Großer Beleg

Exploring Inter-Core Message-Passing for Fiasco on the SCC

Markus Partheymüller

21. Dezember 2011

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Julian Stecklina
Dipl.-Inf. Björn Döbel

Aufgabenstellung

Der Intel Single-Chip Cloud Computer (SCC) ist ein Many-Core-Forschungschip, der 48 verhältnismäßig schwache Pentium-Cores in einem Mesh-Netzwerk auf einem Die vereinigt. Die Architektur des SCC unterscheidet sich stark von aktuellen x86-Multi-Core-Prozessoren, da er keine Cache-Kohärenz bietet, Kommunikation sowohl über den nicht-kohärenten Speicher als auch über Message-Passing erlaubt und darüber hinaus kein BIOS oder eine ähnliche Firmware bietet.

Fiasco.OC ist ein an der TU Dresden entwickeltes capability-basiertes Mikrokern-Betriebssystem. Basierend darauf wurde eine Laufzeitumgebung zur Entwicklung von Multiserver-Systemen, das L4 Runtime Environment (L4Re) implementiert.

Im Rahmen der Belegarbeit, soll die aktuelle Version von Fiasco.OC und dem L4Re auf den SCC portiert werden, so dass pro Core eine Instanz von Fiasco.OC mit eigenem Userland läuft. Die Änderungen am Code sollen modular gehalten werden und in den Entwicklungszweig von Fiasco.OC/L4Re integrierbar sein. Wesentliche Unterschiede zwischen dem SCC und der PC-Plattform sollen dokumentiert werden. Weiterhin soll untersucht werden, inwieweit das Message Passing des SCC zur Kommunikation zwischen den Instanzen genutzt werden kann und welche Kommunikations-Leistung damit erreicht werden kann.

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dipl.-Inf. Julian Stecklina
	Dipl.-Inf. Björn Döbel
Beginn:	1. Juli 2011
Einzureichen am:	31. Dezember 2011

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 21.12.2011

Markus Partheymüller

Abstract

The craving for ever increasing performance of computer systems is an omnipresent factor in the design and development of hard- and software. Another trend, rather contrary to the desire of more and more speed, is the pursuit of energy efficiency because of the energy transition from fossil fuels towards renewable energies.

In this context, research and experiments with new computer systems as well as new software designs became very important. A very prominent example of such new computer systems is the multi-core or even many-core design. Instead of a single CPU, an increasing number of computing units is integrated into one chip. But by having more CPUs, the performance of the software running on the system does not necessarily increase, it might even perform worse than on a single core. Thus it is necessary to investigate how software can harness the power of many-core systems.

In this thesis, the Fiasco.OC microkernel, developed at the TU Dresden, was ported to a many-core system made available to researchers by Intel, the *Single Chip Cloud Computer*. The goal was to evaluate how difficult it is to run the microkernel system on this kind of system and also to explore opportunities and shortcomings that come along with it.

The process of porting is described in terms of the steps needed to make Fiasco.OC bootable on the chip as well as obstacles that had to be resolved because of the chip's particularities. After having ported the microkernel itself, preliminary experiments were conducted to learn about an essential concept of many-core systems: inter-core message passing. To harness the power of many cores on one chip, the communication among them is critical to the success of this design.

The findings of the port itself and the message passing experiments led to several potential future projects that can be based on the work done in this thesis. These ideas are presented at the end of the thesis as an outlook.

Contents

List of Tables

List of Figures

1 Introduction

> "At some point, Moore's law will break down. The question is, when?" - Seth Lloyd

For decades, performance problems of software resolved themselves over time, when processors grew faster and faster. A common understanding of this evolution is widely known as *Moore's Law*, stating that the number of transistors doubles every two years [11]. However, the increase of computing power already started to change. While modern processors still follow Moore's Law, the performance of older applications already tends to stagnate since the frequency could not be increased anymore, but instead multiple CPUs were integrated into one chip. To harness this power, applications have to exploit parallelism in their algorithms. At some point in time, the development of processors is expected to reach the physical boundaries of nanotechnology. At the latest then, other approaches to improving computing power have to be investigated in order to meet the exploding requirements of consumers.

The uprising research projects of the semiconductor industry prove that the awareness of this issue is growing: The trend goes to *multi-core*, even *many-core*. Additionally, to move the computing power (and also storage) away from the end user into a controlled environment, becomes a more and more important concept in modern systems. This is widely known as cloud computing and provides more opportunities to exploit parallelism at a higher degree than it can be on a single desktop PC or notebook.

One example for such a research project is Intel's *Single-Chip Cloud Computer* (SCC) [12], an experimental processor that was made available to researchers all over the world to investigate the opportunities, difficulties and limitations of a many-core environment. The TU Dresden operating systems group, being member of Intel's Many-Core Applications Research Community, has access to an SCC chip in the US, which was used for all experiments for this thesis. As the name suggests, the SCC is meant to be a symbiosis of a many-core approach and cloud thinking. It integrates 48 cores on one single die, resembling a cluster used for a computing cloud.

To benefit from this architecture, software has to be extended or new applications need to be developed. Intel provides a set of basic applications (e.g., an SCC version of Linux and a message passing library for communication between the nodes of the cloud).

But instead of using the provided Linux, the research on parallel programming can be combined with the pursuit of a minimised trusted computing base Especially for cloud computing, the customers' demand for stability and security will grow. Microkernels pose a promising alternative to monolithic systems, as the portion of software that needs to be guaranteed for is kept as small as possible.

The microkernel developed at the TU Dresden, *Fiasco.OC* [22], was chosen to be investigated in the SCC environment. In the context of Intel's *Many-Core Applications Research*

Community (MARC), a Fiasco.OC-based system was used to assess performance characteristics as well as difficulties that can be encountered when working with the SCC, or many-core setups in general.

In this thesis, the process of porting Fiasco.OC to the SCC architecture, extending the existing software and implementing new applications to make use of the SCC-specific features is presented.

After introducing in Chapter 2 the technical background needed to understand the design and implementation, Chapter 3 explains the details and difficulties that needed to be dealt with in the process of porting Fiasco.OC and developing software for the SCC. After evaluation of the work and experimental results in Chapter 4, the thesis closes by discussing ideas for future work in the context of the SCC and a conclusion is drawn in Chapters 5 and 6.

2 Technical Background

In this chapter, prerequisites, basic terms and concepts that are necessary to understand this thesis are introduced. Also, related work on topics similar to those discussed in this work is presented.

