FreeBSD on Freescale QorIQ Data Path Acceleration Architecture Devices

Michał Dubiel, Piotr Zięcik
Semihalf,
md@semihalf.com, kosmo@semihalf.com

Abstract

This paper describes the design and implementation of the FreeBSD operating system port for the QorIQ Data Path Acceleration Architecture, a family of communications microprocessors from Freescale. These chips are modern, multi-core, PowerPC based SoCs, which feature a number of specifically designed peripherals, addressed for the high performance networking devices, which are increasingly common in modern communication infrastructure.

The primary focus is the *Data Path Acceleration Architecture* (DPAA) with the new approach to network interface architecture. It has significant influence on the FreeBSD device drivers design and implementation. The paper describes how the full network functionality was brought forward, and also covers other major development tasks like the e500mc quad-core SMP bring up and support for other integrated devices.

1 Introduction

Increasing number of services available through the Word Wide Network, creates new challenges for the telecommunication industry. Bandwidth requirements are continuously growing, which implies more and more packet processing power. Situation is getting worse, as security concerns has to be taken into account. Modern telecommunication systems have to perform a real-time deep packet inspection, classification and be immune to sophisticated attacks.

To cope with the problems mentioned above, System-On-Chip (SoC) designers have been introducing specially dedicated architectures. These combine a sophisticated hardware modules, which take over more and more processing responsibilities from CPU cores. One example of such an architecture is Freescale QorIQ Data Path Acceleration Architecture, which is the platform for this work.

In order to utilise the facilities brought by those architectures, a new software is necessary. There have already been several advanced, secure and stable operating systems like FreeBSD available for years. It seems obvious to combine their powers with the capabilities of the new hardware.

In this paper, a FreeBSD port for the Freescale QorIQ Data Path Acceleration Architecture SoCs is presented. Design and implementation details, development process and challenges encountered are all given. Also, an overview of the QorIQ DPAA SoC hardware is outlined to help better understand design and implementation decisions made.

This work is based on the existing FreeBSD port for Freescale PowerQUICC III series of SoCs ([E500BSD]). The result is a fully functional FreeBSD system running on Freescale QorIQ DPAA SoCs, which facilitates the DPAA hardware components to achieve significant network performance advance.

2 Hardware

Freescale QorIQ Data Path Acceleration Architecture consists of up to eight Pow-

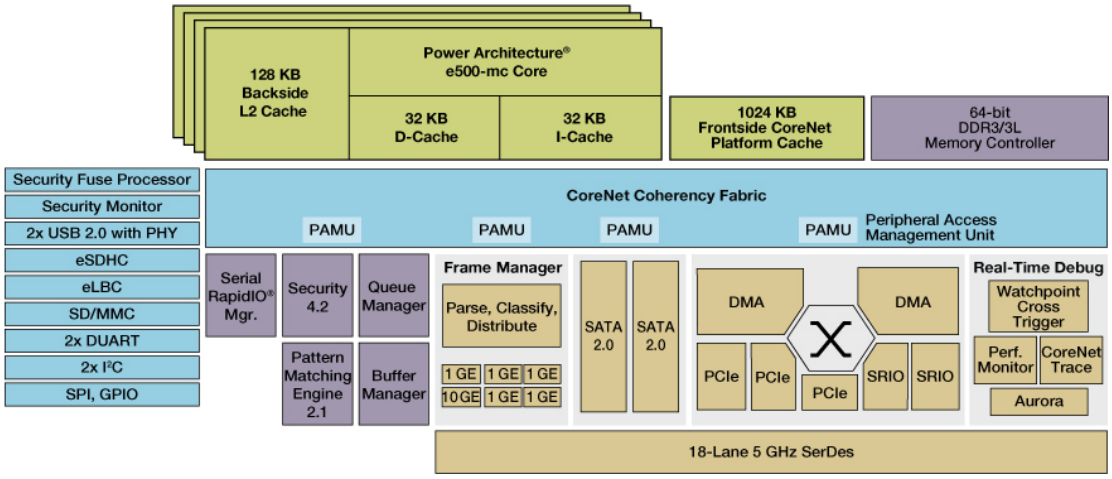

Figure 1: QorIQ P3041 Communication Processor (Source: [P3041FS]).

erPC cores supporting virtualization, security-enhanced interconnect fabric and several hardware components offloading CPUs from packet processing. The Figure 1 presents internal structure of the P3041 SoC, a middle-class member of the QorIQ DPAA family. The following subsections describe the architecture in greater details.

2.1 e500mc core

The e500mc core is a next generation of the e500 core described in [E500BSD] with greater details. The extensive description of the core is out of the scope of this paper and only major improvements of the e500mc core are presented here.

