FPGA and Server Integration in a Small Form Factor Data Center
Intermediate Version

Co-funded by the Horizon 2020
Framework Programme of the European Union

DELIVERABLE NUMBER	D6.5
DELIVERABLE TITLE	FPGA and Server Integration in a Small Form Factor Data Center – intermediate
RESPONSIBLE AUTHOR	IBM

GRANT AGREEMENT N.	688386
PROJECT REF. NO	H2020- 688386
PROJECT ACRONYM	OPERA
PROJECT FULL NAME	LOw Power Heterogeneous Architecture for Next Generation of SmaRt Infrastructure and Platform in Industrial and Societal Applications
STARTING DATE (DUR.)	01/12/2015
ENDING DATE	30/11/2018
PROJECT WEBSITE	www.operaproject.eu

WORKPACKAGE N. | TITLE	WP6 | Low Power Small Form Factor Datacentre
WORKPACKAGE LEADER	NALL
DELIVERABLE N. | TITLE	D6.5 | FPGA and Server Integration in a Small Form Factor Data Center – intermediate
RESPONSIBLE AUTHOR	J.Nider (IBM)
DATE OF DELIVERY (CONTRACTUAL)	31/05/2017
DATE OF DELIVERY (SUBMITTED)	31/05/2017
VERSION | STATUS	V4 | Final
NATURE	R(Report)
DISSEMINATION LEVEL	PU(Public)
AUTHORS (PARTNER)	NALL, HPE

VERSION	MODIFICATION(S)	DATE	AUTHOR(S)
1	First Draft	16/04/2017	J.Nider (IBM)
2	Second Draft	07/05/2017	J.Nider (IBM)
3	Integrated comments from reviewers	17/05/2017	G.Renaud (HPE), R.Chamberlain (NALL)
4	Final version	29/05/2017	J.Nider (IBM)

PARTICIPANTS		CONTACT
STMICROELECTRONICS SRL		Giulio Urlini Email: Giulio.urlini@st.com
IBM ISRAEL SCIENCE AND TECHNOLOGY LTD		Joel Nider Email: joeln@il.ibm.com
HEWLETT PACKARD CENTRE DE COMPETENCES (FRANCE)		Gallig Renaud Email: gallig.renaud@hpe.com
NALLATECH LTD		Craig Petrie Email: c.petrie@nallatech.com
ISTITUTO SUPERIORE MARIO BOELLA		Olivier Terzo Email: terzo@ismb.it
TECHNION ISRAEL INSTITUTE OF TECHNOLOGY		Dan Tsafrir Email: dan@cs.technion.ac.il
CSI PIEMONTE		Vittorio Vallero Email: Vittorio.vallero@csi.it
NEAVIA TECHNOLOGIES		Stéphane Gervais Email: s.gervais@lacroix.fr
CERIOS GREEN BV		Frank Verhagen Email: frank.verhagen@certios.nl
TESEO SPA		Stefano Serra Email: s.serra@teseo.clemessy.com
DEPARTEMENT DE L'ISERE		Olivier Latouille Email: olivier.latouille@isere.fr

ACRONYMS LIST

ASIC	Application specific integrated circuit
ASID	Address space identifier
CAPI	Coherent accelerator processor interface
FPGA	Field programmable gate array
NIC	Network interface card

LIST OF FIGURES

EXECUTIVE SUMMARY

D6.5 reports the efforts of task T6.5 – about the integration of the FPGA into a small form factor data center, represented by the POWER8 server. We have chosen to use the FPGA to implement a low latency network interface card dedicated to serving remote memory page faults. This is important for two reasons – first, we show how to use the FPGA developed by NALL to accelerate a network function (in this case, serving remote page faults) which saves power over traditional networks in use today. Second, the network itself is used to accelerate the cross-ISA container migration use case, which is used to further save power by integrating heterogeneous compute resources.

We discuss the various design points of the network protocol required to support remote page faults, and discuss the implementation details that are known at this point in the project. We touch on the measurement methodology (but details will be available in a later deliverable), and the future of such implementations with the development of several open protocols that are becoming available.

Position of the deliverable in the whole project context

This relates to the other tasks in WP6 through the use of the FPGA card being developed by NALL. We have taken the same card developed for other use cases (such as the video analysis in the "truck" use case) and repurposed it to become a network interface card. This highlights the versatility of the platform, and the flexibility of the design to handle multiple real-life workloads.

The work is also related to WP4 in which power measurements are taken of the FPGA performing the container migration, and serving page faults. This is an important task to reach the goals of the OPERA project by demonstrating how to perform work more efficiently.

There is also a relation to WP5 which focuses on workload decomposition and the container migration mechanism needed to provide the flexibility to a PaaS cloud data center. Through the mechanism developed in WP5, we will showcase the strength of the FPGA card by performing container migrations with practically no downtime, and practically no impact the customers who rely on the cloud services.

Description of the deliverable

The deliverable describes the specifics of the design of the FPGA firmware code, and the networking protocol and explains the decisions that were taken along the way. We go into detail on how the FPGA can accelerate remote page faults.