First of all, the working environment regarding hardware and software is described.

2.1 Experimental Environment

The work described in the following chapters takes place in the context of the current Fiasco.OC microkernel [22].

As an L4 based microkernel, Fiasco.OC is the only portion of the software stack that is running in privileged processor mode. Conforming to the philosophy of microkernels, it only provides functionality that is essential to building a system, such as scheduling, address spaces, threads and inter-process communication. All other policies and features are left to be implemented on top of the microkernel in user level.

Fiasco's main concepts to provide the basic functionality for the operating system environment are capabilities [16], which can be viewed as references to objects, protected by the kernel. These objects can be implemented either in the kernel itself (e.g., tasks, threads or interrupts) or also in the userland (e.g., memory managers or device drivers). Any application holding a capability to a certain object can make use of the object's functionality by using the invoke() system call. The choice of the name Fiasco.OC reflects the essential role of these correlating concepts: Objects and Capabilities.

Policies, such as resource management, are implemented on top of Fiasco.OC. A collection of basic services is available with *L4 Runtime Environment* (L4Re) [23], which provides a set of libraries and servers commonly used (e.g., a C library, memory managers, ELF loaders) as well as a variety of additional software, such as a graphical user interface implementation, ports of libraries including Qt, an sqlite database implementation and also the debugging tool Valgrind [17].

The boot process of a Fiasco based application involves bootstrapping the kernel itself and starting the servers providing the most basic services. The first task called *sigma0* is only used as the root pager for the actual root task *moe*. Moe incorporates a program loader, address space manager as well as a memory allocator interface. An overview of this structure is given in Figure 2.1.

Fiasco.OC and L4Re lay the foundations necessary to implement all kinds of user-level applications. One example of a highly demanding application running on top of Fiasco.OC is L4Linux [7], a para-virtualised version of the Linux kernel. It allows for running arbitrary binaries that were built for Linux, while still offering the advantages of a virtualised environment. Because L4Linux is run as an unprivileged application, it can be isolated and

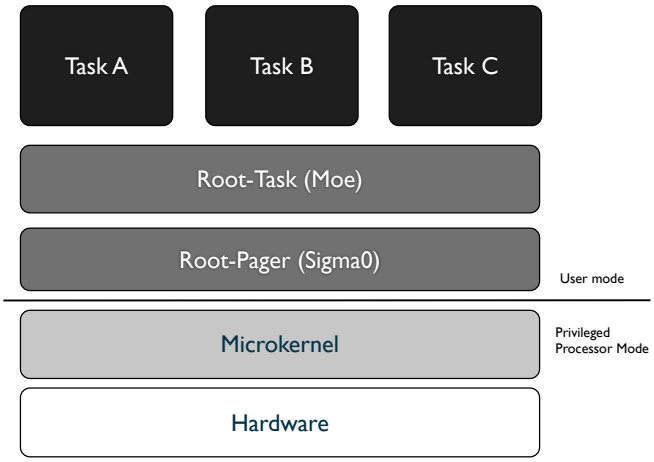


Figure 2.1: Basic structure of an L4Re based System

even constrained by an application from the outside. Furthermore, resource utilisation can be further increased by running multiple instances of L4Linux on top of one microkernel.

The entire software stack can be integrated into one binary by the L4 Build Infrastructure. These ELF [1] images contain all needed binary files as well as information about the desired location in memory in a single file. The produced files consist of a *Program Header Table* and associated data sections. In the headers, destination address, offset in the file and section size are specified. Using this information the data can be transferred to the SCC's memory at the desired locations.

2.2 Overview of the SCC

The SCC is an experimental multi-core processor developed by Intel in the context of their *Tera-scale Computing Research Program*, integrating 48 Pentium P54C cores on one chip. As these cores feature a full x86 architecture, they can run all kinds of software compiled for this architecture, including bare metal applications or Linux.

Figure 2.2 shows how the 48 cores cores of the SCC are connected through a mesh network. The chip consists of 24 tiles arranged in a 6x4 mesh array, each containing two cores, two separate L2 Caches, one message passing buffer (MPB) and a mesh router. In addition, four DDR3 memory controllers are connected to the tiles (0,0), (0,5), (0,2) and (5,2), each of which can address a maximum of 16 GiB of memory. All memory and message passing requests are handled by a mesh interface unit (MIU), also present on each tile. The whole system is configured and managed by a Management Console PC (MCPC) using a PCIe System FPGA board, connecting to a System Interface (SIF) attached to tile (3,0).

In this section, some important aspects of the chip design are described, pointing out similarities and differences to the common PC platform or the original x86 architecture. For a better understanding of the thesis, a basic introduction to caches and their strategies is given first, as these terms are used throughout the entire work.

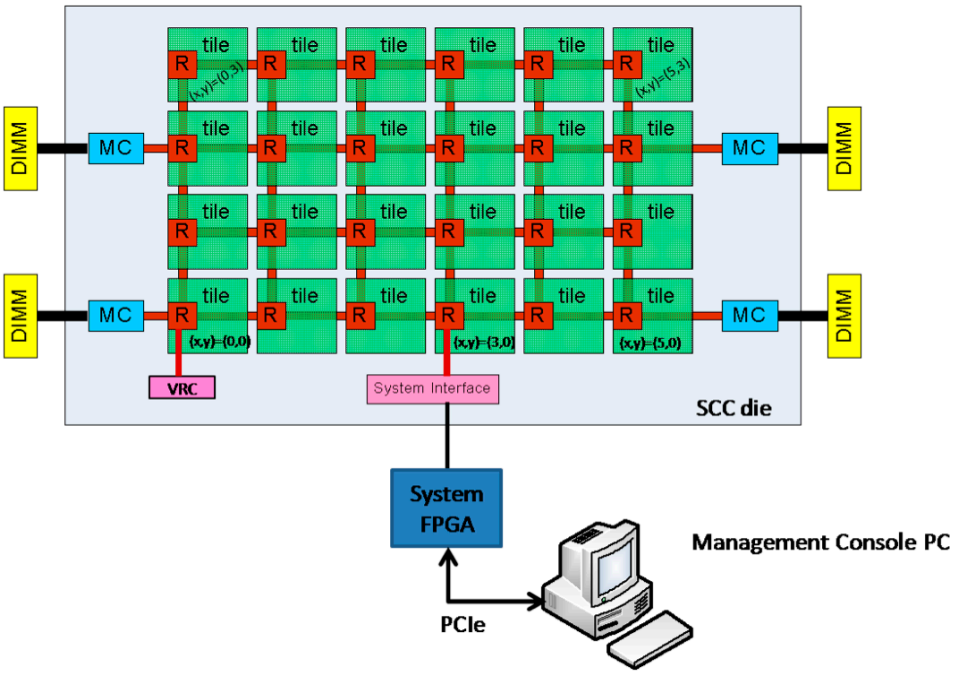


Figure 2.2: Overview of the SCC architecture (Image provided by Intel [13, p. 8])

2.2.1 Cache Strategies

Since the late 1960s, caches are used in PC systems to overcome the growing gap between processor and memory performance at moderate prices. Exploiting data locality, they act as a small intermediate layer of storage, providing fast access to data that is likely to be used in the near future.