From the perspective of performance, the e500mc core has introduced an integrated L2 cache and an improved FPU. Furthermore, the TLB sizes have been increased. There are 512 constant size entries and 64 variable size entries. The core also come out with a decorated load/store instructions.

The decorated load/store instructions add meta-data (decoration) to the basic load/store operations. This meta-data are used to define additional actions, which may be performed during I/O transactions. For example, the value that is being stored can be added to the actual word located in the memory instead of just overwriting it. The QorIQ DPAA SoCs define several decoration actions providing basic arithmetic and logic operations as well as min/max and accumulation functions.

Another new feature introduced by the e500mc core is an additional execution privilege level (right above the supervisor level present in the classic PowerPC architecture), which extends the virtualization capabilities. Program running on this level is allowed to use another unique feature of the core — external PID load/store. These are special instructions, which execute I/O requests in the different context (defined by hypervisor) than the current running one.

The core also provides better integration within the SoC. Power management of both the core and the SoC is now combined, which reduces programming efforts. Inter-core communication has been improved by adding message-send and message-clear instructions using a new core2core doorbell interrupt.

The last but not the least feature introduced by the e500mc core is cache stashing. With assistance of the SoC interconnect, selected part of any given device ↔ memory

transfer can be placed also in the CPU L2 cache. The stashing mechanism is targeted to accelerate software packet processing. For example, every time when Ethernet Interface receives TCP/IP packet, the first couple of bytes, containing protocol headers, can be stored in the cache. Then, the CPU during packet parsing will fetch data directly from L2 cache instead of performing expensive memory access.

2.2 CoreNet Interconnect Fabric

The e500mc core and all major peripherals are connected with CoreNet Interconnect Fabric, which represents modern Network-on-Chip approach. Its design solves security problems introduced by virtualization. Guest operating systems are not able to access each other directly (due to protections built-in the e500mc core), however they are able to configure a hardware component (such as DMA engine) to access all available memory, including areas occupied by other guest OSes. To avoid such situation, called DMA-attack, hypervisor virtualizes all accesses to the hardware. This simple solution has significant performance impact, which can be avoided by using *Peripheral Access Management Units* (PAMUs) integrated in the CoreNet interconnect.

2.2.1 Peripheral Access Management Unit

The *Peripheral Access Management Unit* (PAMU) is a hardware, through which every I/O and memory transaction passes. Each transaction, characterised by *Logical I/O Device Number* (LIODN) and address, must match a specific set of rules defined by software in the very similar way as MMU entries. The matching rule may allow, deny or redirect the transaction to other destination, as well as modify its cache coherency and stashing attributes. Worth noting is that LIODN consists of two fields. One is hardwired to requesting device, whereas second may be generated by the hardware by using data included in a request. For example, Ethernet controller's DMA may assign specific LIODNs to packets received from given 802.1q VLANs.

2.2.2 Platform Cache

Another interesting component of QorIQ DPAA SoCs is Platform Cache. In difference to CPU caches, the Platform Cache is placed between memory controller and CoreNet interconnect. As a result all requests to memory are accelerated, not only these generated by processors. This reduces memory controller contention by small transfers, such as reception and transmission of small Ethernet packets.

2.3 Data Path Acceleration Architecture

The Data Path Acceleration Architecture is the key component characterising QorIQ DPAA SoC family. It provides an infrastructure for hardware-accelerated packet processing, such as bridging, routing, packet filtering and IPSec processing, to name just a few. The DPAA offloads CPU in four areas: data buffer management, queue management, packet distribution, and policing.

Data Buffer management is realised by a specialised component called *Buffer Manager* (BMan). Data buffers can be allocated and deallocated (by software and hardware components) from user-defined pools. The BMan automatically manages these pools, only signalling depletion state when the number of buffers in the pool falls below predefined threshold level.

Queue management is realised by *Queue Manager* (QMan). Data buffers (not necessarily allocated form BMan) might be organised into frames containing one or more buffers, and then put further into queues, which are fully managed by the QMan. The queues provide an effective data exchange infrastructure between CPUs and/or DPAA components.

The *Frame Manager* (FMan) is responsible for packet distribution and policing. Each frame can be parsed, classified and results might be attached to the frame. This meta data might be used to select particular QMan queue, which the packet is forwarded to, or to

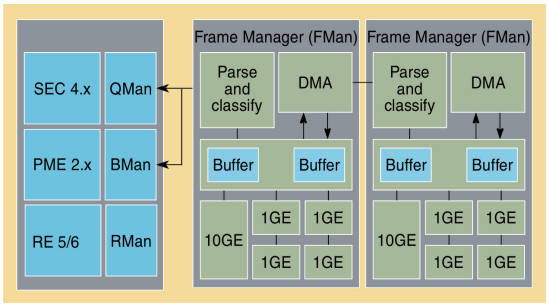

Figure 2: Example of QorIQ Data Path Acceleration Architecture (Source: [DPAARM]).

control flow bandwidth. The FMan supports both online (when packets arrives from integrated MACs) and offline (when packets are taken from memory) processing.