List of actions and roles

LIST OF ACTIONS				
ACTIVITIES LIST AND PARTNERS ROLES	IBM	NALL	HPE	NEAVIA
Summary of the initial requirement	P	P	P	
Components to be integrated	P	I	R	
Design of components	P	R	R	
Writing of the deliverable	P	R	R	R

- P = Participating (includes I & R)
- I = Input delivery (Includes R)
- R = review

IBM is the main author of this deliverable. They are the main contributor to the task T6.5 and are responsible for the design of the networking protocol, and ultimately for the design of the FPGA firmware that will implement it.

NALL leads the work package. They are responsible for the design and requirements of the FPGA card, and led the effort to write the specifications that would allow us to use the FPGA successfully. NALL contributed to the writing of this document, and reviewed it for completeness and correctness.

HPE contributed to the task by providing technical details regarding the Moonshot server, and sharing their experiences with the planned integration of the FPGA card into the Moonshot server. HPE also reviewed this document, and helped shape the direction of the task.

Neavia reviewed the deliverable, and provided comments leading to the final version.

TABLE OF CONTENTS

1 INTRODUCTION

FPGA technology (field programmable gate array) is one of the most powerful concepts in computer engineering today. On one hand, it is essentially hardware - composed of thousands (or even millions) of small, generic components that can operate at a very high speed, with low power consumption to implement practically any design imaginable. On the other hand, it is programmable - generic components that can be combined in a non-permanent way to bring to life designs that could until recently only be implemented in ASIC (application-specific integrated circuit) technology at a huge cost in both time and tooling. These features make FPGAs the ultimate prototyping tool - designs can be tested at a fraction of the cost before being implemented in ASICs, and multiple revisions can be produced and verified in a fraction of the time. FPGAs are also the ultimate application accelerator, since they are programmable they can be used to implement algorithms that run thousands of times faster than they would on a generic CPU.

In T6.5, we have chosen to show the versatility of the FPGA by taking the same card developed by NALL (as described in deliverable D6.3) used as an accelerator in other tasks, and repurposing it as a prototyping tool. We have chosen to prototype a NIC (network interface card) that can be used to move memory pages between two physical servers during VM or container migration. If this design is proven successful, we can potentially reuse the design with very few modifications for inclusion in a future ASIC for mass production.

The case for migration of execution contexts (virtual machines, containers, processes, etc) has been made in the past, and support for migration is very common in various hypervisors and orchestration layers in use today. The case for post-copy migration has been made, as post-copy saves network bandwidth, and down-time of essential applications. The case for cross-ISA container migration has recently been made in academia, and we are aiming to bring this case to industry with a real use case. To support fast migration, it is necessary to reduce the latency of page faults to minimize the impact on running applications. By designing a low latency dedicated network, we aim to reduce the latency to the point that page fault latency is practically negligible.

We rely on the state-of-the-art FPGA technology in order to implement such a prototype network interface. Nallatech's 385a-SoC card will enable us to measure the latency to provide evidence of the viability of this solution, with the hopes that it will be adopted later in future system designs.

2 USE CASE

2.1 VIRTUAL DESKTOP

We have chosen to apply the FPGA technology to optimizing the virtual desktop (VDI) use case. The power consumption of the VDI use case is already being optimized in a couple of ways (powering off under-utilized servers, application of heterogeneous hardware to best fit the execution requirements) which are enabled by breaking the applications down to their components, and serving these components as containerized microservices (as detailed in D5.1).

The VDI use case integrates a heterogeneous mix of server architectures (ARM, POWER8, x86) which gives us the opportunity to try multiple configurations of hardware to see what works best. Our goal for the first iteration is to connect servers of the same architecture in order to prove the functionality of the interface cards and the associated techniques without the additional complexity of dealing with multiple architectures. Once the design and implementation mature to a point where we can run real workloads and perform measurements, we will move to include additional architectures for a true heterogeneous mix of machines in a single data center.

2.2 POST-COPY CONTAINER MIGRATION

We can further optimize the movement of the microservices by applying the post-copy migration technique. Post-copy migration has already been used successfully for processes [1] and VMs [2] and our recently posted patches to the CRIU [3] project which were developed as part of OPERA have been accepted upstream. Post-copy migration means moving the bulk of the memory image on demand, once the context has already be restored on the target system. Post-copy container migration allows reduction of application downtime and reduces overall network bandwidth used for application migration. A detailed description of post-copy can be found in D5.3.

2.3 WEAKNESSES

While post-copy greatly improves container migration efficiency, it still has inherent weaknesses that should be addressed. The main weakness is the latency involved when a restored process requests a memory page that has not yet been migrated. In this case, the request must be sent to a remote machine, which requires two-way communication over a network. In addition, the page must be mapped into the address space of the calling process once it arrives, involving some operations of the kernel to update the page tables and internal memory management structures. We aim to address this weakness by applying the FPGA technology to build a low-latency link for moving memory pages with minimal impact on process execution.

An additional weakness that should be mentioned is the lowered resiliency to hardware failures. It is possible that failure of either the network or source machine during the migration can cause the loss of the migrated context. Let us take for example the case of a context migration in which the initial stage of the migration has completed, and the context has resumed execution on the target machine. At this point, the only remaining data on the source machine is some memory pages, which are required for continued execution of the context. If the source machine fails or becomes unavailable (network failure) the memory pages will not be accessible to the executing context in the case of a page fault. It is highly likely that execution of the context would need to be halted at this point. If the network failure is temporary, the executing context can potentially be resumed when the network returns. If the failure is more catastrophic (such as the source machine losing power or rebooting) there is no chance for recovery of the executing context. We do not attempt to address this weakness of post-copy container migration in the scope of the OPERA project, as it is strongly related to resiliency and availability, and not at all related to power consumption, thus it is beyond the scope of this research.