Many dimensions of optimisation exist in order to tweak caches to yield maximum performance. Because caches are devices for buffering data that is often used, improving their efficiency inherently raises the question how to deal with data not present in the cache. For example, the handling of accesses to locations not present in the cache (*cache misses*) or how modified data should be propagated to the higher levels of the memory hierarchy can have a large impact on the overall performance of a memory hierarchy. There exist two orthogonal strategies concerning Write Hit and Write Miss, respectively.

On a Write Hit, a cache can update the cache line and propagate the change to higher hierarchy levels, which can be another cache or the main memory. This is called a *Write-Through* strategy. Contrary, the *Write-Back* policy only writes back to higher levels on a cache flush or upon cache line eviction.

When a write request results in a Write Miss, there are two options: Allocate a new cache line and then proceed just like Write Hit, or directly write to the higher level. These

strategies are called *allocate-on-write* or *non-allocate-on-write*, respectively. The latter is also often called *Write-Around*[1].

The SCC's caches are strictly *Write-Around*, whereas the Write Hit strategy can be configured through the page table entry. This results in a very poor performance of pure `memset/memcpy` operations, as every write operation is sent as individual transaction instead of being combined into a full cache line write. Therefore, an investigation of memory bandwidth using different countermeasures to this problem became an interesting research topic [25].

2.2.2 Processor Design

The Pentium P54C architecture, developed in the 1990s, was Intel's first superscalar x86 microarchitecture, featuring two integer pipelines, on-chip separate data and instruction L1 caches. It also contains an on-chip interrupt controller (*Local APIC*), which is used as timer source and for generating interrupts using the local interrupt pins [10].

Only a few changes were made to this original processor design to make it more suitable for the use in the SCC environment.

The cache architecture was extended from 8 KiB to now 16 KiB of non-unified L1 cache. Also, a 256 KiB unified L2 cache has been attached to each core. Additionally, the originally off-chip front side bus-to-cache controller interface (also known as M-unit) was equipped with a *Write Combine Buffer* (WCB), 32 byte in size, and is now integrated in the core. This buffer combines write requests to DDR memory into full cache line writes.

Note that there is no hardware coherency between caches on the SCC. In order to perform message passing, ensuring coherency between the memory locations involved (including caches) is inevitable. For instance, It has to be prevented that the receiver reads stale data when getting a message. In common multi-core systems, this is done using hardware cache coherency (e.g., using the MESI [18] protocol: When the sender writes a shared memory location, all caches are forced to invalidate the corresponding cache line.

On the SCC, without hardware cache coherency, the problem has to be addressed in software. To facilitate this, a new memory type called *Message Passing Buffer Type* (MPBT) was introduced, which can be used to mark message passing data.

In the original P54C architecture, there existed three memory types, determining the cache-related behaviour: *Uncacheable*, *Write-Back* and *Write-Through*. Figure 2.3 illustrates how these memory types can be configured via bits in the corresponding page table entry by setting or clearing the bits as given by Table 2.1.

The new memory type was instrumented with a new instruction CL1INVMB to simplify dealing with coherency between cache and message passing data by the following means:

- All data tagged with MPBT bypasses the L2 cache.
- The new instruction CL1INVMB invalidates all L1 data cache lines of the local CPU tagged with MPBT.

[1] A more elaborate explanation of these cache strategies is given in *Computer Architecture: A Quantitative Approach* [8, p. 74 ff.].

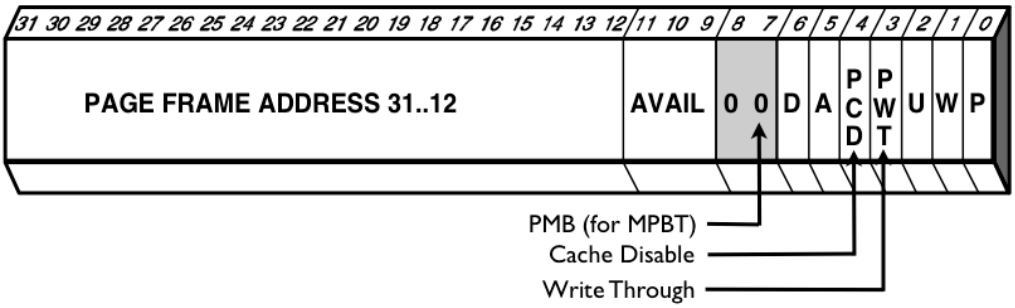


Figure 2.3: Page table entry with cache related bits (Image provided by Intel [9, p. 11-20])

PCD	PWT	PMB	Memory type
0	0	0	Write-Back
0	1	0	Write-Through
1	0	0	Uncacheable
1	1	0	Uncacheable
0 or 1	0 or 1	1	MPBT

Table 2.1: Supported memory types and corresponding bit settings

In order to use the MPBT, an additional flag (Bit 11) in the control register CR4 has to be set. The MPBT flag itself can be set in the page table entry as described above. Because it shares its location with the PSE bit used for superpages, the Page Size Extension has to be disabled in CR4 when MPBT is intended to be used. Enabling both results in exceptions when the PSE bit is set.

2.2.3 Configuration Registers

The SCC provides several configuration registers used for configuring parameters of caches, processor state, frequencies and voltages as well as some read-only status values. The most important registers for this work are the Tile ID Register, which can be used by systems code to query the coordinate of the tile it is running on, and the Core Configuration Registers, which are used for triggering interrupts. This was achieved by directly connecting the corresponding bits of the register to the interrupt pins of the Local APIC.