The Figure 2 presents the interactions between the DPAA components. Some of them are described in greater details in the following sections.

2.3.1 Buffer and Queue Manager

As it was mentioned, the BMan holds and manages pools of data buffers. Each buffer is represented only by a pointer. To allocate the buffers, software just reads the pointers from special set of registers (called portals). Deallocation is just an opposite: software has to write the pointers to the portal. There are up to ten portals, which can be assigned to the CPU core and/or application. As accesses to the portal are atomic, no locking is required as long as the portal is not shared. Such mechanism eliminates lock contention, which greatly degrades performance in multi-core systems. The same access method applies to Queue Manager. Putting a frame into a queue is in fact writing of a frame descriptor into a QMan portal.

Portals are also able to signal various conditions by raising an interrupt. For example, QMan portals asserts interrupt when there are more than predefined number of frames in the queue, or when a frame is being held in the queue over a specified time.

2.3.2 Frame Manager

Frame Manager is central and the most complex part of the Data Path Acceleration Architecture. It is responsible for packets reception, transmission, parsing, classification and finally routing them into relevant QMan queues. Simply speaking, it is the heart of a hardware packet processing. The above tasks are fulfilled by dedicated FMan sub modules, described later on in this subsection.

There are several 1Gbit/s and 10Gbit/s Media Access Controllers (MACs) integrated into the Frame Manager, each associated with single physical Ethernet interface. Its numbers vary from one SoC version to another, but in most cases there are five 1Gbit/s and one 10Gbit/s interfaces. Those are directly responsible for packet transmission and reception over the media.

Whereas FMan MAC connects FMan to network, Buffer Manager Interface (BMI) and Queue Manager Interface (QMI) sub modules provide communication means with the Buffer Manager and the Queue Manager respectively. Those are necessary, as full potential of the Data Path Acceleration Architecture can only be exploited while using all BMan, QMan and FMan components cooperating with each other.

FMan MACs, BMI and QMI sub modules offer packet frames interchange between the Frame Manager, network and other SoC modules like CPU cores, *Security Engine* (SEC), etc. However, as it was mentioned earlier, Frame Manager allows also for packet parsing, classification and policing. These features are provided by FMan Controller, Parser, KeyGen, Policer and Frame Processing Manager (FPM).

A central unit responsible for hardware frames processing inside the FMan is Frame Processing Manager. It distributes frame processing tasks amongst the other FMan sub modules, where those are processed next.

The FMan Controller, Parser, KeyGen and Policer form so called Parse Classify and

Distribute (PCD) flow. The Parser is used to identify the incoming frames. It recognises many standard protocols and also allows for custom and/or future protocol parsing. Its output, called parse result, is then used by classification modules.

There are two categories of frame classification: hash based and table lookup. The former is realised by the FMan KeyGen sub module. Hashing can be performed from many different fields in the frame and/or from the parse result. The second, table lookup, is performed by the FMan Controller. It looks up certain fields in the frame, which are selected by the combination of user specified configuration and what fields the FMan Parser actually encountered. The classification of frames (either hash based, table lookup or both) determines a subsequent action to take.

An example of such an action might be frame policing, performed by the FMan Policer. It supports many policing profiles based on two-rate, three-color marking algorithm ([RFC2698]). Burst sizes, sustained and peak rates are all user-configurable.

Finally, results obtained from the FMan sub modules described above are used to select a queue to which a frame is to be enqueued. Those queues might be associated with different modules within the SoC (CPU core, Security Engine, etc.), which dequeues the frames and perform further processing.

For example, the FMan MAC receives a frame and forwards it to the FMan Parser. After parsing, it turns out that the frame is encrypted. An encryption keys are then attached to the frame and it is send to the Security Engine, where it is decrypted without using any of the CPU cores. Next, the already encrypted frame is given back to the FMan for further classification, policing and then send to either the CPU core for software processing or transmitted elsewhere through the network.

Each FMan sub module is highly configurable. Many of its functionalities can be tuned in a run time, which makes the FMan a powerful hardware frame processor.

3 Software

Having a first-class hardware is worth nothing without a high quality software. In this chapter, process of porting the FreeBSD to the QorIQ DPAA devices and challenges in the DPAA integration with the OS's TCP/IP network stack are presented. The whole process, step by step, starting from toolchain adjustment and ending with fully functional FreeBSD system utilising facilities of the QorIQ DPAA platform is described in the following subsections.

3.1 Toolchain

Introduction of the new e500mc core has implied a necessity of toolchain adaptation. The existing FreeBSD PowerPC code had already supported the e500mc predecessor, thus only addition of a few new instructions was required. To fulfil this, appropriate changes to both binutils and gcc were applied. Fortunately, the e500mc patch for binutils had been already available through the community and gcc patch had been delivered to us by SoCs supplier — Freescale.