3 DESIGN

The key to moving memory between servers on demand during live migration is the ability to react to a page fault, and service the fault by copying the memory from a remote machine. To handle a remote page fault, there must be a common addressing scheme in place to be able to identify the correct memory page that should be copied. The two key pieces of information are the *memory address* and the *address space*.

The memory address is simply the virtual address that caused the page fault. Since we are essentially dealing with a single process[1] that is distributed among multiple servers for a short time, the virtual address can remain constant across all servers. For a concrete example, if we know that a migrated process on the target server generates a page fault at address 0x100001000, then we must request this exact same address on the source server. This realization is the key to being able to offload nearly all support into hardware, and make the system fast. So once the interface cards use virtual addresses, it is a relatively simple matter for us to be able to transfer a particular virtual address, without modification, from the target server to the source server to transfer the memory. The second piece of information is a bit more challenging.

The address space refers to the virtual memory subsystem in which each process has its own continuous range of memory addresses, but is sparsely populated (i.e. not every address is mapped to a physical memory location). In Linux (and this holds true in general), each address space is associated with a particular process in the operating system, and is referenced by an address space identifier (ASID). The ASID is primarily used by the processor for setting up and maintaining page tables and caches. The assignment of the ASID to a process is under the control of the operating system. In fact, two local processes may share the same ASID (in which case they share the same virtual memory) and are better known as threads. The assignments are unique to a particular server, and no server should know or should need to know how ASIDs relate to processes on any other server. Even if we were to decide that this knowledge is necessary for fast container migration, it is not practical for each server to know about all of the ASIDs of all of the other servers. The number of ASIDs would grow exponentially with the number of servers, and just the communication to manage them all would not scale even for a small number of servers. Luckily, we can reduce the problem to include only ASIDs related to processes that are currently migrating (i.e. all of the processes of a particular container), and the number of servers that need to know these ASIDs to only the set of servers involved in the migration (which may be more than two as we will see shortly). This leaves us with a relatively short list that needs to be updated only when a migration begins or ends. This list can be offloaded to the interface card as well, further simplifying the software, and minimizing the latency of a given page fault.

3.1 SETUP

Before a process can be migrated, it must be registered so that the interconnect is notified of the address space used by the process. We will call this the local address space identifier (local ASID). The local ASID is unique to that particular host and is used by the local interface card when accessing memory on the host by virtual address. When the process is to be migrated, it must first be registered with the local interface card by providing the local ASID. The local interface card must translate this local ASID to a globally unique ASID, and hold it in a lookup table on the card. Any communication by the interface cards over the interconnect must use the global ASID. Through some out-of-band communication method (i.e. standard TCP/IP network), the target machine must be notified of the migration to prepare to receive the context. At this point, the minimal context is migrated (including

[1] It is true that a container is a collection of processes, but in terms of migration, each process acts individually under its own address space

registers, stack, etc). As part of this process, the target machine must also register with its interface card (called the remote interface card) to establish a mapping between the local ASID of the target and the global ASID representing the migrating process. All subsequent page requests will include the global ASID in the request header. Before an interface card can initiate communication with its host (DMA actions), it must first use the global ASID in the request to look up the local ASID to know how to find the correct page table for virtual-to-physical address translation.

3.2 REMOTE PAGE FAULTS

After the process has been migrated and it resumes execution, it will continue to run until it hits a memory address that is marked as not present in the page tables. At this point, the CPU will generate a page fault which needs to be handled before the process can continue execution.