2.2.4 Memory Architecture

In order to flexibly assign DDR memory portions to individual cores, an additional level of address translation was installed. The 32-bit physical address of a core is transformed into a 34-bit system memory address by a Lookup Table (LUT) as illustrated in Figure 2.4, resulting in an address range of 16 GiB. Using the additional LUT fields (a destination and sub destination ID), these addresses can be routed to each of the four memory controllers.

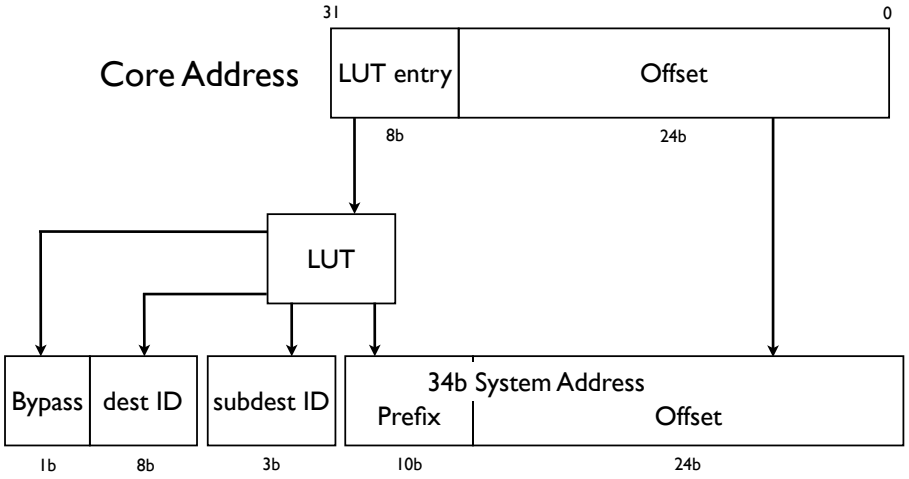


Figure 2.4: Scheme of LUT address translation

With this mechanism, the maximum total of 64 GiB of memory can be distributed among the cores in a free manner, as the LUT is fully configurable for each core independently. A default map for an even distribution of memory is given in the SCC External Architecture Specification [13].

Not only the DDR memory is mapped using the LUT, but also configuration registers and the message passing buffers. Properly configured, each core can access all configuration registers and MPBs by using the defined addresses.

2.3 Intel sccKit

As an interface from the MCPC to the SCC, Intel provides a software package called *sccKit*, providing several tools and APIs needed to access the chip's memory and configuration registers. For instance, tools for resetting cores, initializing the platform, accessing memory (DDR, MPB) and an API for handling I/O requests issued on the cores are available. Furthermore, Intel developed a convenient booting toolchain which can preload and boot software images to a given number of cores, set all necessary configuration values (e.g., LUT entries, cache parameters) and boot all the cores. The graphical user interface *sccGui* serves as a general interface to the chip. With it one can read/write memory, boot cores, initialise the system, monitor the cores' performance, and, among other things, emulate serial console and graphical text console output. Detailed information about *sccKit* is given in the *sccKit User's guides* [14].

2.4 RCCE

To facilitate getting started with message passing research, Intel provides a library called RCCE. Consisting of interfaces for both basic and advanced programming as well as a power management API, this library allows for experiments on message passing without having to develop the entire software stack. All effort can be put into the development of the actual multi-core application rather than the communication mechanisms.

Basically, RCCE provides a set of functions that abstracts all SCC specific functionality. This way, a programmer can more easily implement message passing applications by just adhering to the given basic interface. If needed, the advanced interface can be used to gain access to more low-level implementation details.

Related work on communication using RCCE or derived applications like iRCCE [3] includes investigation of memory performance issues [25], high performance computing applications on the SCC [5], but also research into the SCC's thermal sensors and their use for mitigating chip deterioration [2].

2.5 SCC Particularities

The SCC, being a research processor, shows some differences to common PCs, which can lead to unexpected behaviour or the need for workarounds and modifications to the operating system environment. Mostly, the particularities are due to the concept of a many-core system which is not intended to be used in the consumer area. Thus, the common hardware environment is not present because it is not needed. Consequently, most of the differences manifest as absence of devices.

2.5.1 Missing Devices and Functionality

As a brief overview of the deviations from normal PCs, there is a number of devices or functionality not present on the SCC:

- There is no BIOS.
- There is no Programmable Interrupt Timer (PIT), I/O APIC or Real-Time Clock (RTC).
- There is no periphery like storage devices, keyboard or graphics card.
- I/O requests are not handled by the corresponding devices (as there are none), but instead routed to the management PC.

In the next chapter these particularities will be discussed in more detail, as well as how the work with Fiasco.OC and L4Re is affected, together with the discussion of possible solutions.

3 Porting Fiasco

As pointed out in Section 2.5, running Fiasco on the SCC is different from using a normal PC in the boot process as well as in the availability of certain devices. In this chapter, the process of porting and investigating inter-core message passing is described along with the obstacles that had to be handled.

3.1 Fiasco on the SCC

For porting Fiasco.OC to the SCC, it was necessary to decide how the chip should be seen by the kernel. One possibility is to consider the entire chip as a symmetric multiprocessor system with a shared address space (SMP). However, for this approach cache coherency among the cores has to be maintained. On the SCC, there is no hardware cache coherency mechanism, therefore it has to be dealt with in software. The other option is to see the SCC, as the name suggests, as a *cluster on chip* and port Fiasco.OC in a way that every core runs exactly one instance of the kernel. Because the software cache coherency problem is not yet reliably solved, in this work the per-core approach is followed.

Due to the particularities described in Section 2.5 the existing *L4.Fiasco* microkernel cannot be run unmodified on the SCC. Given the only little deviation from the original IA32 architecture, implementing the changes as alternative code, configurable using the kernel configuration tool, directly into the existing x86-32 kernel code is preferred over the addition of a new subarchitecture to the Fiasco codebase. In this section, existing subsystems of both Fiasco.OC and L4Re affected by the SCC's specifics are presented, together with concepts and discussion of the necessary modifications.

3.1.1 The Boot Process

The critical part of porting an operating system is to enable it to boot on the new target platform. The boot process strongly depends on the underlying hardware and is therefore the starting point for the port. In this section it is described how Fiasco.OC was booted on the SCC and what difficulties had to be managed.