3.2 Early kernel initialisation in *locore.S*

The *locore.S* contains an assembly, architecture dependent code, which is executed at the very beginning of the FreeBSD start up. The main goal of this code is preparation of the environment for C-code execution.

In PowerPC architecture, the *locore.S* is responsible for TLB and kernel stack initialisation. This topic is well described for the e500 core in [E500BSD], in the example of MPC8572 SoC bring up. This subsection focuses only on the changes introduced by the new e500mc core.

The first thing that differs e500mc from its predecessors is a bigger Transaction Lookaside Buffer (TLB). This has implied slight

changes of bit-fields length in registers controlling the TLB. Because the *locore.S* code uses these registers in multiple places, this small hardware change has enforced thorough line by line analysis and many adjustments in the assembly code.

Besides the bigger TLB, new hypervisor privilege level present in e500mc core has introduced new MMU assist registers (MAS). These registers have to be set accordingly during each TLB access. Simply adding direct references to those new registers would broke backward compatibility, as accesses to non-existent registers are causing exception and eventually system hang-up (exceptions in this stage can not be handled yet). To cope with this, special procedures were added (`zero_mas7` and `zero_mas8`). This routines perform a run-time core identification and write to the MAS registers (*MAS7* and *MAS8* respectively) only if the running processor actually has those. Analogous tweaks were necessary to handle Hardware Implementation-Dependent Registers' (HIDs) accesses correctly.

The machine-dependent portion of virtual memory subsystem (*pmap(9)*) also had to be slightly changed. However, as the most *pmap(9)* modifications are related to Symmetric Multi-Processing, they have been described in appropriate chapter: 3.4.

3.3 QorIQ Data Path Acceleration Architecture

As OpenPIC and UART drivers had already existed in the FreeBSD, the next major task was networking bring up. Because of the SoC architecture, this involved creation of device drivers for variety of DPAA components, including the most complicated one — the Frame Manager.

During the research, preceding drivers development, it was found that effort could be greatly reduced by using drivers from Freescale NetCommSw software pack. The NetCommSw is a packet processing framework, able to run on a bare-bone hardware. It consists of low-level OS agnostic device drivers, protocol stacks and development tools, everything targeted to rapid development. Unfortunately, this software pack is proprietary licensed.

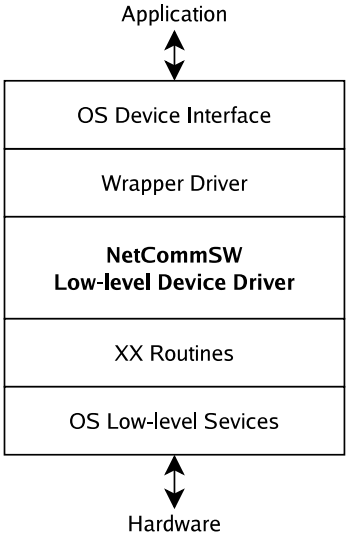

Figure 3: NetCommSw Driver model.

However, it was found that one part of the NetCommSw — the Frame Manager driver, had been already available under the BSD license elsewhere. Utilising a ready to use driver for the most complex DPAA hardware component, greatly reduced the development time. Moreover, Freescale, who is the owner of the NetCommSw software, agreed to release additional parts of this suite under the BSD license.

In the end, all necessary for this work NetCommSw Low-Level Drivers (Buffer Manager, Queue Manager and Frame Manager) were available under the BSD license and ready to integrate into the FreeBSD Project.

3.3.1 NetCommSw drivers integration

All NetCommSw drivers share the same programming model, shown on the Figure 3, which is optimised for easy integration with various operating systems. In the centre of the model the Low-level Device Drivers lie. The

drivers export high-level device-specific object-oriented API abstracting device functionalities. Only the middle part of this model, the Low-level Device Drivers, had been provided by Freescale. Both Wrapper Drivers and XX Routines had to be implemented.

All accesses to the hardware are performed through several operating systems hooks called XX Routines. Those are also used to obtain OS resources like locks and memory, as well as to register interrupt handlers. The XX layer makes the drivers fully OS-agnostic.

A Wrapper Driver, placed over the Low-level Device Driver, is also required. It is responsible for attachment to the operating system and translation of OS calls to the device-specific Low-level Device Driver API. At this level all accesses to the hardware have to be serialised. This implies lock utilisation.

The XX Routines interface is quite straightforward and well defined. Mapping it to the FreeBSD kernel's internals was not a hard task (what should be a body of the `XX_Malloc()` function?). Nevertheless, not all functionalities required by this layer had been already available in the FreeBSD kernel. The most clear example is physical to virtual address translation.

