nevertheless, this approach requires a special design of the hardware itself. This paper discusses efficient system architectures for I/O sharing in the environment of high-criticality embedded systems and the exemplary analysis of Freescale's proprietary Data Path Acceleration Architecture (DPAA) with respect to generic hardware requirements. Based on this analysis we also discuss the development of possible architectures matching with the MILS approach. Even though the analysis focuses on avionics it is equally applicable to automotive architectures such as AutoSAR.

1 Introduction

Future aircraft applications demand higher processing performance for providing more and more functionality to pilots, crew, air traffic management, airlines or passengers. Contrary, the deployment of additional computing hardware is difficult due to considerations on fuel economy in particular by saving weight. In addition to this trade-off, nowadays avionic system designer are encouraged to reduce costs for their processing platforms, for example, by reducing the physical size, weight or power consumption of the platform, but also by avoiding special-custom hardware designs [1]. Recently, to solve the differences among those three streams the trend of using multicore Commercial Off–The–Shelf (COTS) products has introduced safety-critical embedded systems usable also for avionics. COTS multicore hardware address all the presented issues by providing high performance with usually lower power consumption and small processing boards for affordable costs [1, 2]. Coincidentally, COTS multicores introduce new issues like higher complexity, concerns about life-time support of the vendors, and new questions regarding the processing platform's safe operation property [2, 3]. Especially long-term availability builds a major reason for preferring one hardware architecture over another one. However, today to reduce weight the available multicore architectures allow even tighter integration of avionic functions that is commonly known as new generations of Integrated Modular Avionics (IMA) [4].

Lately, aviation authorities also worry about security properties, in particular for cases where security vulnerabilities can affect safety properties of the aircraft [5]. Note, that for avionics the primary security objective is the assurance of availability and integrity. Confidentiality is less emphasized since it usually does not affect the system's reliability [6]. A challenging task for security engineers of avionics is the conjunction of relatively new security considerations with the well-entrenched safety standards. The concepts of Multiple Independent Levels of Security (MILS) [7] for developing secure systems fits well with the safety-driven concept of IMA since both approaches use the idea of isolated runtime environments for hosting applications.

To enable external communication in IMA or MILS systems, both architectures share similar issues regarding the realization of management and handling of I/O data streams. Usually, applications running isolated among each other on a common hardware require sharing of hardware devices, too. The scope of this paper is the discussion of current and novel approaches to implement I/O communication management in MILS systems. Nowadays deployed software-based approaches usually do not demand special hardware properties on the device itself to ensure secure (and safe) operation. However, those approaches induct increasing certification efforts, due to additional lines of code and also decrease performance [8]. For example [9] argues that a hardware implementation of separation simplifies the software stack, and high assurance could be gained thanks to formal verification of the AAMP7 processor. Novel hardware components provide special capabilities to off-load software parts into hardware and may lead to improve the communication management. The adaption of these hardware-based technologies for communication handling to aviation environments is novel. Thus, we extend known work on I/O sharing on MILS systems [10] to dependable secure embedded systems. In our paper we also assess one special COTS device with a proprietary I/O architecture of the leading chip manufacturer to process wired network traffic based on our generic requirements for secure operation.

In section 2 we discuss the concept of MILS and develop generic views on optimal system architectures with direct access to shared I/O devices based on MILS principles. Section 3 deals with general requirements on hardware devices for supporting direct interactions of applications running isolated in partitions. Section 4 provides an overview of embedded systems and their solutions for high-performance I/O processing. Section 5 discusses the proprietary Ethernet-based I/O system of Freescale's P4080 QorIQ hardware platform—the Data Path Acceleration Architecture (DPAA)—based on the requirements introduced by the previous section 3. Section 6 explains our measurement setup for analyzing the temporal interference of the DPAA components. The final assessment is discussed in section 7. In section 8 we develop a real system architecture based on the results of our hardware investigation and discuss its difference to the optimal approach of section 2. In section 9 we discuss other publications related to the topic of I/O sharing and related work with respect to the P4080 platform. Finally, section 10 summarizes our paper.

2 Generic MILS architecture with I/O

2.1 Multiple Independent Levels of Security (MILS)

The basic ideas of MILS systems trace back to the early 1970s [11]. Rushby re-introduced these concepts and termed them MILS in the 1980s [12]. He explains MILS as a two-level system approach [7]. First, on the *system-wide level* the system designers specify *security policies* to define what "secure" means for a specific system. This in particular includes information flow policies describing which software components of the system are allowed to communicate among each other and their use of communication resources. Second, these security policies are mapped to define rules on the lower *resource sharing level.* Note that integrated systems always share resources, like memory, devices or processing cores. Afterwards the system designers have to prove that the low-level security configuration on the resource sharing level follows the top-level specification on the system-wide level. According to Rushby, "*the crucial feature of the MILS approach is that it separates the problems of enforcing security policy from those of securely sharing resources*" [7].

Generally, MILS systems are based on two design principles:

1. *Strict Separation* of processing resources for the hosted applications.
2. *Controlled Information Flow* to allow information flow among those by default isolated applications.

Thus, a MILS-based system requires both, 1) a specific system layer that is able to separate the processing resources for ensuring the property of strict isolation and 2) the ability to provide well-defined communication channels for controlled information flow. A hardware approach fulfilling the first requirement is the strict physical separation of resources into isolated hardware processing environments. However, this approach usually demands specially designed hardware platforms, e.g. multiple physically separated memories, and is hardly feasible for COTS components.

As software approach, a Separation Kernel (SK) also implements the separation of resources based on COTS hardware by using hardware support, e.g. Memory Management Units (MMUs). Consequently, an SK can act as foundational component for a MILS system. In terminology of SKs the separated runtime environments of the system's resources are called *partitions.* Partitions host usually small software components for building up a secure system. To enable information flow among partitions the SK also provides controlled communication channels.

The original motivation for MILS was to improve assurance in complex Multiple Levels of Security (MLS) systems that span several security levels at the same time. The MILS approach now allows mapping of MLS parts to SKs, which can be certified to the required assurance [13]. Inside a partition the running application (component) should process only data belonging to one security domain. These components are known as Single–Level Security (SLS). However, when designing a system out of components, it is unlikely to deploy SLS components only. Some MLS components remain in a MILS systems. The SK or hardware components are examples of such MLS components, since they are able to access all data of the system. However, since MLS components have to ensure the strict separation property of data, they should be used with special care. Due to the de-composition of a system into analyzable *components* and their arrangement in layers as suggested by the MILS approach, a comprehensible and certifiable MLS *system* with the needed assurance can be constructed at affordable cost. This "divide and conquer" approach fits well current avionic approaches of IMAs. Furthermore, the clear structure supports correctness arguments in the certification process [14].

2.2 Interaction of Processing Cores with Hardware Devices

Essentially a processing core interacts with devices by one of the following mechanisms [15]: The first (1) mechanism is Programmed I/O (PIO), in which the core is in charge of the interaction with the hardware device. PIO can be implemented in two different ways:

(1a) Isolated I/O:
The core performs special assembly commands for I/O interaction with the device registers.

(1b) Memory-Mapped I/O:
The registers of the device are mapped into the system memory where the core can access them for performing common read/write transactions.

The second (2) mechanism is Direct Memory Access (DMA), where the hardware device transfers data between itself and the system memory and signals the completion of this transfer to the core, e.g. by interrupts. Interrupts are the third (3) mechanism of the communication from devices to cores. While some interrupts are triggered by DMA to special target addresses, so called *Message Signaled Interrupts* [16, section 8.6], others are triggered via dedicated pathways. Memory-Mapped I/O (1b), DMA (2), and interrupts via dedicated pathways (3) are important to understand this paper.

2.3 I/O Sharing

Of particular interest for MILS-based IMA systems interacting with hardware devices, is the question of how to realize the data exchange of partitions with the device. However, special requirements on avionic communication, e.g. introduced by using Avionics Full Duplex Switched Ethernet (AFDX) [17], require the system to not only route incoming data to one target partition, but also demands duplication and distribution of data streams to multiple target partitions. Such use-case-specific functional requirements are usually not covered by the functionality of COTS devices, which still leads to additional software support.

Figure 1 depicts three conceivable approaches of realizing safe and secure I/O management in IMA systems.

In software-based sharing (cf. Figure 1a) the driver of the hardware device is part of the SK and runs within its address space and with the highest privilege level. Apart from its task of managing the device the driver's task is to implement the distribution mechanism. This mechanism is in charge to classify the ingress data and to route it to the corresponding target partition. Hence, this driver is a MLS component since it processes all data flowing in and out of the system. The advantage of software-based sharing is its high flexible possibility of realization since the entire component is implemented in software. Another positive aspect of this design is that a partition does not require any device-specific knowledge. The driver uses the communication channels provided by the SK to interact with the partition and thus is able to provide hardware abstraction to a standardized interface. However, running MLS driver and routing algorithm within the address space of the SK requires high robustness and trust into its implementation due to the impact of possible fault propagation to the entire system in case of a device or driver fault. Thus, such a driver requires the same level of certification as the highest subsystem running on this platform managed by the SK, independent whether the subsystem uses the device or not.