3.1.1.1 Multiboot

> "Every operating system ever created tends to have its own boot loader. There is little or no choice of boot loaders for a particular operating system — if the one that comes with the operating system doesn't do exactly what you want, or doesn't work on your machine, you're screwed." [6]

Every operating system needs a boot loader to initialise the hardware and start the kernel. For a long time, there did not exist a standard how such a boot loader has to work and a variety of operating systems also featured a variety of boot loaders. This problem led the developers of Multiboot to create a standard interface between boot loaders and the operating system. Various details about the PC can be passed to the operating system using this interface, including available drives, memory, a command line parameter or video hardware. An OS with Multiboot support can be easily booted on another system by simply providing the necessary information. While booting, an operation system usually uses BIOS calls to discover present devices, such as hard disks, main memory or a graphics card. These BIOS calls can be replaced with the use of the Multiboot information structure. When an operating system is designed to use this structure, the machine has to be in a specific state:

- The EAX register must contain the magic value 0x2BADB002.
- The EBX register has to contain the address where the information structure is located.
- The segment registers have to be properly configured.
- In the Control Register CR0, Paging has to be disabled, while Protected Mode needs to be enabled.
- In the EFLAGS register, Virtual 8068 Mode and the Interrupt Flag have to be disabled.

Because Fiasco.OC can be loaded by a Multiboot-compliant boot loader, it is sufficient to provide the boot process with this standardised environment. Thereby it is possible to circumvent the need for emulating BIOS calls or replacing all code related to the boot process by SCC-specific initialisation code (resulting from the absence of a BIOS on the SCC). Beside the register state, in the case of the SCC one has to provide only information about accessible memory regions. Additionally a command line string is passed to the operating system. After having set up this structure and established the needed machine state described above, the system can be booted without any modification of Fiasco's bootstrap code.

3.1.1.2 Timer Calibration

While booting Fiasco.OC, a timer calibration is performed, which calculates the ratio of clock cycles (accessible through the TSC register) to time values like nanoseconds. This is usually done using a second timer with configurable frequency. By setting this frequency to a known value and measuring the clock cycles during a fixed time interval, the ratio can be derived. On the SCC a second timer (e.g. the PIT) is missing and therefore there is no way for the core to determine the ratio on its own. To address this issue two solutions are possible: The first one is to calculate the ratio beforehand and hard-code it into the kernel. However, this assumes a fixed core frequency which is not necessarily the case on the SCC because of frequency scaling mechanisms. The second solution is to pass the core frequency as a command line parameter. It allows for this flexibility and was therefore preferred. Using this approach, dynamic frequency scaling is not yet taken into account.

On the SCC, the information necessary for the kernel to discover the frequency setting on its own is only partly available through standard methods and therefore dynamic frequency scaling was not pursued in this work. Although It is possible to access the FPGA through the System Interface, the need for developing a driver to use this interface led to omitting this concept.

3.1.1.3 Input and Output Devices

Without a graphics card and keyboard, the input and output of the kernel (and also user programs) has to be handled in some other way. One possible implementation is the text buffer, normally mapped at address *0xb8000* in memory. A PC that has a graphics card uses this location to handle the frame buffer output. Even if there is no video hardware on the SCC, it is still possible to use this buffer by backing this memory location with main memory. The data written there can be read and interpreted by the MCPC.

Alternatively, an emulated serial console on the management PC can be used. It was decided to follow this approach because of two reasons: First, not only output, but also character input is desired for debugging purposes. Second, some I/O requests used in Fiasco's boot process have to be handled by the MCPC anyway to keep the kernel modifications minimal, thus it makes sense to extend this handler to provide for the input and output functionality as well. Consequently, this handler is designed to also handle requests directed to a serial console. The existing keyboard driver is switched off using a compiler switch and is replaced by the serial console character input. While this driver normally relies on interrupts to signal that new input has arrived, it can be forced to poll for input by setting the corresponding interrupt identifier to an invalid number, passed to the boot process as command line parameter. Assuming there is no attached interrupt to serial input, the driver then falls back to the polling mechanism. In this mode, the driver continuously sends read requests to the serial console, which can be answered by the management PC.

3.1.2 Additional Functionality of the SCC

Beside booting, the operating system is used to provide access to specific hardware features. Consequently, the SCC port does not only need special modifications due to missing devices, but also because it provides additional features concerning the multi-core setup as well as the message passing facilities.

3.1.2.1 Message Passing Facilities

In order to be able to use the message passing buffers, it is necessary to make them available to user-mode programs. As these regions are a special kind of memory accessible through a specific physical address range, it makes sense to see them as I/O memory just like ordinary devices provide it.

In the current *L4Re* tree, there exists a server and a corresponding library for handling I/O memory by providing a virtual PCI bus to user applications, configurable by special configuration files. These packages (*io* and *libio-io*) are configured to provide the standard memory regions of the SCC as user-mappable memory.

3.1.2.2 Interrupt Notification

To establish communication and synchronization mechanisms, it is useful to have interrupts among cores. Although on the SCC there is no notion of Inter Processor Interrupts (IPI), it is still possible to trigger interrupts on remote cores through their Local APIC. The manufacturer achieved this by connecting the interrupt pins of the controller to dedicated bits in the configuration registers.

For using the Local APIC as interrupt source, the Local Vector Tables (LVT) were configured according to the manual [9, p. 19-4] to use the *FIXED* edge-triggered delivery mode to interrupt vectors *0x2d* and *0x2e* (IRQ13 and 14), ensuring that the interrupts are accessible from user-space through these IRQ numbers.

In normal computer systems, the Local APIC interrupt pins forward interrupts from external sources (e.g., I/O APIC, PIC). The use of the LAPIC as dedicated interrupt source is therefore not common and it is thus not intended in the current kernel version to use the fixed delivery mode. As a result, an additional interrupt acknowledgement (i.e., a write to the APIC EOI register) had to be added to the interrupt handling, which is only needed for this mode. Otherwise, only the first interrupt signal would ever reach the processor.

A signal can then be sent by toggling the interrupt bit in the remote core's configuration register. Using the level-triggered mode (i.e., the sender sets the bit, the receiver acknowledges by clearing it) is also possible, but was not used in this setup because of the need for busy waiting on the interrupt bit as long as the receiver is occupied with processing the interrupt.