Such translation is not used and thus not supported in the FreeBSD kernel at all. Moreover PA to VA mapping is ambiguous, as multiple virtual pages might be linked with one physical page. To deal with this problem a list of all active mappings referencing given physical page has been added to the machine-dependent part of `vm_page` structure. Each time, when VA to PA mapping is created, the `pmap_enter()` function adds new entry to the list. Alike, the entry is removed when given mapping is destroyed in `pmap_remove()`.

The XX Routines use a given physical address to get the appropriate `vm_page` structure and then utilise it to obtain a list of active translations. In case of multiple mappings, only the first found (last added) entry is used to extract virtual address.

The other class of problems found during XX Routines implementation was this related to the Symmetric Multi-Processing. Those are described in appropriate section 3.4.1 of this article.

Writing of the Wrapper Drivers, which are just newbus compliant attachments, was almost effortless. However, as well as in the case of XX Routines the SMP implementation required additional work, also described in section 3.4.1.

3.3.2 Frame Manager driver

Integration of Frame Manager Driver required a special care. Unlike Buffer Manager and Queue Manager, the FMan consists of several subsystems (described in section 2.3.2). Each of them is represented by separate APIs in the NetCommSw Frame Manager Low-level Driver. Because of that, no single Wrapper Driver was created, but instead some sort of software partitioning was applied.

At first, the common part used by all sub modules was implemented as a single driver in the FreeBSD newbus approach. It is responsible for early Frame Manager initialisation and also manages internal FMan resources used by sub module's specific drivers.

Every driver utilising any of the Frame Manager capabilities, has to acquire resources from the FMan driver. One of the example is a *Datapath Triple-Speed Ethernet Controller* (dTSEC) driver, which implements the entire network functionality.

3.3.3 Datapath Triple-speed Ethernet Controller driver

From a perspective of operating system, the dTSEC is a classical network device driver. It implements a set of special networking interfaces defined by the FreeBSD kernel, extensively utilising the following DPAA components: Buffer Manager, Queue Manager and Frame Manager.

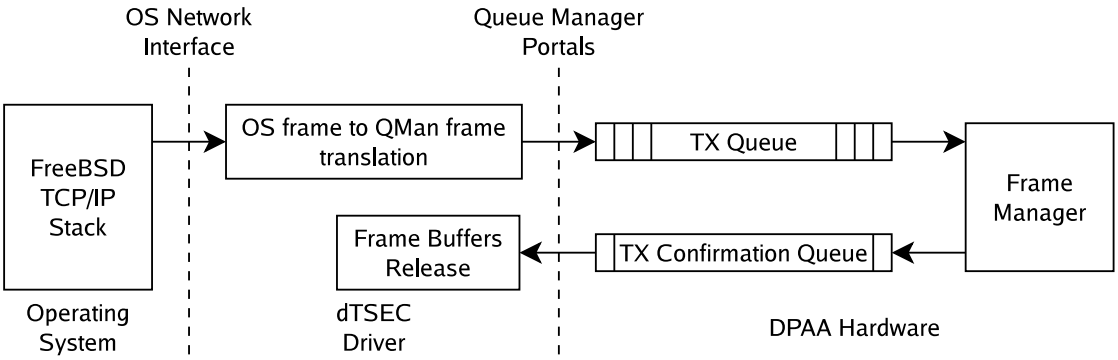

Figure 4: dTSEC packet transmission data flow.

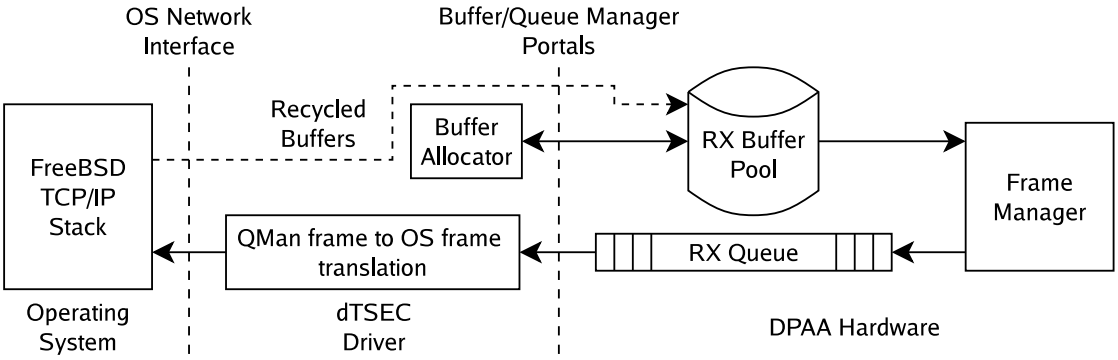

Figure 5: dTSEC packet reception data flow.