To reduce this impact the approach of I/O sharing places the driver inside an isolated address space, mostly known as *I/O partition* (cf. Figure 1b). In this case the I/O partition is still an MLS subsystem, since it still manages data streams of all partitions requiring communication, despite their actual security classification. However, encapsulating this functionality in a separated partition, reduces the impact of failure propagation to the partitions interacting with the I/O partition only (e.g. loss of communication ability) and allows the I/O partition itself to recover in failure case. Consequently, this encapsulation might allow to reduce certification demands of the I/O partition to the level of the highest subsystem interaction with the I/O partition, while still allowing higher certified system to run on the same SK as long as they do not interact with the I/O partition.

For further improvement it is possible to deploy one SLS I/O partition for each certification level in the system (cf. Figure 1c). This reduces the required certification effort of each I/O partition to the highest of the connected subsystems, only. Each of those SLS I/O partitions has a dedicated interface to its hardware device, which could be in one case one device for each I/O partition, but could also leverage the new hardware capabilities of self-virtualization as shown in Figure 1c. Self-virtualizing COTS hardware allows processing, classification and distribution of data streams already in the device. Thus, the connected SLS I/O partition will only "see" data streams of its classification and does not have to separate data according to these classification parameters anymore.

To use this self-virtualization approach a small management driver runs in high privilege mode (either in the SK or in a privileged partition) to set up the device via a management hardware interface. Of course, this driver requires a high certification level, but efforts can be kept very small since it might only run at system boot time. The self-virtualizing device provides a configurable number of virtual functions (runtime interfaces) appearing to the system software as dedicated devices. The SK is able to separate the access registers of those virtual devices (e.g. by memory mapping) and assigns the virtual functions to the SLS I/O partition directly. Within each SLS I/O partition a small runtime driver manages the direct interaction with its virtual function. The approach introduces several advantages:

1. The separation of the driver software into management part and runtime part allows to implement drivers with small component code footprints. In addition only the management driver itself is of high criticality level to the system. The criticality level of the SLS runtime drivers depend on the connected applications, only.

2. The idea of using SLS I/O partitions reduces the required certification efforts for those I/O partitions, which only manage data stream of low critical systems. It also goes in line with the phrased goal in subsection 2.1 of reducing MLS components in MILS systems.

However, the down side of this self-virtualization approach is the increasing complexity (e.g. routing algorithm, various interfaces) required for the hardware device itself and additional requirements on the hosting hardware platform [8]. Nevertheless, we base higher trust on this as this COTS component is used frequently. Also it is able to be certified according to special certification rules [18, section 13]. This approach is justified by the frequent use of components and, hence, ability to eliminate design failures.

3 Device Requirements for Self-Virtualization

Our use case in this paper is driven by the avionic IMA I/O distribution having also the MILS principles and high I/O performance in mind. Avionic communication has to unify two demands: First, the usage of COTS devices combined with second, the introduction of special functional requirements demanded by aviation-specific technologies, like AFDX. However, generalizing the discussion of subsection 2.3, an avionic I/O partition is essentially a partition that shall directly interact with hardware devices. Thus for generalization, the deployment of self-virtualizing devices in MILS systems requires the devices to ensure the basic MILS properties of separation and controlled information flow as well. These properties are necessary for avoiding interference of one partition interacting with a device interface with another partition interacting with a different device interface. Separation must be implemented by *Spatial Separation* and *Temporal Separation*:

Requirement A: ***Spatial Separation***, the non-interference property between physical resources, has to be enforced on the one hand to the internal resources of the device in a way that runtime data of one runtime interface cannot interfere with data of another runtime interface, except the devices information flow policy allows this interference. Additionally, also the management interface, required for global device configuration, should be non-interferable by the runtime interfaces.

The spatial separation requirement also demands the device to be capable to the access control mechanism of the hardware platform to allow the SK the mapping of the device's interfaces directly into the partitions according to the system's information flow policy. Usually this separation is realized by interfaces assigned to different address spaces. Then the SK is able to separate the various address ranges with the processor's MMU as long as the interface's address range matches with the MMU's granularity.

Requirement B: A special aspect of spatial separation is the consideration of ***DMA***-capable devices. The device (or hardware platform) has to provide means to mediate DMA transactions to interact with different memory areas, dependent on whose behalf the DMA is performed. If the device's runtime interface performs the DMA, the access should be allowed to the memory regions belonging to the associated partition, only. DMA transfers required for the device's operation (e.g. private swap space residing in the system memory) should be isolated from the memory of partitions. Mediating DMA is commonly implemented by the interaction of various components located on the device and the hardware platform (e.g. I/O Memory Management Units (IOMMUs)).

Requirement C: ***Temporal Separation***, the non-interference property in time, requires the device to implement its functionality in a way that accesses of one runtime interface do not affect the processing time of another runtime interface or at least that any interference is bounded. This also applies to background operations, which can be triggered by a runtime interface. Since self-virtualizing devices share device-internal resources (e.g. the physical network connector of a network card) the device requires to implement internal arbitration of the tasks initiated by the device's runtime interfaces. In the best case this property also applies to the management interface. However for avionic purposes this might not be demanded, since the device will be initialized before normal operation, when strict execution times are not a requirement yet. In particular temporal separation could be challenging in multicore environments, where (runtime) interfaces can be accessed simultaneously by different cores.

Requirement D: Another major issue for Temporal Separation is also the device's ***Interrupt Handling*** since interrupts do not follow predetermined scheduling tables. Remember, MILS-based IMA systems usually have a strict hard real-time scheduling of multiple partitions among its cores. Runtime interrupts should affect one runtime interface and interrupt the related partition only, in the best case when the partition is active. This means for secure interrupt handling it is intended that only the connected runtime driver is informed, which interacts with the device interface that triggers the interrupt. Interrupt buffering mechanisms within the device, the hardware platform [19], or the SK are necessary to buffer the interrupt until the partition is scheduled on its cores. Global device interrupts should be routed via the management interface to the management driver only. In general those global interrupts should be handled immediately, which might disturb the temporal separation property of the MILS system. Another approach is to disable interrupts completely and poll the device interfaces for retrieving the status when the driver is active. This polling approach has the advantage of predictability but introduces performance overhead and delays.

Requirement E: ***Secure Initialization*** requires the device to block all runtime interface interactions until the device has been configured by the management interface and it has reached a secure state after its initialization sequence.

4 Hardware Overview

This section gives an overview on possible self-virtualization-capable implementations using the example of Ethernet-based I/O sharing. Unfortunately, publicly available information on the following hardware platforms is limited, which make judgments difficult on whether the implementation actually implements self-virtualization or not. However, we think that all platforms could support self-virtualization to a certain level. The x86 architecture normally uses the Peripheral Component Interconnect Express (PCIe) standard of Singe-Root I/O Virtualization (SR-IOV) [20]. SR-IOV targets device development capable to self-virtualization, which leaves it up to the developer of the PCIe device to fulfill our defined requirements of section 3. However, the hosting computing platform also needs to fulfill special requirements [8].

For proprietary implementations we researched mainly on communication processors of different vendors. Marvel realizes in its Xelerated family a component-based network accelerator [21]. Tilera provides such a communication processor with a network accelerator for wired networks called mPIPE [22]. Cavium also has introduced a network accelerator into their OCTEON III MIPS platform [23]. Qualcomm [24] and Texas Instruments' OMAP [25] series are less interesting for our study since they are targeting wireless communication. Last but not least Freescale provides also a component-based network accelerator, called Data Path Acceleration Architecture (DPAA), on its recently launched high-performance QorIQ PowerPC communication platforms [26]. For our study we focus on Freescale products, which are preferably considered for future avionics. Furthermore, Freescale matches best with the requirement of low power consumption compared to other market competitors and acts currently as largest supplier for embedded processors in the field of communication processors [27].

5 Freescale's P4080

5.1 Hardware Overview

The P4080 is an eight-core processor from Freescale's QorIQ family and depicted in Figure 2. It supports various virtualization extensions such as a high-privileged hypervisor mode, virtual CPU provision to guest applications, a high-performance network on-chip, called CoreNet, and several memory protection units to control and translate memory accesses by cores (through MMUs) and by external devices (through Peripheral Access Management Units (PAMUs)). Note, that the MMU and the PAMU are both important for a secure system configuration. Both management units operate as guards and have a minimal control granularity of the page size (4 kB). This infers that they can open a *window* to grant a core memory accesses to a consecutive page-aligned memory region of at least 4 kB. The PAMU—the P4080's IOMMU—requires for its operation an identifier attached to each DMA transfer trigger by a device. This device-configurable identifier is called Logical I/O Device Number (LIODN).

For external Ethernet communication the P4080 provides a high-performance Data Path Acceleration Architecture (DPAA) on-chip and additional interfaces such as PCIe or RapidIO for plug-in devices. Each of the five available PAMUs in the P4080 is placed in-between a subset of the external devices and the CoreNet for the purpose of observing accesses of the devices to the system's address space.