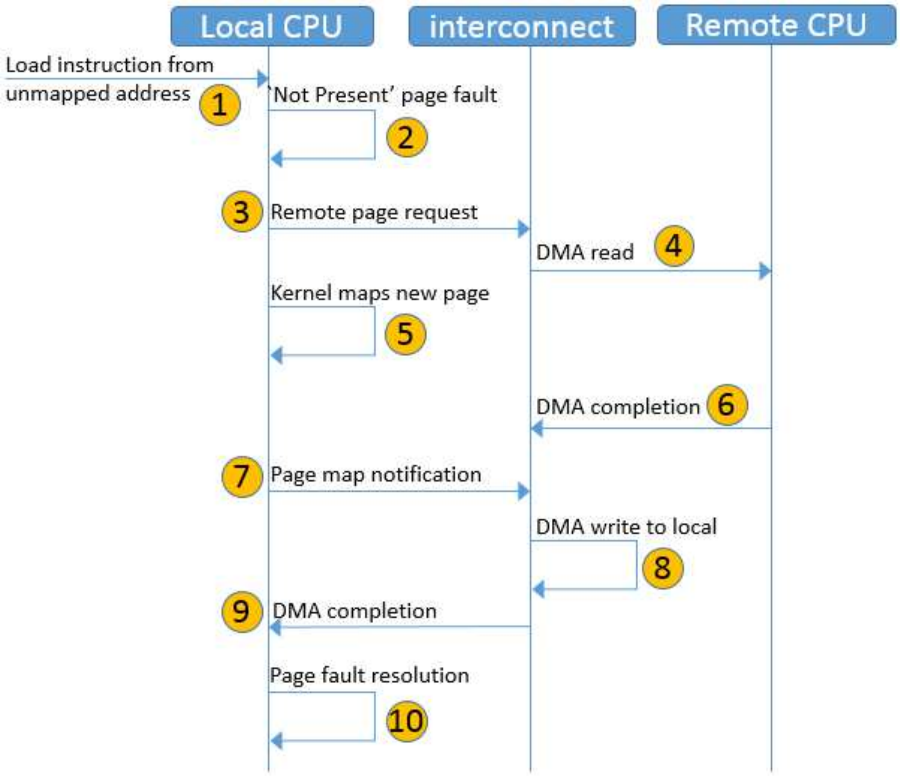


Figure 1 - Sequence diagram detailing handling of a remote page fault

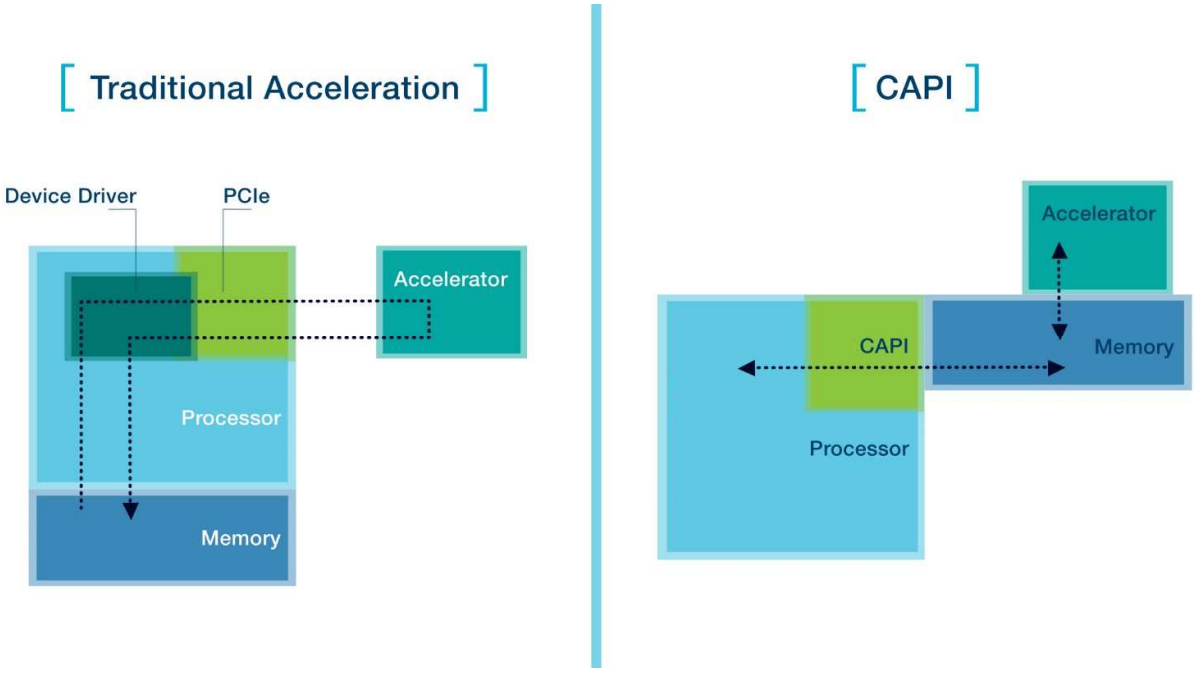
shows the sequence of steps taken to copy a memory page from a remote CPU (source system) to a local CPU (target system). It assumes that all steps have been taken to set up the migration, and details a single page fault for a single process.

1. When a migrated process attempts to access a memory location that has not yet been copied (but is otherwise valid), the access generates a page fault.
2. The fault causes the processor to switch to a privileged mode of execution which gives the operating system an opportunity to handle the fault. This is the page fault handler module of the operating system kernel. In order to fetch the page from a remote server, the page fault handler must communicate with the interface card to submit the 'read' request. The process is frozen until the page fault is handled, but other processes may be scheduled to run on this processor in the meanwhile.
3. After the local interface card receives notification of the missing page, it sends a memory read request to the remote interface card, specifying the virtual address, global ASID, and number of pages to read.

4. The remote interface card looks up the corresponding local ASID, and performs a DMA read on the requested virtual address from the host's memory.
5. For remote page faults, the kernel must first mark the thread as blocked (pending page fault resolution), allocate a new physical memory page (or draw from the page pool) and map it to the virtual address that caused the fault. The physical page must be mapped before the local interface card can copy remote data into the host's memory.
6. When the DMA read completes, the remote interface card fulfills the request by returning the data with a sequence number to the local interface card.
7. By this time, the kernel should have finished mapping the physical page into the destination address space. If the DMA write is attempted before the kernel completes the mapping, the MMU of the local interface card will not be able to translate the virtual address for the DMA request, and instead will notify the kernel and stall. The kernel can notify the local interface card upon completion of the mapping, and the DMA transfer will be retried.
8. The local interface card performs a DMA write to put the data in the memory of the local host, and then notifies the local kernel of completion.
9. When the local kernel receives notification that the DMA write is complete, it can complete the page fault resolution by setting the blocked thread as ready-to-run.