3.1.3 Summary of the Modifications to Fiasco/L4Re

The process of porting Fiasco to the SCC was intended to cause as little modification to the kernel as possible. However, in some cases kernel modifications were inevitable or other solutions not acceptable. For these situations, the general approach for extending or modifying the kernel functionality was to add conditional code sections, which can be switched on or off using the kernel configuration tool. To make Fiasco suitable for the SCC, four compiler switches were added:

- *CONFIG_NOSCREEN*: Using this variable one can disable graphical output completely.
- *CONFIG_NOKEYBOARD*: Disables keyboard driver (initialisation, timer functionality).
- *CONFIG_MCEMU*: All specific modifications, including disabled I/O requests and timer calibration, Local APIC interrupt handling and command-line parsing for frequency setting, take effect depending on this configuration.
- *CONFIG_MPBT*: When MPBT is intended to be used, this switch organises all necessary modifications (i.e., register flags, page table setting and disabled super pages).

3.1.4 Environment

To boot the binary produced by the build infrastructure, the data has to be loaded into the SCC's memory. Also, the Multiboot environment has to be established.

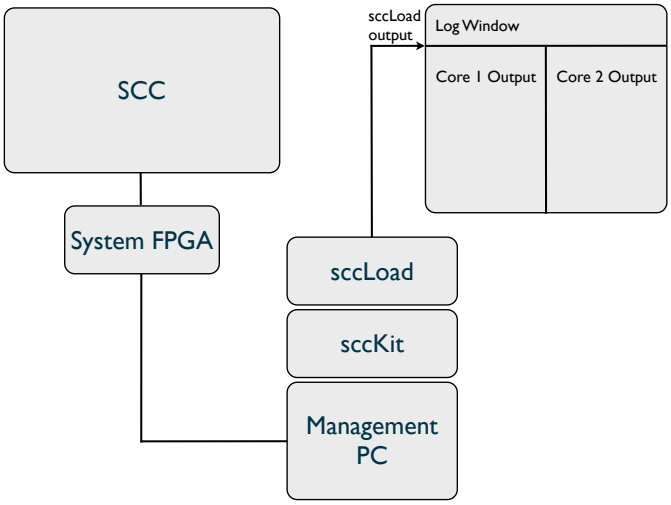


Figure 3.1: Environment setup for monitoring two Fiasco.OC instances

As described in Section 2.2, a management PC is available for transferring data to and from the chip, as well as for configuring the needed parameters. This PC is used to run a utility *sccLoad*, which was developed in the process of this work and is based on sccKit's *sccWrite*. This tool should incorporate all necessary steps to load, start, monitor and debug the software run on the SCC and was designed in three steps.

At first, the main use case of the utility is to transfer arbitrary binary data to the SCC's memory. The existing *sccWrite* was extended to accept a file input, the content of which is then written to the desired memory location.

Based on this initial functionality, *sccLoad* was extended to also prepare the boot environment (i.e., setup the Multiboot information structure and extract an ELF image) and boot a specific core to run the ELF image as well as handling its input and output requests. Additionally, a multiple instance setup was implemented, where several instances of the same ELF image can be run, monitored and controlled.

Figure 3.1 shows an exemplary scenario of such a setup with two instances, where the utility started two cores with the same image and then displaying their serial console output side-by-side.

3.1.4.1 I/O Handling

The rudimentary serial console emulation was implemented on the basis of Intel's softUART widget for *sccGui*.

Although the idea is quite straight-forward, simply redirecting all I/O port accesses to the implemented handler, the mechanism to route the requests and process them correctly did not prove to be very stable. Early experiments producing a lot of debug output suggested that race conditions and/or deadlocks seem likely to occur when a lot of requests are being processed (i.e., a lot of simultaneous text output by several instances). It was not possible

to let both cores start up at the same time, the resets are now released with a delay of five seconds. Otherwise, when booting two cores simultaneously, at least one core, sometimes both, did not produce any output. Furthermore, it sometimes happened that one of the instances hung in the boot process.

All in all, the mechanism of redirected I/O requests from the SCC chip to the management PC is an undocumented and obviously not very stable way to interact with the chip. The current implementation used for this work is thus in experimental stage and may be replaced by something more reliable in the future.

3.2 Communication

To investigate the potential held by the SCC for future software developments, benchmarking programs are needed for both DDR memory bandwidth and message passing performance to provide basic performance numbers and knowledge about specific difficulties when programming for the SCC, which can later be used to decide which way to go. In this section, two packages that were used to gather this data, are presented: An existing benchmark utility called *pingpong* was used for general performance measurements, while a new package called *scctalk* was implemented for experimenting with the message passing facilities of the SCC.

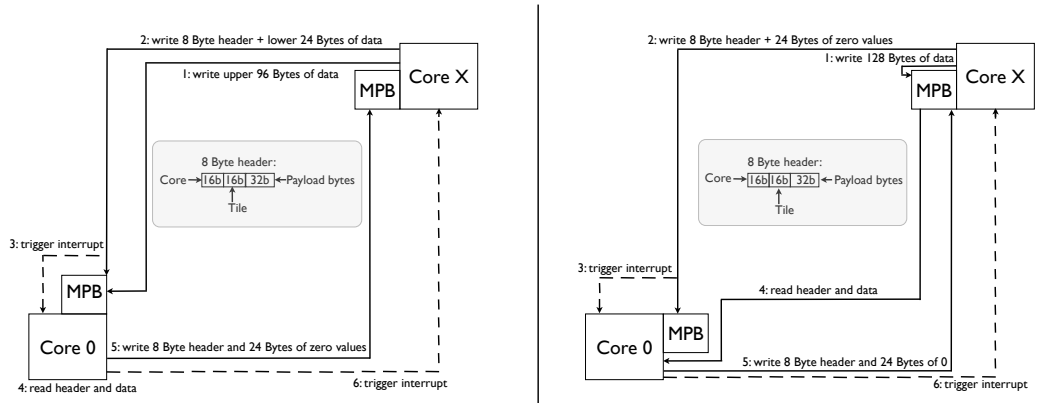
3.2.1 Pingpong Benchmark Utility

This existing benchmark suite contains tests for inter-process communication, various kernel activities like page faults and exceptions, as well as memory bandwidth with different implementations. Except for some `memcpy` implementations that make use of instructions not existent on the P54C, the funtionality could be used without modification to measure memory bandwidth.

3.2.2 Message-Passing Package

The message-passing package *scctalk* is used to evaluate the performance characteristics of various message passing implementations on the SCC on top of Fiasco. The general setup is that two cores on the chip are booted with this package, performing message passing between each other. In order to be able to vary the distance between the two instances, only one core is fixed to be core 0 of tile 0, while the other instance is placed on a core somewhere else on the chip. Both instances read their own tile register to find out their position, reacting accordingly: Core 0 waits to be messaged, the other core initiates the communication and performs the measurements. The exact mechanism of the communication can be varied in the data format, memory location and size, as well as notification method. When using the MPB as shared location, the memory that is used to transfer the actual data can be either the receiver's MPB (*Push*) or the sender's *Pull*.