Of special interest for this paper are the DPAA components: Buffer Manager (BMan), Queue Manager (QMan), and Frame Manager (FMan). Those components allow hardware support for high-performance network I/O processing and off-load the processing cores.

5.2 Buffer Manager (BMan)

BMan's task in the DPAA is to provide buffer pointers of allocated memory to the other components. For this purpose a hardware component (e.g. a core or FMan) releases buffer addresses, referencing to the I/O space in the system memory, to BMan. This pointer again can be acquired by one of the DPAA components. Except its private swap memory space, BMan does not allocate memory inside the system memory. The memory allocation for I/O transactions is task of the cores and thus of software during initialization and runtime.

For its basic configuration, e.g. setting the boundaries of its private swap space or setting the LIODN for accessing this space by DMA, BMan provides a management interface inside the P4080's Configuration, Control and Status Register (CCSR) area. For runtime interaction, i.e. releasing and acquiring buffers, BMan offers ten page-aligned and memory-mapped *software portals*. Software portals in general contain the device registers required for core-device interactions. The access of a core to a software portal is controlled by the MMU using access windows. Due to the layout of the register map of a software portal the most fine-granular possibility of mediating accesses is by defining the following four windows of 4 kB in the MMU:

1. Performs buffer acquisition, e.g. receiving buffer pointers from BMan.
2. Performs buffer releases, e.g. providing buffer pointers to BMan.
3. Required for setting the buffer release commands, depending on the operational mode configured in window 4.
4. Configuration of the software portal's operational mode and its interrupt handling.

Thus it is exemplary possible to permit one core the buffer acquisition and releases without granting the same core rights to change the operational mode of the portal or handling interrupts.

For interrupt handling BMan provides eleven interrupt signals to the P4080's interrupt controller: one line for each of the ten software portals and one for global BMan interrupts usually under control of the management partition.

For the handling of different buffer shapes (e.g. different buffer sizes), BMan defines 64 buffer *pools*. Each pool is accessible by all software portal, i.e. each software portal can release or acquire buffers from one of the 64 pools without isolation inside the hardware. Additionally, BMan does not store information about the buffer shapes held inside the buffer pools. Thus, it does not perform runtime integrity checks of the provided buffer addresses, except pool depletion notifications.

5.3 Queue Manager (QMan)

QMan's task in the DPAA is to provide various queuing mechanisms for buffer pointers. The operational entity of QMan is a Frame Descriptor (FD). The essential value of an FD is the buffer pointer and the LIODN_OFFSET. Each buffer pointer points to the memory address a packet is stored. Hence, QMan does not manage the packet data by itself but manages their pointers. This behavior is comparable to BMan. The LIODN_OFFSET in the FD is required for calculating the final LIODN, which FMan uses to read the packet from memory.

Several FDs can be enqueued into a Frame Queue (FQ) identified by unique identifiers. Each FQ is capable to a different dequeue behavior, i.e. an FQ can be either parked or scheduled. If the FQ is parked, software needs to ask for dequeuing from this FQ by using the corresponding identifier. In contrast a scheduled FQ is automatically associated to a Work Queue (WQ) and a WQs again to a channel. In this case QMan uses an internal (priority-based) scheduling algorithm to provide the next FDs to a software portal. The 8 possible WQs per channel introduce the priority into the entire organization. Each software portal has access to 16 channels, where channel 0 is exclusively accessible by a software portal and the other 15 channels are shared among all software portals. Figure 3 provides an overview of the formerly described association. Note that QMan does not provide access control or integrity checks of software portals to the mentioned processing structures.

The configuration of an FQ including its dequeue behavior and its assignment to WQs and channels happens during runtime via the software portal.

Also QMan also provides 10 page-aligned and memory-mapped software portals for runtime interactions. In addition the management interface resides again in the CCSR. With the help of this area the trusted software configures global settings as private swap memory, the required LIODN for accessing this swap memory, or the previously mentioned LIODN_OFFSET different for each software portal. The most fine-granular access control to one software portal can be realized by dividing the portal into the following five pages, each page independently controllable by the MMU windows of 4 kB:

1. Enqueues FDs into FQs.
2. Retrieves the last dequeued FDs dependent on the command configured in window 5 (see point 5).
3. Reads failure notifications, e.g. occurred during processing enqueue commands of window 1.

4. Configures the FQs, e.g. the state or its assignment to a WQ or channel, and contains index registers for accessing the correct registers of the enqueue and dequeue rings[1]

5. For configuration of the dequeue command and dequeue mode, the portal's interrupt handling, as well as contains index registers for accessing the enqueue and dequeue rings[1]

As BMan, QMan is connected to the interrupt controller by eleven dedicated lines, one for each software portal, and one line for global QMan interrupts.

5.4 Frame Manager (FMan)

FMan is responsible for processing ingress and egress network data packets within the DPAA. For this purpose FMan manages the physical network ports and uses (in normal mode[2]) BMan and QMan for the address management of data buffers. Contrary to those components, FMan does not provide software portals for runtime interaction. However, runtime software does not have access to FMan directly. Runtime interaction to this component is performed by the software portals of BMan and QMan.

For configuration purposes FMan offers a management interface located in the CCSR. This management interface holds configuration value for the interface to BMan, QMan and the network ports, packet classification, parsing patterns, or packet routing settings. Each network port can use up to 8 buffer pools of BMan to acquire buffers for storing the received packets by this network port. The interface to QMan is configurable with default enqueue and dequeue FQs or with queue identifiers provided during the internal packet classification process.

5.5 Putting them Together

This section shows the packet flow and the interaction between the three previously introduced DPAA components. For simplicity we assume a faultless packet flow and do not deal with error handling. However, the components are capable to support this feature.

5.5.1 Ingress Flow

For an ingress packet (cf. Figure 4a) the network port receives the packet data. FMan retrieves information about packet sizes and asked BMan for an appropriate buffer address in accordance to its set of receive pools. After retrieving a valid buffer address, FMan copies via DMA the packet data following the first 256 bytes to the system memory using the configured LIODN for this network port. The first 256 bytes (i.e. the packet header) is copied into FMan's private in-component memory for further packet classification. During this classification process FMan determines the identifier for the target enqueue FQ for this data packet. Afterwards, FMan transmits the first 256 bytes into the system memory using DMA and enqueues the address of this buffer encapsulated by an FD to the gained FQ.

Now software can receive the packet address by interacting with one of QMan's software portals and can process the packet including its payload. After processing, software is

[1]Depending on the configured processing mode in window 5 either the registers in window 4 or registers in window 5 are required.

[2]FMan supports two operational modes: normal and independent. For high-performance data processing with on-chip classification of network traffic only the normal mode is of interest. Thus, only this mode was part of our investigation.

required to release the data pointer back to the correct BMan pool using a software portal (the pool identifier is part of the FD).

5.5.2 Egress Flow

For an egress packet (cf. Figure 4b), first software has to acquire a buffer address from BMan using a software portal. The packet's payload can now be created on this address. Afterwards, software has to enqueue the address to an FQ using one of QMan's software portal. Then, FMan receives the packet pointer from QMan and copies the packet via DMA into FMan. As LIODN for this DMA transfer, FMan adds the given LIODN_OFFSET of the FD and an internal LIODN_BASE value, which is related to the network port. After transmission of the packet to the network, FMan releases the packet pointer to BMan's buffer pool given by the FD and previously set by software during enqueue. The latter avoids the usually required interrupt for signaling the buffer deallocation to the core and increases performance.

6 Measurement Environment

The goal of our DPAA assessment is to determine whether temporal interference is observable when multiple cores interact concurrently with different runtime interfaces of a DPAA component. This is important for being able to rate whether the requirement of Temporal Separation is fulfilled and, thus, to say whether the platform is suitable for MILS systems considering its temporal behavior. We concentrated on the question of component-internal interference since the DMA capability of the components for sure has interference on the interconnect and competes on this interconnect with other cores and DMA devices [31].

For our DPAA assessment we implemented driver software for the previously described DPAA components. These drivers use a POSIX-alike interface to an in-house developed bare-metal OS-layer, providing the basic abstraction of the P4080's processing units. For temporal assessment we measured the write cycle of releasing and enqueuing buffers to BMan and QMan. One write cycle uses a loop of 16 iterations incl. special memory synchronization commands to write 16 consecutively arranged 32 bit registers.

Our driver code is executed by the cores coincidentally. Those cores interact with the memory, e.g. for fetching instructions or the command words supposed to be written to the DPAA component. Previous work on the assessment of the cores of the P4080 while interacting with memory coincidentally [32] shows a measurable increase of the execution latency dependent by the number of cores accessing the memory. In our assessment we only want to measure the timing behavior of the DPAA components. Since our setup unavoidably also uses the cores and memory we have to remove *probe effects* [33, Section 12.2] reasoned by those components. In our setup we expect three sources of probe effects:

1. Interference due to unfinished operations on the interconnect executed before our code sequence under examination.
2. The memory interference due to the concurrent fetch of instructions by all cores.
3. The concurrent load of the 16 parts of the 64 bytes command word supposed to be written to the DPAA component.