3.3 CAPI

In order to build a working prototype for testing with real hardware available today, we are looking at the best option available, which is the CAPI protocol as implemented in the IBM POWER8 and OpenPOWER systems. CAPI (Coherent Accelerator Processor Interface) is a cache coherency protocol intended for providing a cache-coherent view of system memory with attached accelerators over PCIe (I/O bus). The physical and electrical characteristics of CAPI are currently implemented as a PCIe-compatible standard, meaning a CAPI expansion slot inside a POWER8 server is in fact a standard PCIe slot, with additional hardware inside the controller that implements the CAPI messaging protocol and logic. That means we can get the required functionality of accessing memory through virtual addresses, but the physical layer is less than ideal in terms of latency since it relies on PCIe rather than being directly attached to the system bus.



In the future, we can replace CAPI with other interfaces that may be implemented on a wider variety of hardware. Recently, the GenZ [5] specification has been drafted, which is intended to be an

architecture-agnostic, high-speed, low-latency interconnect for attaching memory-mapped devices to the system bus. Such a bus would make an excellent candidate for building a controller for handling remote page faults between machines. The GenZ consortium is composed of leading industrial companies, which is a sign that there is a need for such an interconnect in commercial systems.

Similarly, specs for CCIX [6] and OpenCAPI [7] have been drafted and announced. These both specify ISA-agnostic cache coherency protocols for accelerators. We can envision building an interconnect interface card that connects to the host using GenZ, and keeps cache coherency between its own host and other hosts through CCIX or OpenCAPI to implement remote page reads. This may take the form of a remote memory controller (RMC) as described in soNUMA [8]. So while today we are confined to using CAPI when working with POWER8, in the near future, we expect to see similar protocols adopted on multiple platforms, ensuring interoperability.

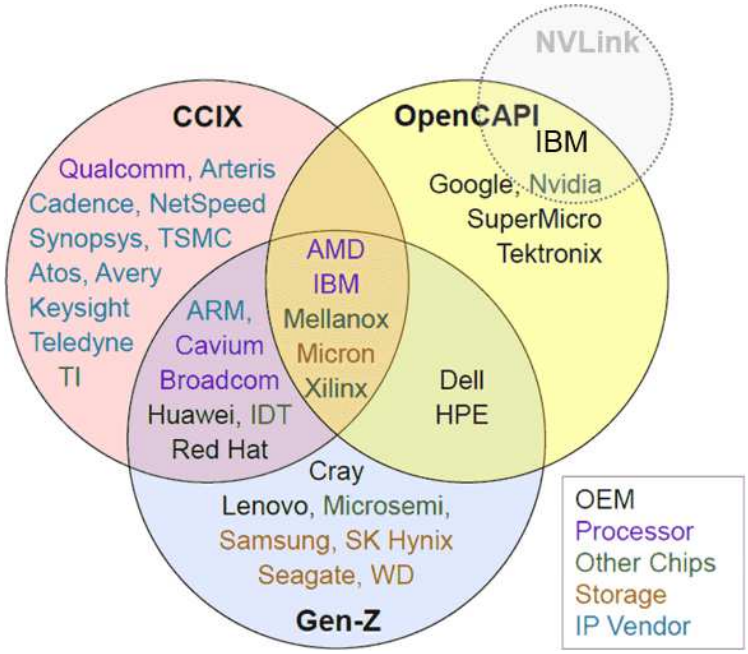


Figure 2 - The new fabric consortia showing participants and areas of interest

3.4 NETWORKING

Latency is the biggest issue when handling remote page faults. In this work, we aim to develop techniques for reducing latency in the software and hardware together. We believe future systems will reduce this latency by at least one order of magnitude by using special purpose hardware designed for memory accesses rather than I/O. To take advantage of the low latency, we must modify the operating system to also support low latency operations.

We expect the majority of container migrations to take place at the rack-scale. Racks in a data center are often viewed as a logical unit, share a top-of-rack switch for network communication, and it has been shown that most traffic inside a cloud data center is local to the rack [9]. That means we must have an interconnect that can operate well with a node count of between 10 to 40. There are at least two viable options as shown in the SCI (Scalable Coherent Interface) spec [10], and any option that provides low latency connectivity may be used.

The whole point of context migration is to take advantage of different server characteristics in order to save power. If our migration solution mitigates those gains by generating too much overhead, then we have not attained our goal. Therefore, it is imperative that the entire cost of container migration be low power, in order to guarantee a net gain in power savings.

Our first implementation of the network will be point-to-point (two servers back to back) in order to develop all of the necessary components in the system. But our design is longer term, and we recognize that some of the decisions we make to today will impact the future direction of our system. As such we try to be cognizant of those implications, and not make decisions that will affect scalability in future implementations.

3.4.1 PROTOCOL

At the link layer, we are using Ethernet frames for data encapsulation, and Ethernet MAC addresses for card addressing. At the network layer, we have developed our own protocol specifically to handle remote page faults. All communication between interface cards uses a synchronous request/response message system.