First thing the dTSEC driver does, is the configuration of FMan sub modules: BMI, QMI, FPM and MAC. Those combine a minimal set of required components that have to be initialised in order to be able to send and receive Ethernet packets. All these units are fully abstracted by two NetCommSw Frame Manager Low-Level Drivers' interfaces: FMan MAC and FMan Port.

The FMan MAC interface is used to communicate with the MAC sub module only, whereas FMan Port interface abstracts accesses to the FPM, BMI and QMI. It is, hence, the only way to control data flow from and to the Frame Manager. Moreover, each instance of the FMan Port interface is able to handle either the transmission or the reception flow and can be associated with exactly one MAC. As a consequence, each Ethernet network interface driver has to utilise at least two FMan Ports and one FMan MAC instances.

But this is not enough to exchange network packets with the MAC. Since Frame Manager uses BMI and QMI for all accesses to system memory, Buffer Manager and Queue Manager have to be also utilised by the dTSEC driver. For example, received frames' data is written to memory buffers provided by Buffer Manager and simultaneously represented as a new frame enqueued to the Queue Manager.

On the transmission path, the dTSEC driver creates two QMan Frame Queues, and attaches them to FMan TX Port. The first queue, called Transmission Queue, is used to feed the Frame Manager with a data to be transmitted. The driver only enqueues frames

into the queue. Nothing else is necessary. Transmitted frames are then returned through the second queue, called Transmission Confirmation Queue. After checking the status code for errors, the driver frees memory associated with the transmitted frames.

On the reception path, the dTSEC driver creates only one QMan Frame Queue and one Buffer Pool managed by the Buffer Manger. Both the Frame Queue and the Buffer Pool are then associated with the FMan RX Port. After the FMan MAC receives a packet, a free Buffer is allocated from the given Buffer Pool to hold the received data. Then, a frame referencing the buffer is created and enqueued to selected Frame Queue. The driver job is to only fetch frames from the Frame Queue associated with the FMan Rx Port.

Quantity of free buffers in the Buffer Pool is maintained by the Buffer Manager itself. When a number of the buffers in pool falls below the defined depletion threshold, an interrupt is asserted. As a result the BMan driver notifies the dTSEC driver, which uses *uma(9)* zone allocator to provide fresh memory buffers. Buffers used by received packets are released back to the Buffer Manager Pool, unless it has predefined number of buffers. Otherwise, the recycled buffers are freed back to *uma(9)*.

Described above data flow is illustrated on the Figure 4 and the Figure 5.

As all data between dTSEC driver and hardware transfers are performed through specialised BMan an QMan portals, briefly described in 2.3.1, the software overhead is negligible. Moreover, the performance is further improved by interrupt throttling embedded into the Queue Manager.

3.3.4 DPAA in newbus hierarchy

As was mentioned earlier, all DPAA components had been implemented as newbus compilant drivers. Their layout in driver hierarchy reflects SoC memory map. The drivers are attached and initialised by applying depth-first tree search algorithm to this structure. This stays in conflict with DPAA hardware initialisation requirements.

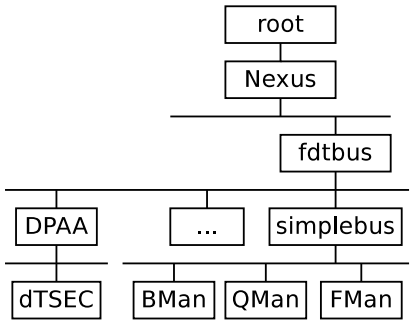

Figure 6: DPAA in newbus hierarchy.

To cope with this situation a virtual bus named DPAA was introducted. The drivers utilising more than one DPAA components (BMan, QMan and FMan) had to be placed under this bus (see the Figure 6 for details), which ensures proper initialisation order.

3.4 SMP bring up

The QorIQ Data Path Acceleration Architecture SoCs are equipped with up to eighth e500mc cores. The possibility of utilising its power using Symmetric Multi-Processing, gives system designer an additional advantage.

The FreeBSD had already supported SMP on the e500mc core predecessors (see [E500BSD]). As differences introduced by the e500mc core are not directly related to multiprocessor support, it seemed that enabling the SMP support would not require extra effort.

However, existing model of *Inter Processor Interrupt* (IPI) handling had not taken into account hardware limitations. It worked well with up to two cores, but adding new one disturbed inter-core communication. When an IPI was send to multiple cores, only one of them actually received the message.

Investigation showed that IPI messages were send one-by-one through single communication channel without its availability checking.