Our required code sequence comprise 15 assembly commands and thus fit into one cache line of the P4080. For avoiding probe effects 1) and 2) we configured the P4080 to use the L1 instruction cache only. This L1 cache is pre-loaded by double-aligning the code sequence to pages by introducing **nop** commands. The first alignment (16 **nop**) forces the cores to idle some time for leaving the interconnect finishing previous commands after core synchronization for avoiding probe effect 1). The second alignment (one **nop** before having the 15 commands of the loop) shall force the core to load the entire sequence into one cache line for avoiding further instruction fetch from memory during the measurements to avoid probe effect 2). However, to determine the effect of probe effect 2) we need to measure the time influence of the instruction fetch. In the best case its influence is static by increasing the number of cores.

Probe effect 3) is avoided by the alignment of the command word in the memory. The command word is 64 bytes and fits into one cache line of the L1 data cache. The command word is loaded from memory during the first iteration of the measurement loop. We can determine the effect of probe effect 3) by measuring the load sequence only, without writing the command to the device. By comparing these times with the times of the normal device access we should be able to determine the overhead of the load cycle. In the best case the offset between both values is positive (since writing to a device every time costs time) but constant by an increasing number of accessing cores.

Having these probe effects in mind we defined four setups for measuring the time interference induced by the DPAA component:

Setup 1: inst_fetch
We measured the time for instruction fetch without load (**lwz**) and store (**stw**) instruction by increasing the number of accessing core step-by-step from 0 to 8 cores. This is to determine the influence of probe effect 2). Figure 5 represents this setup by step 1).

Setup 2: cmd_fetch
We measured the behavior of the necessary data load from memory, only. This data holds the command supposed to be written to the specific DPAA component in the required format. For this second setup we skipped the store command (**stw**) to the device. This setup is necessary to determine the influence of probe effect 3). Figure 5 represents this setup by steps 1) and 2).

Setup 3: device
We measured the normal behavior, i.e. the normal interaction with the device. Figure 5 represents this setup by steps 1), 2) and 3a).

Setup 4: sleep_fakedev
We replaced the target addresses of the store commands by an address pointing back to a region in the main memory. This setup provided us confidence that (write) interaction with the devices differs from interaction with the memory and thus, verifies the results of setup 3. Due to the result presented in [32] we expect that this setup shows higher slopes of latency by concurrent access since each core additionally writes back to the main memory. Figure 5 represents this setup by steps 1), 2) and 3b).

For our measurements we performed 200 write cycles and averaged the results over 10 measurement repetitions. Despite those average value in our discussion we do not focus on the absolute results but on their relationship to each other for formulating a general statement of the behavior. Note, that the variation between our measured minimum and maximum values do not exceed 5% of the minimum value. Thus, those value are not recognizable in our diagrams and can be neglected in our opinion. In the following we argue with the average values of our measurements. Without loss of generality all values are retrieved from the runtime of core 0 accessing one runtime interface interfered by the other cores accessing other runtime interfaces of the same device. The results for BMan will be discussed in section 7.1.1. In section 7.2.1 we discuss the values for QMan. Due to the complexity of the DPAA we omitted time measurements of FMan. Furthermore, FMan operating in normal mode does not provide runtime interfaces to cores. This makes reliable interference measurements difficult. However, we discuss the theoretically possible time interference of this component in section 7.3.1.

7 Security Assessment of DPAA Components

7.1 Buffer Manager (BMan)

7.1.1 Security Assessment

Spatial Separation In subsection 5.1 we mentioned that interaction with BMan requires access to two memory areas. First, trusted software configures BMan via the management interface residing within the CCSR. Second, trusted or untrusted software (usually applications) can interact with BMan via one of the software portals providing the runtime interfaces. Due to the alignment, access control to those software portals can be mediated in a fine-granular way by the MMU. Thus, spatial separation from the perspective of the access to the runtime interfaces is sufficiently achieved.

However, security issues arise from BMan's internals of handling buffers. Software interacting with one software portal can access all buffer pools managed by BMan. For example this raises the threat scenario that one malicious application is able to acquire all buffers previously released by a correct behaving other application. BMan also does not offer a mechanism to limit the amount of pointers a software portal releases to a buffer pool. This opens the threat scenario concerning availability by exhausting the internal memory to prevent other software portals releasing their pointers. Additionally, each software portal can retrieve the depletion states of all buffer pools without hardware security checks.

DMA BMan only performs DMA for managing its private swap memory space. Those accesses can be controlled by the PAMU sufficiently. The for this purpose required LIODN for each BMan transaction accessing this swap space is configurable by BMan's management interface. For performing DMA BMan uses the DMA controller of QMan on hardware implementation level. However, we do not see how this implementation could be used to affect the system's security properties.

To summarize, BMan's DMA capability do not raise security vulnerabilities.

Temporal Separation For the temporal assessment of BMan we used the measurement setup explained in section 6. Figure 6a shows the behavior of all four setups. The green (and lowest) line in the diagram shows the result of setup 1, i.e. the core interference during instruction fetch of the measurement loop. This line illustrates that the cores almost do not interfere during instruction fetch, since the code blocks are cached in the L1 cache of each core. Thus, memory interaction is reduced. The brown (and highest) line in diagram 6a shows the behavior of the fake device (cf. setup 4), i.e. the values when the command is read from the memory but also written back to memory. Compared to the most important blue and red lines (middles ones of diagram 6a) we see that in particular the memory writes introduce much more interference compared to the memory read. The blue line (note that the blue and red line are almost equal) represents the setup 2 when the command is read from memory but the write command is replaced by a **nop** command. The red line illustrates setup 3 for the normal behavior of the device access. Both lines show almost the same values, however it is expected that setup 3 have a higher time influence than setup 2 due to the write command. Figure 6b shows the offset between both measurement setup, where values of setup 2 are subtracted from values of setup 3. The diagram confirms our expectation but also shows that the offset is constant. To conclude, BMan fulfills our requirement of Temporal Separation, since the timing of a core accessing one software portal does not get influenced when an increasing number of other cores access the device via other software portals concurrently.

Nevertheless, due to BMan's usage of DMA for offloading its internal memory structures, it is still conceivable that those DMA interfere with memory accesses of cores or other devices [31].

Interrupt Handling BMan has eleven dedicated interrupt lines to the P4080's interrupt controller. Each software portal owns one line. Consequently, portals among each other cannot interfere on one interrupt line. The eleventh interrupt is used for global BMan interrupts and is separated from the portal interrupts. The interrupt controller, which is under control of the SK, is able to route or mask interrupts to cores from each of those lines. Additionally, because of the software portal's layout of the register map, the access to the register required for the portal's interrupt handling can be further restricted by the MMU.

In our opinion the provided interrupt architecture is sufficient for secure interrupt handling of BMan.

Secure Initialization After system reset BMan is deactivated. The initialization of BMan is performed by the management interface residing in the CCSR and restricted to the SK only. The MMU protects this configuration area from other accessing entities than the SK. For enabling BMan the SK has to set up the memory address and the size of its private swap memory. Writing the size register triggers the initialization procedure of the component and enables access to the software portals. In the error case BMan might signal a failure via a global interrupt. However, the component does not provide a dedicated register signaling its proper initialization, and thus, it is not observable by the trusted software when this process has finished to allow untrusted software to start interacting with the component.

7.1.2 Improvements and Workarounds

With respect to the hardware improvements we recommend to implement the ability for strict spatial separation inside the buffer management part of BMan. For example this could be implemented by additional configuration registers in the CCSR, which configure access matrices of software portals to buffer pools. Furthermore, a limitation of pointers for each buffer pool will be helpful to prevent software portals to exhaust BMan's memory. In addition, we also recommend to add a flag showing that the initialization procedure of BMan has been successful.

A possible workaround for dealing with the vulnerability of shared pool accesses in the P4080 while still being able to use it for a MILS system is to extract the driver part from the untrusted parts of the system. This driver is placed into an extra partition and is required to demonstrate its proper operation. This is achievable since the trusted driver is very small and it only interacts with one software portal performing buffer acquisition and releases. The main security achievement of this driver is to introduce the missing access control matrices of software portals to buffer pools. Using all ten software portals separately can be achieved by instantiating this small driver ten times.

7.2 Queue Manager (QMan)

7.2.1 Security Assessment

Spatial Separation For QMan we see a security issue in terms of *Confidentiality* by the possibility of accessing shared pool channels by all software portals without further access control on hardware level. Via those channels data can be transmitted from one partition to another one by abusing QMan, which builds finally a covert channel [34]. Abilities of restricting access of software portals to the shared pool channels and using the dedicated channel would solve this issue. So far we do not see security issues concerning *Integrity* or *Availability* in the QMan architecture.