Request Message

At this point, we have the need for only a single request type (memory read), which contains the following fields:

Field Name	Field Description
Requester ID	unique identifier of the interface card on the network
Global ASID	The globally unique identifier of the requesting process
Page Count	How many sequential pages does the requesting process need
Virtual Address	The (4K aligned) virtual address of the first requested page

We guarantee that each page exists exactly once in the system, and that any memory read request that is sent to the network is for a valid address. If a process attempts to access memory which has never been mapped, the request will be stopped at the local host (handled as a regular page fault) and will not be sent to the network.

We cannot however, know in advance which server is the owner of the requested page. In the trivial example of two systems, the page must clearly exist on one of the two. In the case of three or more servers, it is possible that a process has been migrated a second time (before the first migration completed), and therefore has pages distributed across all three servers. For this reason, we send the memory read request as a broadcast to all nodes, but expect only one to answer.

3.4.2 TOPOLOGY

The read request is sent to all other nodes in the network to discover who is the owner of the request page. Depending on the network topology, the message may be sent as a broadcast (star or bus topology) or be forwarded from node to node until the owner responds (ring topology). We have not yet made a final determination as to the best network topology. At this point we are leaning towards a ring (or torus) topology since the primary concern is latency, and we believe a switch would incur more latency on average than a ring. To prove this theory, we are building a simulator in Python that can be used to measure the latency and test various network configurations. The simulation is written in Python, and is based on the NetworkX [11] module for modeling the nodes and links. The code for the

simulator will be released as open source on Github. The NetworkX module is already open source, and available from Github.

3.5 HARDWARE BLOCKS

CAPI is implemented on both the POWER8 host as well as the expansion card. On the host side, the hardware controller is called CAPP - The POWER8 chip has a Coherently Attached Processor Proxy (CAPP) unit which is part of the PCIe Host Bridge (PHB). This is managed by Linux by calls into OPAL. Linux doesn't directly program the CAPP. In OPERA, we don't make any modifications to the host side, and only require that it works according to specifications.

On the expansion card (accelerator), the design can be described as a combination of several hardware blocks. The FPGA (or coherently attached device) consists of two parts. The POWER Service Layer (PSL) and the Accelerator Function Unit (AFU). The AFU is used to implement specific functionality behind the PSL. The PSL, among other things, provides memory address translation services to allow each AFU direct access to user space memory.

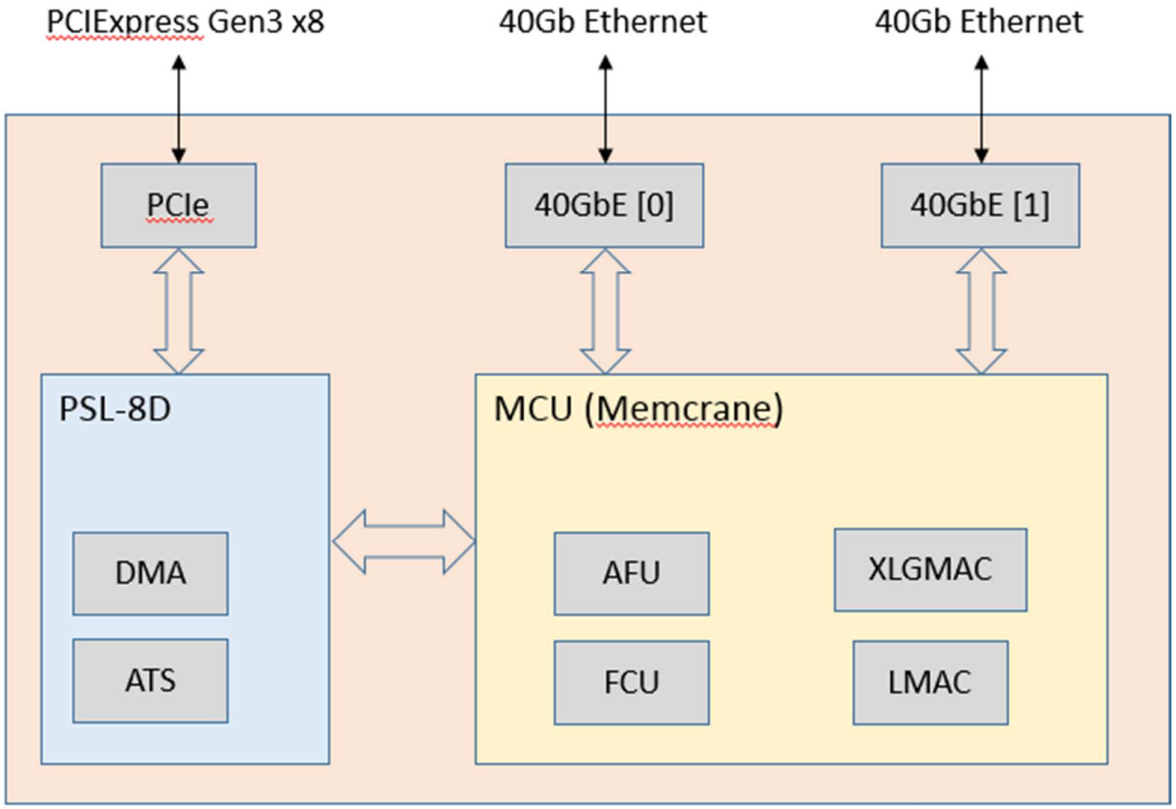


Figure 3 - High level block diagram of FPGA design

The PSL contains a DMA (direct memory access) engine and an ATS (address translation service) block. The DMA engine accesses host memory over the PCIe channel without any interference from the host CPUs. Thus the card can read (in the role of a source machine) or write (in the role of a target machine) memory pages without any dependencies on software. This greatly increases the speed at which memory can be transferred, since all logic is self-contained in the FPGA hardware. The DMA is programmed to access memory by virtual address, but the PCIe bus works with physical addresses, thus the DMA engine requires the services of the ATS to translate to a physical address before it can send a request to the host memory.