In the following, the detailed design decisions regarding message format, memory location and notification are discussed.



(a) Push ***(b)*** Pull

Figure 3.2: Scheme for sending a 128 B packet one-way

3.2.2.1 Message Data Format

To construct a reliable communication mechanism in terms of correctness and completeness of received data, one would have to use some kind of higher level protocol design (e.g., TCP/IP). Otherwise, lost or unintendedly modified packages cannot be detected and dealt with. However, to assess the pure performance of the message passing infrastructure is difficult when using a protocol with potentially high overhead. Thus, the simplest type of protocol was used where the only overhead in a message packet is an eight byte information header, containing the sender's tile and core number, and the packet size. Although the minimal header size is only twenty bits, it was decided to use a structure that spans across eight bytes. Thereby, it is possible to only use writes of 64 bits and thus keep the transfer code as simple as possible.

Figure 3.2 depicts the message passing process for both *Push* and *Pull* using the example of sending a message of the described format, 128 Bytes in size, one-way from some core to core 0. The labels of the arrows describe which action is performed and in which order. Note that dashed lines are only valid when using interrupts.

The idea of this message format is to keep away as much influence on the raw message passing performance from the measurements as possible. Later, the gathered data could facilitate implementing a message-passing library, which protocol implementations can be based on.

3.2.2.2 Memory

The SCC provides two options for passing data between cores - shared off-tile DDR memory or the on-tile MPB memory.

Using some portion of the DDR memory as shared memory for message passing is possible, but has some disadvantages. The first problem one encounters is the missing L2 cache coherency. Although research is going on in the development of reliable software routines for maintaining coherency [15], it is not 100% tested and thus adds additional uncertainty

and pitfalls to the work. Additionally, not only the software cache coherency, but also the non-allocate-on-write policy is likely to lead to a performance problem.

Other research groups already presented results for another experimental implementation addressing this issue [25]. The new memory type MPBT is not restricted to use with the MPB, it can be used as well for DDR memory. Tagging DDR memory as MPBT results in memory accesses bypassing the L2 Cache, but at the same time write requests being combined in the WCB to full cache line writes.

Although this approach is an interesting topic, it was decided to investigate only the on-tile MPB memory. Its intention is specifically to accelerate message passing by having fast on-tile SRAM. The only downside is its size of only 16 KiB, which makes an evaluation of bandwidth for a larger amount of data very interesting, as it has to be packetised. In particular, the impact of packet size on the performance has to be investigated, because in later applications, potentially all 48 cores run an instance of an application and thus the MPB becomes a highly contended shared resource. Therefore, the packet size used to transfer a given amount of data has to be configurable in the program.

3.2.2.3 Notification

In order to pass messages between cores, the recipient has to be notified about the arrival of a message by the sender. There are two alternative ways of waiting for a message: Polling the message buffer for changed data, or blocking on an interrupt.

In interrupt mode, the program makes use of the existing library *libirq*. To get notified on the arrival of an interrupt, it is sufficient to bind to the interrupt number and then call a wait function which blocks the thread. As soon as the interrupt has arrived, the function returns and the thread gets woken up to process the request. To issue such an interrupt request, a simple *read-modify-write* function is used to toggle the interrupt bit in the core configuration register.

In contrast to interrupt-driven notifications, polling is a busy-waiting method. The wait function for a signal is implemented as follows:

1. Set the MPB header to an invalid state. This is done by setting the core number to two, the tile ID to 0x36 and the length of the message to five bytes. No real message can produce a header like this, as the core number has to be either 0 or 1, the highest tile ID is 0x35 and in our setup message sizes are always multiples of 32.

2. Invalidate the MPBT cache lines in L1 by calling the new instruction CL1INVMB

3. Read the MPB header bytes

4. Compare the header to the invalid state. If all three values changed, exit the wait function, otherwise go to 2.

Clearly, the interrupt mode has the advantage of reducing CPU usage while waiting for a message to arrive but carries the disadvantage of a potentially high latency due to the user-level interrupt handling. On the other hand, continuous polling could actually slow down the transfer of the message because of contention on the buffer.

Consequently, a comparison between the two methods is very interesting and therefore both were implemented in the message passing package.

As mentioned in the above description of the wait function, messages (and thus, in polling mode, also signals) have to be a multiple of 32 bytes in size. Due to the write combining it is necessary to evict the data from the buffer. This can be done by writing to an address outside the current cache line or to fill the WCB. As in initial experiments filling up the WCB was actually faster than writing to a non-consecutive location, always entire cache lines are written to the MPB.

While waiting for a message is more expensive than waiting for an interrupt, the notification gets simpler. There is no difference between a message and a signal anymore. The arrival of a message is signalled by the header appearing in the MPB. Hence, for sending a message containing actual data it is essential to write the first cache line, which contains the header information, after all other data.

Because in the polling mode, the MPB is accessed by both the sender and the receiver at the same time, it has to be taken care of possible race conditions and inconsistencies. A message can be sent only if the MPB was set to invalid first, and the waiting method is only allowed to return if all three values changed. In early tests it occurred that an inconsistent header was read, where for instance the core number and byte count was correct, but the tile ID was still 0x36.

3.2.3 Measuring Round-Trip Time and Throughput

The actual performance evaluation between two instances *A* and *B* is measured by instance *B*. Its location is varied on the chip from core 1 through 47, while *A* is fixed at core 0.

For measuring the round-trip time of a message, *B* sends a 32 byte message to *A* with its sender information. *A* receives the message, reads the content and immediately replies with another 32 byte message. After *B* has read the reply payload, the number of cycles needed is saved.

The throughput is determined by having *B* sending 32 MiB of data in variable packet sizes to *A*. Depending on the packet size, the resulting payload varies by the header size multiplied with the number of packets needed to transfer the 32 MiB of raw data. Packet sizes reach from the maximum of 8 KiB (the portion of the MPB assigned to each core at an even distribution for two participants) to a minimum of 32 bytes and can be chosen almost freely, as long as they are multiples of 32. This limitation is imposed due to the write combining. As the buffer only gets flushed when a full cache line is written or a memory location corresponding to a different cache line is written, it is a safe choice to only allow writes of 32 bytes.