DMA QMan performs DMA for managing its private swap memory only. An SK-configurable register in the CCSR provides the LIODN for those transactions. Hence, the PAMU then control check those DMAs.

Even if QMan shares its connection to the CoreNet with BMan we do not see how malicious software could use this design decision to attack the system.

Additionally, QMan provides indirectly the LIODN for DMA transfers performed by FMan for the transmission of packets by setting the LIODN_OFFSET value in the FDs. We rate it as security-enhancing, that the LIODN_OFFSET in the FDs cannot be set by software via a software portal but can be configured by the management interface via the CCSR only. Consequently, software cannot misuse other hardware components (like FMan, when performing packet transmission) to access memory areas the software is not permitted to.

Temporal Separation For the temporal assessment of QMan we again performed our measurement setup of section 6. Like in case of BMan, the green line in diagram 7a illustrates the instruction fetch (setup 1) of the interaction sequence with the device without load and store instructions. Again, we can conclude that due to the enabled L1 instruction cache the cores do not interfere significantly while performing the required memory interaction for instruction fetch. Also similar to BMan is the brown/filled circle graph of setup 4 showing again a huge influence when the store commands write back to a memory region. Furthermore, setup 2—the blue line—and setup 3—the red line—show the same behavior like BMan. Diagram 7b plots the offset of both graphs. This diagram again shows that the write cycle to the device only adds and constant offset to the processing time with an increasing number of accessing cores to QMan's software portals. Thus, QMan internally supports the property of Temporal Separation of its software portals sufficiently. However, also QMan uses DMA for offloading its internal processing structures into the memory. These DMA's can induct interference with other cores and devices accessing memory as shown in [31].

Interrupt Handling In essence the assessment of BMan's interrupt handling in section 7.1.1 applies also to QMan. Nevertheless, there is one difference due to the layout of the memory map of a software portal: The interrupt handling registers of a software portal cannot be split apart from the runtime software since registers located in the same memory page are required for processing enqueue and dequeue commands.

However, this does not change our opinion regarding the sufficiency of the implementation for secure interrupt handling of QMan, since each software portal has its own interrupt line to the P4080's interrupt controller.

Secure Initialization QMan's properties of secure initialization are equals with the ones provided by BMan analyzed above in section 7.1.1.

7.2.2 Improvements and Workarounds

To improve the hardware design itself we suggest to alias the producer index register required for writing the enqueuing ring, and consumer index register required for reading the dequeuing ring or the message ring into the memory pages the associated rings are already located. This modification would enable the possibility to grant dedicated access to the required ring pages only. Since the dequeue command register would be still located in a different page (window 5) than the dequeue ring processing registers (window 2), a secure system design were possible, in which an untrusted (receive) driver can just dequeue frames without being able to change the dequeue command for sneaking FDs of other FQs.

However, this hardware improvement still has the security vulnerability that each software portal can enqueue FDs to each FQ, which for example could be used for illicit data flows to other portals. To counteract this vulnerability the hardware should provide a hardware-configurable access matrix in the management interface to control enqueue access of software portal to FQs. If this feature is implemented for this case, it will be also the possibility to introduce the same functionality for the dequeue and message rings, which also would disarm the vulnerability of dequeuing various FDs.

Using the available hardware architecture for secure I/O sharing it is necessary to counteract the identified limitations by software. For this purpose a small trusted and encapsulated runtime driver is required that interacts with a software portal. The security achievement of the driver is to implement the missing access control matrices, which control the access of software portals to FQs. Multiple software portals can be handled by multiple instances of this driver.

7.3 Frame Manager (FMan)

7.3.1 Security Assessment

Spatial Separation In normal mode FMan provides only a management interface for configuration purposes. Runtime interaction are performed on behalf of BMan for buffer management and QMan for data movement. FMan is hard wired to both components for receiving and transmitting packet pointers. The setup of FMan can be done at system boot time and for avionic purposes stay mostly static during runtime. Thus, during runtime direct interactions of untrusted partitions with FMan are not required. This also means that spatial separation of the component's interface is implemented sufficiently. Analysis of FMan's internal structure is difficult to evaluate due to the lack of direct access from a runtime interface. From an analysis view of the documentation, FMan shares its internal memory among all packets. Also the sub-components, responsible for packet parsing and classification are shared. Thus, we assume that packet data will be distributed within FMan according to the available internal resource without having a visible "spatial" separation. However, in our case setup we have not been able to exploit this architecture and to break the internal resource separation of the component. More detailed analysis will require further, probably company-confidential information of FMan's vendor Freescale.

DMA FMan is in charge to perform DMA with the partition's memory for the purpose of receiving and transmitting the data packets. The PAMU is able to control those DMA. The essential LIODN is calculated by values configurable for the physical network interface the packet is processed by and for the transmission part also configured by the management interface of QMan. Since runtime interfaces are unable to modify the LIODN values DMAs can be restricted by the PAMU sufficiently.

Additionally, for packet transmission, i.e. reading data from a partitions memory, we do not see an issue in FMan's internal processing in terms of security properties. The transmitting buffer can reside within the partition's memory space. This memory is protectable by the PAMU. However, the driver has to ensure the correct values for the BMan buffer pool during buffer management.

A security issue arises for the reception part of data packets. As soon as FMan receives a data packet it asks BMan for a data buffer. Afterwards, FMan copies all data bytes following the first 256 bytes via DMA to this buffer located in the system memory. As next step, FMan performs packet classification on the remaining 256 bytes. As last steps, FMan copies the 256 bytes via DMA to the in step one released buffer and enqueues the buffer pointer to the queue determined by the classification results.

The issue arising from this processing sequence is that the data packet is transmitted to the system's memory before the data has been classified. This means that it is impossible to route the DMA into the correct partition directly.

Temporal Separation FMan does not provide a runtime interface to software in normal operation. Thus, we are unable to interact with FMan exclusively. In consequence we are also unable to provide real system measurement values as provided for BMan and QMan. However, due to its DMA capability for reading and writing processed packet from/to memory, we expect again that cores or other DMA-capable devices are able to temporally interfere with FMan. Furthermore, we expect that internal temporal separation exists sufficiently, i.e. an attacker is unable to retrieve information on the internal processing and also unable to manipulate the internal operation of FMan.

Interrupt Handling All FMan components provide various sets of interrupts. However, since FMan does not provide a runtime interface to handle runtime-specific interruption all interrupts have to be handled by the SK. For security this is sufficient but could be problematic for performance reasons in particular since one core could interrupt another one (processing another application) by sending faulty data packets. However, this depends also on the system's configuration and on which core the SK handles interrupts.

Secure Initialization All components of FMan are deactivated on system reset. The initialization is initiated by writing to component-dependent registers. Some components require a special amount of processing cycles until the initialization has been finished. However, those components do not allow to determine the current of the initialization phase. They only block access to their functionality, which could allow data packets to pass FMan without proper classification. Data processing itself can be enabled by writing registers in the BMan Interface of FMan. Errors during the initialization phase might be reported via interrupts. However, the missing status bits for secure initialization arise security issues, since software can not prove the secure state of the component.

7.3.2 Improvements and Workarounds

Most parts of FMan are hard-coded and thus are not applicable to modifications. However, a small part of FMan's processing behavior can be updated by a firmware. After discussing our findings with Freescale we received the confirmation that FMan's partitioning security issue due to selecting the receive buffer before data classification cannot be fixed by changing this firmware in the P4080. However, Freescale has been developed the successive T-series of its PowerPC architecture also containing the DPAA. In new T-series Freescale further improves the FMan implementation and introduces the concept of *virtual storage profiles*, which enable the developers to implement the identified missing functionality [35, section 5.10.5.1.2].

Regarding secure boot we recommend that all FMan components should provide status identifiers signaling their proper initialization.

8 MILS Architecture on P4080

Our assessment of Freescale's DPAA on the P4080 platform has shown deficiencies of separation properties to use the hardware accelerator in a MILS system, when untrusted drivers shall be used for managing the hardware. However, those limitations can be circumvented by using small trusted drivers implementing some functionality in software and running in isolated partitions with dedicated direct access to the runtime interface of the DPAA components. In subsection 2.3 we discussed approaches of realizing I/O in MILS-based IMA systems. Remember, deploying COTS I/O hardware for IMA still require some software to implement special functionality required by avionic-specific communication technologies, like AFDX data distribution. This matches with the outcome of the DPAA analyzes, which also requires (trusted) software for secure data handling. In the following we will discuss possible system solutions based on our results.

8.1 MLS I/O Partition

Figure 1b shows the approach of IMA systems for realizing shared devices. Here the implementation combines the driver and a software-based data distribution in a central MLS I/O partition. This approach is similarly applicable to circumvent the identified limitations of the discussed DPAA components. The reason is that the MLS I/O partition software is system critical and thus highly trusted and each communicating partition is routed to the hardware via this I/O partition. However, MLS I/O partitions introduce a high certification overhead, due to their system critical functionality with respect to security and safety.