The application block contains a generic AFU along with some application-specific blocks that are required to access the Ethernet ports. The application-specific logic includes the implementation of the network protocol, as well as the control path for the DMA and the various ports. The remaining blocks

are generic Ethernet components that are reused for features such as flow control and frame headers. Reusing these components save effort to prevent having to design from scratch for a prototype such as this.

3.6 SOFTWARE

According to our design, only minimal software changes will be required. The two main changes are to write a small driver, and to add some logic to the Linux kernel. Since we are developing software only to control the prototype hardware, it is unlikely that our code will be ready to be included in a future version of the kernel. Our software will be open source for others to use freely, but its true value is in allowing the exploration of fast remote page faults, and their impact on heterogeneous system design.

3.6.1 CAPI DRIVER

Since we are developing new hardware, we must also develop a driver to interface with it. The driver is being developed as open source for the Linux kernel. It is based on the CAPI framework already part of the Linux kernel [12]. Generally, CAPI devices do not require an additional kernel driver, since the AFU functions are meant to be exposed directly to a userspace application. The Linux documentation on CXL states:

```
The kernel has no knowledge of the function of the AFU. Only userspace
interacts directly with the AFU.
```

In our case, this isn't true. We are designing the AFU to be an accelerator specifically for kernel use, which is why we must develop a kernel driver to manage it. The driver will be quite small (estimated to be 250 lines of C code) since the majority of the work takes place in hardware (the FPGA implementation).

3.6.2 KERNEL

The first modification to the kernel is in the page fault handler. When a process is resumed on the target machine after migration, it will generate a page fault as soon as it requests an address that is not present (but should be). The job of the kernel is to recognize that this page fault must be handled by the interface card, and to forward the request to the card as quickly as possible. The kernel can then mark the page fault as pending by suspending the process while waiting for a reply from the card.

The second modification to the kernel is to handle the notification from the card that the page has been received from the remote system, and copied to the correct physical address in main memory. The kernel can then resume execution of the process which will attempt to access the same memory location again, but this time it will succeed.

The third modification is to handle the error condition which arises when the interface card attempts to access the target address (faulting address generated by the process) before it has been mapped to a physical page in the page tables. This is the trigger for the operating system to allocate a new physical memory page, and map it to the virtual address that caused the fault. When the mapping is complete, the kernel will notify the card to retry the access, but this time it will succeed. This is the mechanism by which a CAPI-enabled card can gain fine-grained control over a processes' address space. That means that no pinning is required when using CAPI, which is a big advantage over plain PCIe-based hardware solutions.

4 IMPLEMENTATION

4.1 LOSS OF CAPI SUPPORT FROM ALTERA

Originally, the task of implementing the NIC prototype was to be handled by Nallatech, on the assumption that they would receive the BSP (board support package) required to implement the FPGA code from Altera. Unfortunately, due to unforeseeable circumstances, that turned out to not be the case. Altera (the supplier providing the FPGA chip on which this card is based) was bought by Intel soon after the start of the project. Subsequently, Altera has decided to not support CAPI in their BSP for future designs, including the FPGA chip we are using. This left us with a minor crisis as to how to proceed in testing CAPI functionality, without any code to implement CAPI.

Luckily for us, we identified a development team inside IBM that is able to take up the challenge and help us support CAPI and develop the code for this FPGA, essentially bypassing Altera and providing the code in a raw format (not packaged nicely as a BSP, but still functional). Despite this unfortunate event, we have managed to get back on schedule, with only a minor delay. The change occurred early enough in the project that we were able to find an alternate solution and recover quickly, mostly since the hardware is only now becoming available after the bring-up process.

4.2 AVAILABILITY

Unfortunately, the CAPI code is still proprietary IBM, and cannot be released as open source. It has been licensed to several other companies (including Nallatech) but we don't know of any plans of releasing it as open source. However, a very similar protocol called OpenCAPI [1] is open source, and is gaining popularity. The OpenCAPI consortium was only recently formed (in October 2016) so is not yet mature enough to be used in products. As such, there are not yet any servers which support OpenCAPI available for research projects, and we don't have access to teams with OpenCAPI development experience or capabilities. However, we believe that the design of OpenCAPI closely follows that of CAPI, and that the results that we publish using CAPI will be directly applicable to other protocols such as OpenCAPI, when they become available.