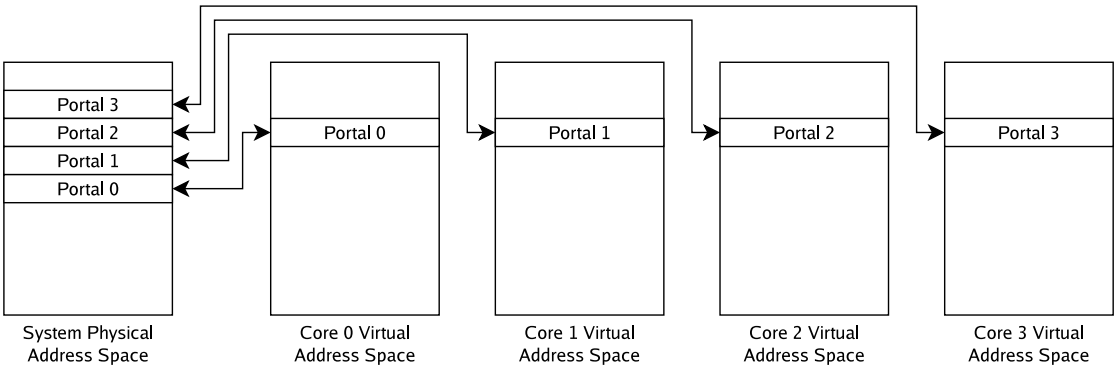

Figure 7: DPAA Portal mappings.

As the hardware treats communication channel as busy until target core acknowledges message reception, only the first message in the burst was actually delivered.

To cope with this problem IPI multicasting was implemented. Instead of sending IPI messages to multiple cores sequentially, a one multi-cast message is used. This change allowed for successful bring up of quad-core SMP.

Nevertheless, the integration of SMP with Data Path Acceleration Architecture enforced further deep changes in both the FreeBSD and NetCommSw drivers.

3.4.1 DPAA in SMP environment

As it was described in 2.3.1, accesses to Buffer and Queue Managers are performed through special registers called portals. While the accesses from different CPU cores are performed through separate portals, there is no need for serialisation. This approach eliminates bottleneck caused by lock congestion in SMP environment.

In order to utilise this advantage, it was decided to assign one dedicated portal to each CPU core. The core $\leftrightarrow$ portal mapping is transparent to CPU. Each portal occupies the same virtual address on each core (see Figure 7) for details).

However, existing FreeBSD SMP support for e500 family allowed for only one, shared by all cores, device mapping. Originally, all device mappings were simply copied from bootstrap CPU to application CPUs. This approach has been partially reused. Now, only mappings marked as shared are copied. The mark is located in special user-defined bits, available in TLB entries of the whole e500 family. This simple change required significant modification of *pmap(9)* part responsible for device mappings, as well as CPU start-up procedures.

The new approach has also implied changes in NetCommSw Wrapper Drivers. When FreeBSD drivers are attached, only the bootstrap CPU is alive. Thus, mappings for the application CPUs can not be set at this stage. Beside that, the driver also has to configure each mapped portal.

In consequence, after FreeBSD start up, only one portal attached to bootstrap CPU is accessible. But, as all cores are running at this stage, accesses to drivers might be initiated from any of them. Hence, the Wrapper Drivers are responsible to map and configure portals during the first access from any of the application CPUs.

However, run-time portal creation caused a new problem to appear. From NetCommSw Wrapper Driver perspective, portal configuration is reduced to just few calls to NetCommSw Low-Level Driver. Then, Low-Level

Driver, uses XX Routines to obtain OS and hardware resources. One of the types of allocated resources is an interrupt. In the FreeBSD, requesting an interrupt might sleep in `intr_event_create()`, hidden deeply in the operating system. As portal might be created at any time, the sleep during interrupt request introduces several locking issues.

For example, the FreeBSD TCP/IP stack might send a packet to network drivers holding a lock related to network protocol. If sending the packet requires portal configuration on any of the application processors, then sleep with lock held may occur, which is prohibited.

To deal with this problem, an additional layer for interrupt management in XX Routines has been introduced. Interrupt registration calls from NetCommSw Wrapper Drivers are not directly translated to its FreeBSD counterparts, but instead obtained from the new layer, which preallocates them during the system start up.

Nevertheless, the interrupt allocation is not the only problem here. An interrupt, asserted by any given portal, might be scheduled for execution on any of the processor cores. As each core has access to only single dedicated portal, the incorrectly scheduled interrupt routine will service other portal than this assigned to it. Thus, the additional interrupt management layer in XX Routines has been also made responsible for binding the interrupt threads to proper CPU cores.

4 Other peripherals bring up

Besides the e500mc core, SMP and DPAA, support for other peripherals like PCI Express, USB EHCI controller, I2C controller and internal SoC's DMA was added. However, these devices had been already well supported in the FreeBSD kernel and bring up effort was reduced to proper assignments of system resources with those drivers only.