On implementation level the driver part of the MLS I/O partition interacts with one runtime interface of BMan and one runtime interface of QMan, only. The required I/O buffers for packet receive and transmit paths are also encapsulated inside the I/O partitions' boundaries. Since the interacting software component is trusted, spatial isolation properties inside the DPAA components are not required. However, this approach utilizes the provided hardware features of the P4080 inefficiently, which reduces performance.

8.2 SLS I/O Partition

First, a small trusted runtime driver in an isolated partition for each provided runtime interface is sufficient **to circumvent the limitation of security features of BMan and QMan**. Combining this need with the idea of implementing the required functionality in software leads to the idea of using multiple SLS I/O partitions (cf. Figure 1c). Other application partitions get access to the SLS I/O partitions via communication channels provided by the SK.

The runtime drivers have two differing tasks:

1. For BMan it manages the (local) buffer pools required for the communication.
2. For QMan it controls the setup of the queues and in particular to restrict the access to valid queues for enqueue and dequeue operations.

Multiple SLS I/O partitions improve the approach of one MLS I/O partition (cf. Figure 1b) regarding reduced code complexity and expect higher performance. Performance stems from enhanced utilization of hardware features. Complexity reduction stems from using SLS instead of MLS components. This SLS approach differs to the one presented in Figure 1c since each system critical runtime driver runs encapsulated in its own partition and outside of the less critical SLS I/O partition. Thus, it decouples the highly trusted drivers from the less trusted distribution functionality of the SLS I/O partition as shown in Figure 8.

Second, FMan does not allow to route receiving data packets directly into the correct target partition. **To circumvent FMan limitations** it is possible to use a global read-only memory resource among all SLS I/O partitions to receive data from FMan. Given by FMan's hardware-internal processing flow, it first transfers the packet data (higher 256 bytes) of a receiving packet directly to this read-only memory. Afterwards, FMan determines the correct SLS I/O partition by analyzing the header of the packet, transfers the header to the read-only memory, and enqueues the buffer pointer into a queue targeting the respective SLS I/O partition. This receiving SLS I/O partition retrieves the valid pointer and can read it from the globally accessible read-only memory. Finally, it can process the entire packet and distribute the payload to other partitions.

The read-only property of this memory area is required to fulfill the integrity property, since otherwise every SLS I/O partitions could modify all data packets, even if they belong to a different SLS I/O partition. As long as confidentiality is not claimed by the system's security policy, this approach allows to decrease the certification effort for the SLS I/O partitions depending on the subsystems connected to an instance of such a I/O partition. For today's avionic system we can assume this fact, since the major concern is the reliable operation of the system (thus availability and integrity) and not its confidential operation. However, if confidentiality is demanded all SLS I/O partitions are required to be highly trusted again. For sending packets the I/O buffer can reside within the SLS I/O partition satisfying all security properties.

9 Related Work

Bradetich, Alves-Foss et. al deny in [10, 36] their recommendation to use the DPAA for general purpose MILS systems. Despite our support for their opinion that the P4080 generally misses features for being used in general purpose MILS systems, their rationale for this conclusion is incomplete from our perspective. In [36] the authors claim limitations of BMan, we have figured during our study as well. Beyond this work, we have analyzed

BMan more detailed and additionally investigated QMan and FMan. Furthermore, we provided possible approaches to workaround the identified limitations of those components in particular with respect to the requirements of avionic MILS systems and other systems requiring availability and integrity properties only.

Münch et. al discuss in [8] the use of PCIe SR-IOV-capable hardware devices, their limitations on Freescale's P4080 platform and possible circumvention. To conclude this work, the P4080 supports the basic functional requirements for SR-IOV devices but does not provide enough hardware protection functionality for strict spatial separation. Thus, SR-IOV-capable network cards cannot be used for MILS systems sharing the network link. The DPAA could replace this technology but requires the implementation of the discussed approaches.

Nowotsch and Paulitsch performed in [32] measurements on multicore interference by using various memory access patterns of cores. Their platform of examination has been Freescale's P4080, too. The motivation of their analysis had a safety background. However, our work in this paper extends their work with the investigation of other on-chip hardware accelerations and adds the security perspective.

10 Conclusion

The scope of our paper was to examine strategies for sharing device access of systems using the design principles of Multiple Independent Levels of Security (MILS). We first defined our view on MILS and discussed afterwards generic strategies for secure device sharing. Some of those strategies have been implemented in current avionic software. However, the demand for higher performance brings us nowadays into the position to examine sharing strategies, which improve utilization of direct hardware access. Thus, we defined general requirements on hardware devices for allowing direct access to the device of various untrusted applications encapsulated by partitions following the MILS principles. In the following we exemplarily investigated the Data Path Acceleration Architecture (DPAA), e.g. available on Freescale's P4080, which is considered as a candidate for future avionic systems. We evaluated the network processing components of the DPAA against our previously generic derived hardware requirements. Our conclusion is that the components, in particular Buffer Manager (BMan) and Queue Manager (QMan) do not provide enough separation properties to allow access to the components by untrusted software. However, by knowing the limitations we can provide workarounds, which still allow an improved utilization of hardware. That promises a gain of performance. This performance increase results in particular due to the avoided payload transfers for passing partition borders usually present in MILS-based systems. We presented a novel secure architecture that addresses availability and integrity properties despite the lack of hardware separation. This architecture is built on Single–Level Security (SLS) software components ensuring separation. Smartly leveraging multiple SLS components instead of a Multiple Levels of Security (MLS) component reduces the code complexity of the MILS system. Finally, this reduces the effort for the certification process required for highly critical embedded systems, such as avionics.

Acknowledgment

This work was supported by the European Union's 7th Framework Programme project EURO-MILS (ID: ICT-318353), and the German BMBF project SiBASE (ID: 01IS13020). We would like to express our gratitude to Freescale for their support.

References

[1] European Commission, “Mixed Criticality Systems,” Brussels, Tech. Rep. February, Feb. 2012.

[2] B. Triquet, “Mixed Criticality in Avionics,” in *Workshop on Mixed Criticality Systems*. European Commission, Feb. 2012.

[3] EASA, “Certification Memorandum - Development Assurance of Airborne Electronic Hardware,” Tech. Rep. EASA CM - SWCEH - 001, 2011.

[4] RTCA, “DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations,” Tech. Rep., 2005.

[5] EUROCAE / RTCA, “ED-202 / DO-326: Airworthiness Security Process Specification,” 2010.

[6] R. D. Cerchio and C. Riley, “Aircraft Systems Cyber Security,” in 30^{th} *Digital Avionics Systems Conference*. Seattle, WA, USA: IEEE, 2011.

[7] J. Rushby, “Separation and Integration in MILS (The MILS Constitution),” SRI International, Tech. Rep. SRI-CSL-08-XX, Feb. 2008.

[8] D. Münch, O. Isfort, K. Müller, M. Paulitsch, and A. Herkersdorf, “Hardware-Based I / O Virtualization for Real-Time Embedded Avionic Systems Using PCIe SR-IOV,” 10^{th} *International Conference on Embedded Software and Systems*, Dec. 2013.

[9] D. Greve, R. Richards, and M. Wilding, “A Summary of Intrinsic Partitioning Verification,” in 5^{th} *International Workshop on the ACL2 Theorem Prover and Its Applications*. Austin, Texas, USA: ACM, 2004.

[10] R. Bradetich, “Framework for Evaluating Information Flows in Multicore Architectures for High Assurance Systems,” *National HCSS Conference*, 2012.

[11] J. P. Anderson, “Computer Security Technology Planning Study,” Tech. Rep. 2, 1972.

[12] J. Rushby, “Design and Verification of Secure Systems,” ACM Operating Systems Review, Tech. Rep., Dec. 1981.

[13] J. Alves-Foss, W. S. Harrison, P. W. Oman, and C. Taylor, “The MILS Architecture for High-Assurance Embedded Systems,” Tech. Rep., 2006.

[14] EUROCAE, “ED-203: Airworthiness Security Methods and Considerations,” no. Draft RC 1.0, 2012.

[15] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman, “IBM Research Report - Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines,” Tech. Rep., 2008.

[16] PSI-SIG, “PCI Local Bus Specification Revision 2.2,” 1999.

[17] Aeronautical Radio Incorporated (ARINC), “Aircraft Data Network Part 7: Avionics Full Duplex Switched Ethernet (AFDX) Network,” Jun. 2005.

[18] EUROCAE / RTCA, “ED-80 / DO-254, Design Assurance Guidance for Airborne Electronic Hardware,” 2000.

[19] W. Huang, "Introduction of AMD Advanced Virtual Interrupt Controller," *XenSummit 2012*, no. August, 2012.

[20] PCI-SIG, "Single Root I/O Virtualization and Sharing Specification - Revision 1.01," 2010.

[21] Marvel. Network Processor. http://www.marvell.com/network-processors/.

[22] Tilera. TILE-Gx Processor Family. http://www.tilera.com/products/processors/TILE-Gx_Family.

[23] Cavium. OCTEON III CN7XXX Multi-Core MIPS64 Processors. http://www.cavium.com/OCTEON-III_CN7XXX.html.