4.3 INTEGRATION WITH MOONSHOT

One of the main goals of the OPERA project is to take advantage of heterogeneous systems for the purpose of reducing power consumption. The particular optimization that we are developing as part of task T6.5 relies on a server-side hardware feature (CAPI) that is only available on one particular architecture (POWER systems). So until such hardware (or similar as described in section 3.3) is available on more systems, we are faced with the problem of how to show the optimization is effective on multiple architectures. Fortunately, there is a viable solution available. As mentioned earlier, CAPI is currently implemented on top of a standard PCIe bus interface. That means the physical dimensions and electrical signalling of the prototype NIC are compatible with PCIe. Thus the card developed by Nallatech will work in both a CAPI and PCIe slot equally well. As proof, we can see that the same card is being used as an application accelerator in the "truck' use case, which uses standard PCIe to connect to the HPE Moonshot. However, the FPGA firmware required to run the card in PCIe mode is a substantially different from the firmware required to support CAPI. Given the effort to produce two firmware images (one to support CAPI, the other to support raw PCIe), the question is then what functionality do we lose by using standard PCIe (rather than CAPI), and whether or not we can connect a pair of cards, with each card operating in a different mode (one in PCIe mode, and the other in CAPI mode). At this point, this is only an interesting, but theoretical question.

In the future, we hope to show that the NIC can function as a standard PCIe device in a server that does not support CAPI, albeit with less functionality. With some additional modifications, it will be possible to

attach the card to a regular PCIe slot when it is in CAPI mode, but it will lose the ability to perform CAPI-specific functions, such as sending and receiving cache coherency messages. Since our network does not (yet) use the cache coherency feature of CAPI, the card will function well enough to at least act as the "source" side of the migration, which is the more passive of the two functions. This will require a new driver (to activate the card in PCIe mode) as well as changes to the FPGA image. If we are successful in making the changes to the system to work in PCIe mode, we should be able to connect it to other architectures such as x86 and ARM which do not yet support CAPI. In this way, we could show functionality of container migration at least in one direction (from any system to POWER), and it would likely be enough to measure power consumption and performance as well. This work is currently out of scope for the OPERA project, since we will not have enough resources available to re-implement the FPGA code to support only PCIe.

5 MEASUREMENTS

Once the implementation is complete, we will take measurements to quantify various aspects during container migration. The most important measurements will be the energy consumption of the card and the energy consumption of the system (including CPU) during migration. We aim to show that the energy consumption is less than a similar system using TCP/IP over Ethernet.

5.1 METHODOLOGY

The power measurements will be taken using both internal and external probes. The numbers will be used to validate the results, and provide insight into components that cannot be measured directly. The POWER8 machine has a set of internal probes that can be used for power measurements. They are designed to measure internal server components on the motherboard such as CPU power. These probes have been validated with external measurement equipment, and found to be very accurate and reliable.

The FPGA is also equipped with internal power measurement probes. We have not yet validated their accuracy, but based on Nallatech's previous experience with similar hardware, we believe that these probes will also be accurate.

We have purchased an external power measurement analyzer from Tektronix called the PA-1000. This will be used to measure the total current drawn by a particular server. We can then take a series of measurements of the equipment during a period of activity and a period of rest with both internal and external probes for comparison. More details on the equipment and procedures will be available in deliverable D4.7 (available in M24).

6 CONCLUSIONS

We are taking a very ambitious step to design a new network interface card and network protocol specifically designed to handle remote page faults. This new network will be used to reduce energy consumption and latency of page faults when performing post-copy migration of containers, virtual machines, or processes.

This kind of project is complicated because it involves the development of many different components including hardware, FPGA firmware, driver software and protocols. We believe that we have scoped the work to be on target with the goals of OPERA, and that will have real results during the lifetime of the project. If successful, this may have a wide-reaching impact on data center design.

7 REFERENCES

[1] Nallatech, "Nallatech 385A-SoC System on Chip FPGA Accelerator Card," 2017. [Online]. Available: http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-385a-soc/.

[2] E. Zayas, "Attacking the Process Migration Bottleneck," *SIGOPS Oper. Syst. Rev.,* vol. 21, no. 5, pp. 13-24, 1987.

[3] M. R. Hines and K. Gopalan, "Post-copy Based Live Virtual Machine Migration Using Adaptive Pre-paging and Dynamic Self-ballooning," *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments,* pp. 51-60, 2009.

[4] CRIU, "Checkpoint/Restore In Userspace," CRIU, 2017. [Online]. Available: https://criu.org/Main_Page.

[5] GenZ, "The GenZ Consortium," GenZ, 2017. [Online]. Available: http://genzconsortium.org/.

[6] CCIX, "Cache coherent interconnect for accelerators (ccix)," CCIX, 2017. [Online]. Available: http://www.ccixconsortium.com/.

[7] OpenCAPI, "Welcome to OpenCAPI Consortium," 2017. [Online]. Available: http://opencapi.org/.

[8] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi and G. B., "Scale-out numa," *SIGARCH Comput. Archit. News,* vol. 42, no. 1, p. 3–18, 2014.

[9] T. Benson, A. Akella and D. A. Maltz, "Network traffic characteristics of data centers in the wild," *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement,* p. 267–280, 2010.

[10] IEEE, "IEEE Standard for Scalable Coherent Interface (SCI)," 1992. [Online]. Available: http://ieeexplore.ieee.org/document/347683/references.

[11] B. Edwards, D. Schult, A. Hagberg, J. Torrents, L. Séguin-Charbonneau and chebee7i, "High-productivity software for complex networks," NetworkX, 2017. [Online]. Available: https://networkx.github.io/.

[12] Linux, "Coherent Accelerator Interface (CXL)," 2016. [Online]. Available: https://www.kernel.org/doc/Documentation/powerpc/cxl.txt.