5 Current state and results

Currently, the FreeBSD fully supports three members of a QorIQ Data Path Acceleration Architecture family: P2041 (quad core, 1.2 GHz), P3041 (quad core, 1.5 GHz) and P5020 (dual core, 2.0 GHz). Besides the DPAA subsystem, there are ready to use drivers for the USB 2.0 EHCI, SD/MMC Controller and other peripherals like UARTs, I2C and DMA controllers. The built-in PCI Express is also supported.

Network tests, has shown that the DPAA architecture significantly reduces CPU utilisation related to packet processing. On P3041 SoC, the iperf tool (`# iperf -c <iperf-server> -l 16M -w 128k -t 60`) gave 897 Mbit/s transfer rate through single Ethernet interface. The CPU utilisation during the test was below 30%. The 90% of this load was caused by interrupt servicing and could be reduced by enabling polling mode, which has not been implemented yet.

The following listing, showing the FreeBSD interrupt counters, lay out almost equal distribution of packets between different Queue Manager portals (interrupts 120-126) and thus CPU cores respectively.

```
p3041# vmstat -i
interrupt          total       rate
irq121: bman0      1           0
irq120: qman0      2784930     3584
irq122: qman0      2194130     2823
irq124: qman0      2263079     2912
irq126: qman0      2148167     2764
(...)
```

Also, a small number of interrupts asserted by Buffer Manager shows that memory buffers management is performed entirely by the hardware and software intervention is not necessary (the one, visible interrupt is asserted during initial Buffer Pool creation).

6 Future work

Although current state of the FreeBSD QorIQ DPAA port is stable and runs on the majority of the QorIQ DPAA family SoCs, support for some integrated peripherals is still missing. This includes built-in SATA controller and few members of the DPAA subsystem like *Pattern Matching Engine* (PME) and Security Engine.

Also not all features of the DPAA, that could be used by the FreeBSD TCP/IP stack are available. The pooling mode, hardware packet checksumming and Jumbo Frames are not supported.

Worth trying would be integration of advanced Frame Manager capabilities, like fully hardware IPSec processing with the FreeBSD TCP/IP stack.

7 Conclusions

The FreeBSD can be successfully used on the QorIQ Data Path Acceleration Architecture SoCs. It is also able to utilise majority of the features available in the hardware. Although the full hardware packet processing provided by those SoCs is not supported, which would require deep changes in the FreeBSD TCP/IP stack, the overall network performance achieved in this work is promising.

8 Summary

In this paper, the FreeBSD port for QorIQ Data Path Acceleration Architecture SoCs was presented.

First, an overview of the QorIQ DPAA SoC hardware was given, including the new e500mc core and CoreNet Interconnect. Special attention was put on the DPAA components: Buffer Manager, Queue Manager and Frame Manager.

Next, process of porting the FreeBSD to the QorIQ DPAA devices, along with the challenges encountered during the DPAA integration with the OS's TCP/IP network stack were presented. This included new toolchain adaptation, early kernel initialisation (*locore.S* part), DPAA components' drivers implementation and Symmetric Multi-Processing bring up.

Finally, the result of the porting process was given. It consists of a fully functional FreeBSD system running on P3041 SoC and utilising the facilities of the Data Path Acceleration Architecture to achieve considerable network performance boost.

9 Acknowledgements

Special thanks go to the following people:

Rafał Jaworowski (Semihalf, The FreeBSD Project), mentor of this project.

Phil Brownfield (Freescale), for help and support with relicensing the NetCommSw code and device tree source files.

Zbigniew Bodek, Piotr Nowak, Tomasz Nowicki, Jan Sięka, Łukasz Wójcik (all Semihalf), for all the work on this project.

Work on this paper was sponsored by Semihalf.

10 Availability

The code described in this paper is (or will soon be) available from the FreeBSD Project Subversion repository, CURRENT branch.

References

[E500MC] Freescale Semiconductor, Inc., *e500mc Core Reference Manual*, Rev. 0 09/2011

[DPAARM] Freescale Semiconductor, Inc., *QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual*, Rev. 2 11/2011

[P2040RM] Freescale Semiconductor, Inc., *P2040 QorIQ Integrated Multicore Communication Processor Family Reference Manual*, Rev. 0 11/2011

[P3041FS] Freescale Semiconductor, Inc., *P3041 Fact Sheet*, Rev. 3 08/2011

[P3041RM] Freescale Semiconductor, Inc., *P3041 QorIQ Integrated Multicore Communication Processor Family Reference Manual*, Rev. 0 11/2011

[P5020RM] Freescale Semiconductor, Inc., *P5020 QorIQ Integrated Multicore Communication Processor Family Reference Manual*, Rev. 0 11/2011

[RFC2698] J. Heinanen, Telia Finland, R. Guerin *A Two Rate Three Color Marker*, 1999

[E500BSD] Rafał Jaworowski, *FreeBSD on high performance multi-core embedded PowerPC systems*, 2009