[24] Qualcomm. Qualcomm Technologies. http://www.qualcomm.com/technologies.

[25] Texas Instruments. OMAP^{TM}Applications Processors. http://www.ti.com/lsds/ti/omap-applications-processors/overview.page.

[26] Freescale. QorIQ Communications Processors High-Performance Tier. http://www.freescale.com/webapp/sps/site/taxonomy.jsp?code=QORIQ_HIGHPERFM.

[27] J. Byrne, T. R. Halfhill, and L. Gwennap, "A Guide to Multicore Processors," Tech. Rep., 2013.

[28] Freescale Semiconductors, "P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual," Tech. Rep., 2011.

[29] Freescale Semiconductor, "QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual," Tech. Rep. 2, Nov. 2011.

[30] S. Srinivasan and T. Peters, "P4080DS U-boot & Linux," Freescale Semiconductors, Tech. Rep. FTF-NET-F0683, 2010.

[31] S. Schönberg, "Impact of PCI-Bus Load on Applications in a PC Architecture," in 24th *Real-Time Systems Symposium.* IEEE, 2003.

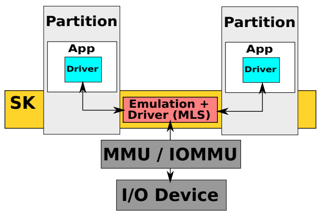
[32] J. Nowotsch and M. Paulitsch, "Leveraging Multi-Core Computing Architectures in Avionics," in 9th *European Dependable Computing Conference.* IEEE, 2012.

[33] H. Kopetz, *Real-Time Systems - Design Principles for Distributed Embedded Applications*, 2nd ed. Springer, 2011.

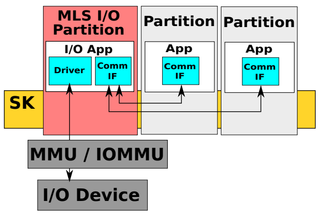
[34] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE Journal on Selected Areas in Communications*, vol. 21, Jan. 2003.

[35] Freescale Semiconductor, "T4240 Product Brief," Jun. 2013.

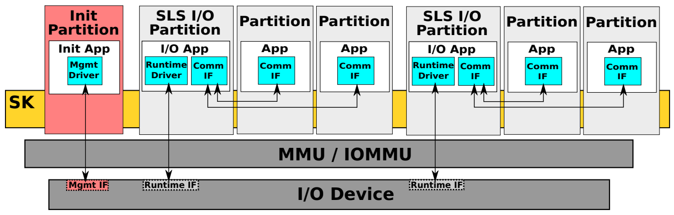
[36] J. Alves-Foss, P. Oman, R. Bradetich, X. He, and J. Song, "Implications of Mult-Core Architectures on the Development of Multiple Independent Levels of Security (MILS) Compliant Systems," University of Idaho, Tech. Rep., 2012.



(a) Software-based Sharing



(b) I/O Distribution (using one MLS I/O Partition)



(c) I/O Distribution (using Self-Virtualization and SLS I/O Partitions of different certification requirements)

Figure 1: Overview of Memory Protection Strategies. The red background color indicates the trusted items required for I/O sharing (in addition to the SK and the hardware)

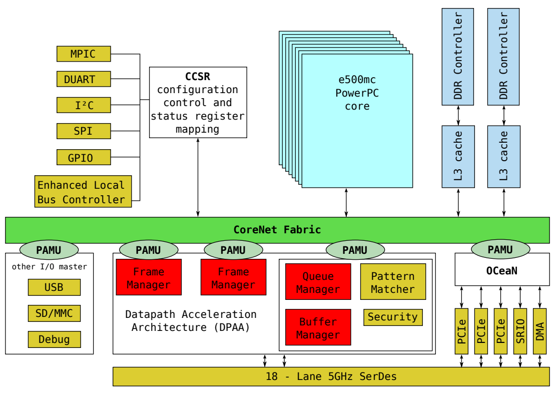


Figure 2: Block diagram of Freescale's P4080 system-on-chip with red highlights on the components of our investigations. [28]

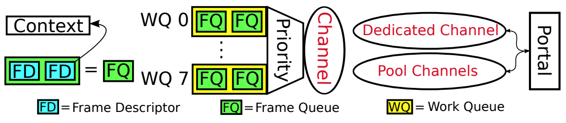
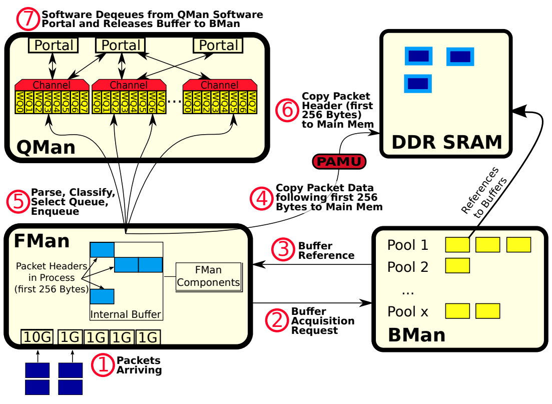
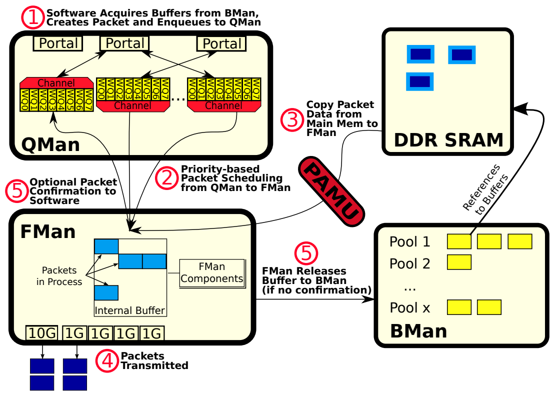


Figure 3: Overview on QMan's internal processing structures [29]


7 Software Dequeues from QMan Software Portal and Releases Buffer to BMan
Portal
Portal
Portal
Channel
Channel
Channel
QMan
DDR SRAM
6 Copy Packet Header (first 256 Bytes) to Main Mem
PAMU
4 Copy Packet Data following first 256 Bytes to Main Mem
5 Parse, Classify, Select Queue, Enqueue
References to Buffers
FMan
Packet Headers in Process (first 256 Bytes)
FMan Components
Internal Buffer
3 Buffer Reference
2 Buffer Acquisition Request
Pool 1
Pool 2
...
Pool x
BMan
10G
1G
1G
1G
1G
1 Packets Arriving

(a) Ingress Packet Flow


1 Software Acquires Buffers from BMan, Creates Packet and Enqueues to QMan
Portal
Portal
Portal
Channel
Channel
Channel
QMan
DDR SRAM
3 Copy Packet Data from Main Mem to FMan
PAMU
2 Priority-based Packet Scheduling from QMan to FMan
5 Optional Packet Confirmation to Software
References to Buffers
FMan
Packets in Process
FMan Components
Internal Buffer
5 FMan Releases Buffer to BMan (if no confirmation)
Pool 1
Pool 2
...
Pool x
BMan
10G
1G
1G
1G
1G
4 Packets Transmitted

(b) Egress Packet Flow

Figure 4: DPAA Network Packet Flow in Normal Mode (modified pictures of [30])

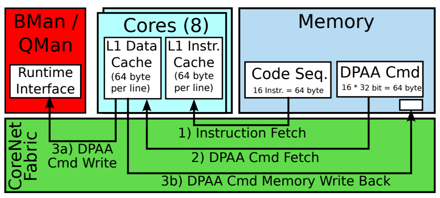
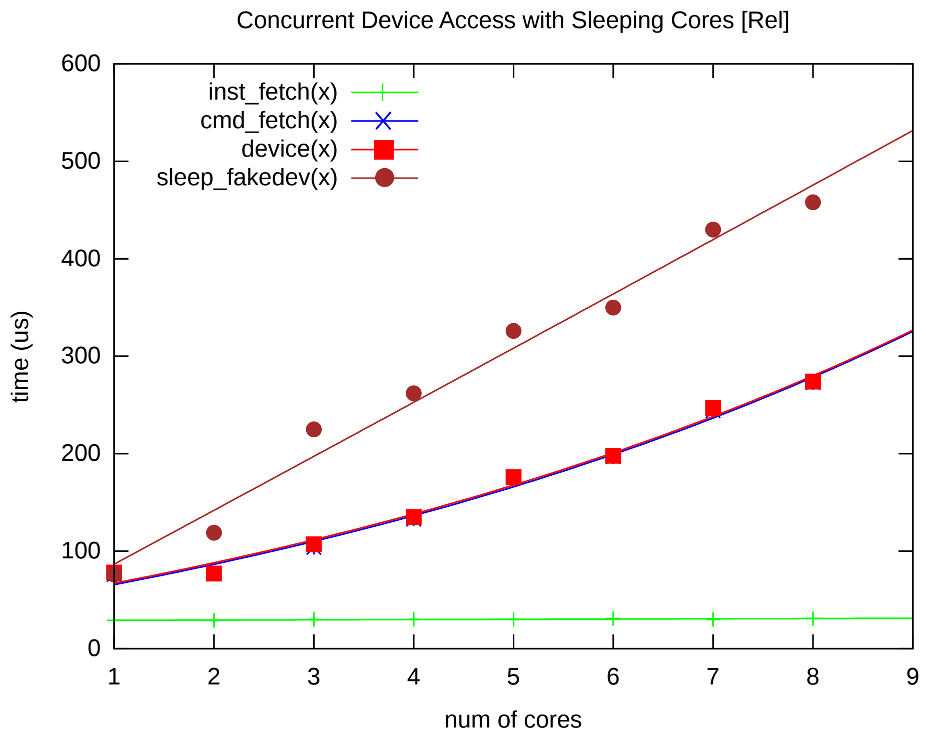
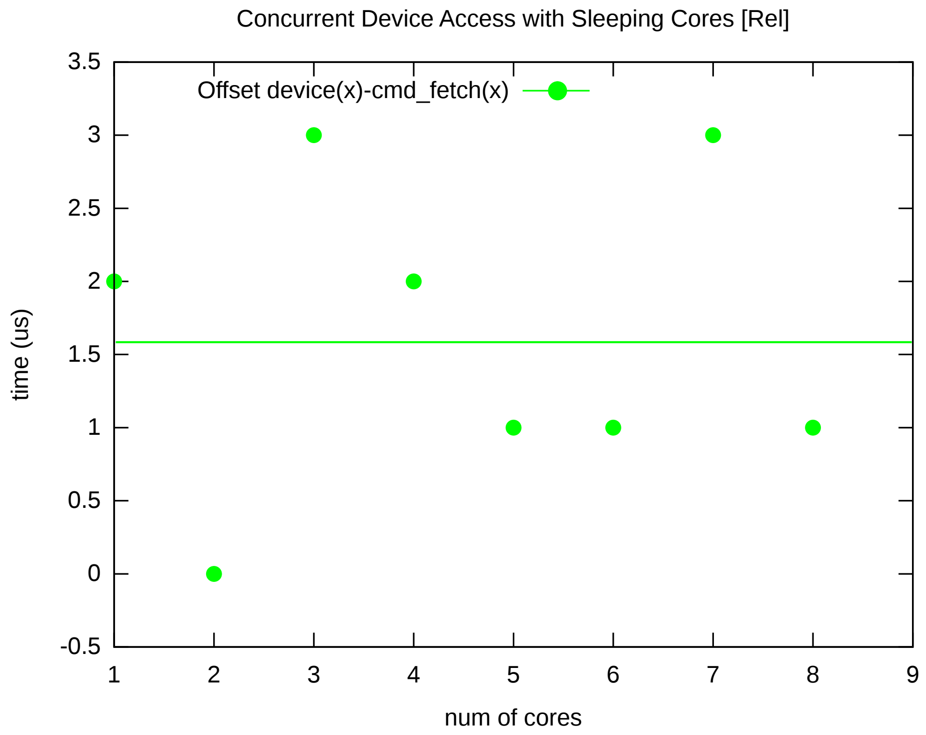

BMan / QMan
Runtime Interface
Cores (8)
L1 Data Cache (64 byte per line)
L1 Instr. Cache (64 byte per line)
Memory
Code Seq. 16 Instr. = 64 byte
DPAA Cmd 16 * 32 bit = 64 byte
CoreNet Fabric
3a) DPAA Cmd Write
1) Instruction Fetch
2) DPAA Cmd Fetch
3b) DPAA Cmd Memory Write Back

Figure 5: Steps of the measurement setup

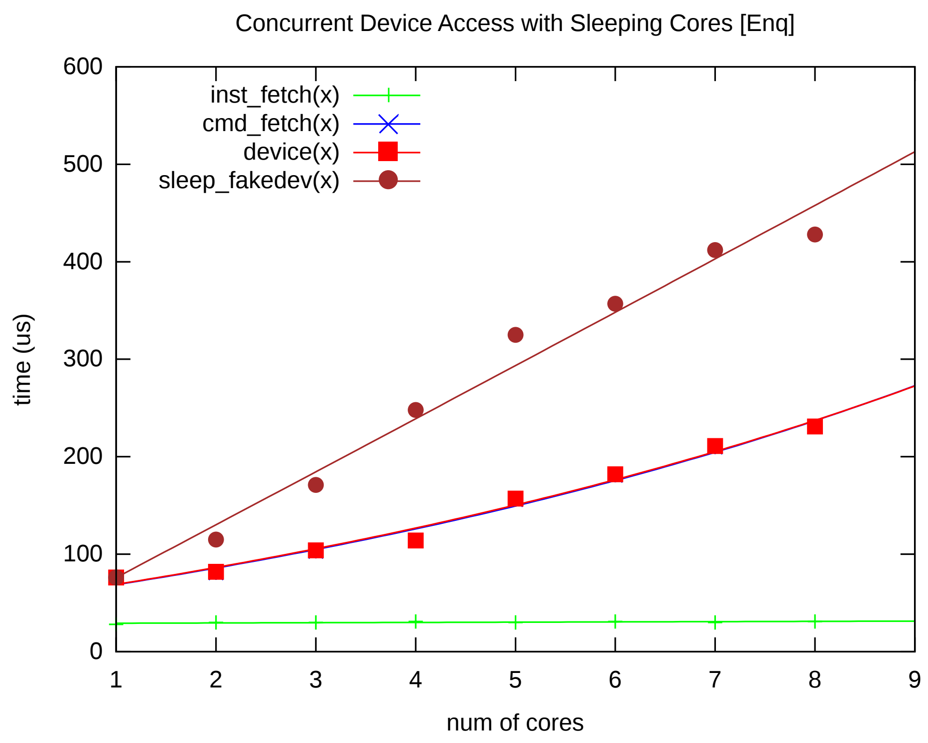


(a) Different Modes of Concurrent Access

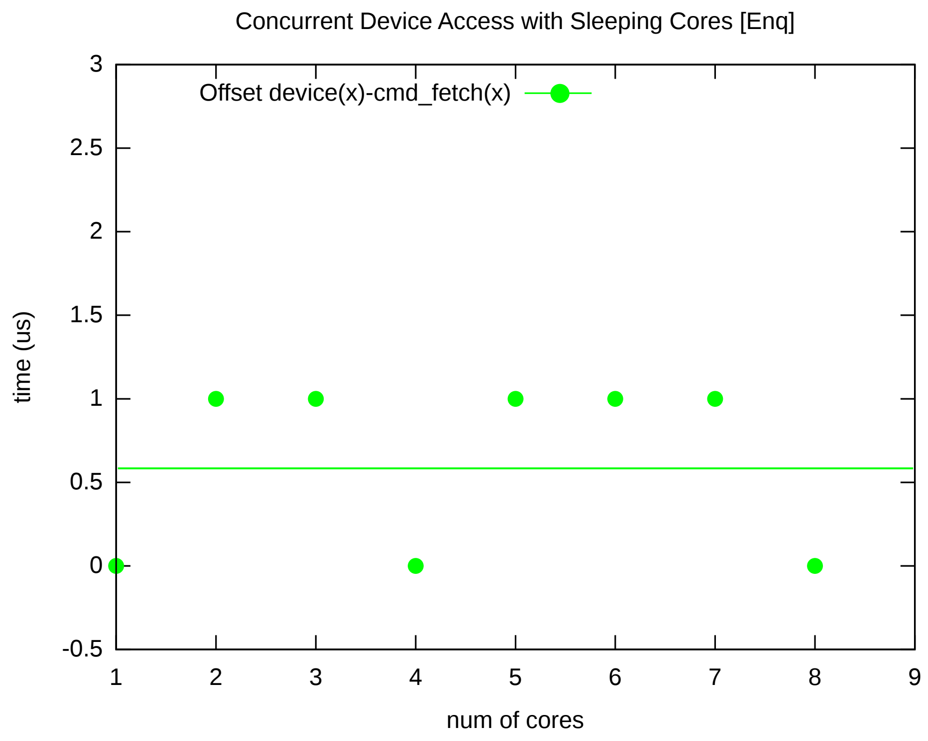


(b) Impact of Device Access only

Figure 6: BMan Measurements



(a) Different Modes of Concurrent Access



(b) Impact of Device Access only

Figure 7: QMan Measurements

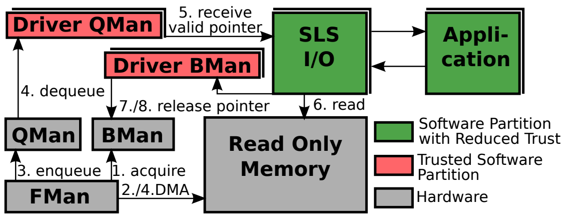


Figure 8: Schematic Architecture with Various SLS I/O Partitions for Receiving Data (the SK is not depicted).