As another interesting dimension of variation, the location of the sent data can be set to be either the sender's MPB or the receiver's. These strategies are referred to as *Pull* and *Push*, respectively, because the data is either pulled from the sender or pushed to the receiver.

In the push model, the sender just writes the data, including the header structure, into the receiver's MPB and, in the case of interrupt notification, sends a signal. The receiver can then read the sender information and data in one place. Consequently, an overhead of eight bytes per packet has to be taken into account when calculating the throughput.

Note that in the pull model, the location of the data and the header is separated. In order to find out the location of the data of a message, the receiver needs to have the sender's header structure in its own MPB. Using the provided tile and core number it is then possible to fetch the actual data from the sender's MPB. When using interrupts as notification mechanism, this results in our case in an additional overhead of 24 bytes per packet[1], compared to the push model.

According to Intel, the RCCE library uses the pull model in response to not further specified implementation problems with the push approach [19].

3.3 SCC-specific Obstacles

In the process of conducting the experiments for evaluation of performance and implementation particularities, two obstructions had to be resolved in order to get reliable results. In this section, the manifestation of these obstructions as well as the actions taken to overcome them are discussed.

3.3.1 Implementations of `memcpy`

The first unanticipated result of the experiments for evaluating the general DDR3 memory performance was a considerably worse throughput compared to that of a traditional P54C system. Given the higher core frequency and faster memory, the bandwidth was expected to be higher. The reason was discovered to be the *Write-Around* strategy of the cache. In simple copy or set operations, no cache lines were present and thus every write operation was sent to memory as individual transaction. The on-tile *Write Combining Buffer* (WCB) is not used for normal memory. Two countermeasures were implemented to evaluate the performance impacts of different improvements.

As a first simple modification it seemed obvious to compensate for the missing allocate-on-write by reading some byte of the destination cache line first. It is sufficient to read one byte every 32 bytes written. Clearly, this is not expected to be the fastest solution, because $\frac{1}{32}$ of read overhead is added.

Directly optimising `memcpy` to use this technique was used in iRCCE to improve message passing performance [4].

To make use of the existing WCB it is also possible to tag the DDR memory regions as MPBT. This results in bypassing the L2 Cache, but combining writes to 32 byte transactions to the memory controller. Note that as a consequence you lose the ability to use super pages entirely, as explained in Section 2.2.2. The problem with this approach is how to implement this memory type in a sound encapsulation. For the basic experiments, only hard-coded address ranges were used that automatically got the MPBT flag set. For later application one would want to have the memory allocation mechanism provide a new memory type mapping.

[1] Although the header is only eight bytes in size, a whole cache line has to be written due to write combining.

Address	Value
0x000000	00 01 02 03 04 05 06 07
0x000008	08 09 0a 0b 0c 0d 0e 0f
0x000010	10 11 12 13 14 15 16 17
0x000018	18 19 1a 1b 1c 1d 1e 1f

Table 3.1: Expected result when reading addresses 0x0 through 0x18.

Address	Value
0x000000	00 01 02 03 04 05 06 07
0x000008	00 01 02 03 04 05 06 07
0x000010	00 01 02 03 04 05 06 07
0x000018	00 01 02 03 04 05 06 07

Table 3.2: Result when reading addresses 0x0 through 0x18 with the content of Table 3.1.

3.3.2 Uncacheable I/O Memory vs. MPBT

The next step after having assessed the various performance properties of the main memory was to use the MPB. As discussed in Section 3.1, the MPB is seen as I/O memory provided by the *io* package on a virtual PCI bus. Thus, this package was made aware of the address range where the MPB memory resides by using a configuration file. Now the library *libio-io* can be used to map this memory into the program's address space. Traditionally, such a mapping of I/O memory is flagged uncacheable (UC), as it is common use for devices. However, this turned out to be a major issue on the SCC. When reading MPBT+UC memory, the result is not correct. Instead of having access to all data (like in Table 3.1), it is only possible to access the first eight bytes. Reading from addresses that are not 32 Byte aligned results in getting the wrong data, as shown by the example in Table 3.2.

Although discussion [20] is going on about the root of this problem and whether this is intentional behaviour, it is very likely that it is a hardware bug. As a consequence, the *io* package had to be modified to implement an additional flag in order to map cacheable or uncacheable memory depending on the configuration of the memory region.

4 Evaluation

In this chapter, an evaluation of the developed components is given. The evaluation focuses not only performance results, but examines also robustness and sustainability of the implementations. In the following sections, the main parts of this work are analysed: First, the results regarding main memory are presented. Second, the performance results obtained in the core field of investigation of this thesis, the inter-core message-passing, are discussed. The chapter ends with a short evaluation of the developed software.

4.1 General Memory Bandwidth

As mentioned in Section 3.3.1, different approaches to compensate for the non-allocate-on-write performance hit were investigated. Measurements were taken for three memory functions (`memcpy`, `memset`, `memzero`), where possible. In the case of the iRCCE implementation and the MPBT for destination only, there is currently no `memset` or `memzero` implementation available. To gather data for comparison, the benchmarks for these functions were at first run unmodified on both the SCC and an original P54C system (System configuration: 133 MHz Pentium P54C, 64 MiB of 100 MHz DDR-SDRAM). As Figure 4.1 shows, the SCC's memory performance clearly suffers from the non-allocate-on-write policy of both the L1 and L2 cache. On the original system, the L2 cache does allocate a cache line upon write miss and thus the performance is much better. The results for the modified function implementations show that the performance on the SCC can be significantly improved by adding read requests before writing. For this implementation the term Manual Allocate On Write (MAOW) is introduced for use in this thesis. About the same performance could be achieved by using iRCCE's `memcpy` implementation.

Another countermeasure to the non-allocate-on-write problem is to tag the memory as MPBT. However, Fiasco.OC's memory allocation mechanisms currently don't support MPBT and the integration of this new memory type in a sound environment is a complex work. In order to assess the potential improvements, a temporary code section in the kernel was used for testing purposes only. By manually configuring a fixed memory section to be MPBT and then requesting a memory mapping for exactly this physical region in the user-mode application *pingpong*, the memory performance could be measured analogously to the MAOW implementation. In Figure 4.1, the results for tagging either only the destination of the `memcpy` operation or both source and destination as MPBT are shown. These results are comparable to those achieved by other research groups [25]: Copying from MPBT to MPBT memory is the fastest solution to the write miss issue. As pointed out in the paper, bypassing the L2 cache removes a delay and thereby further improves the throughput.

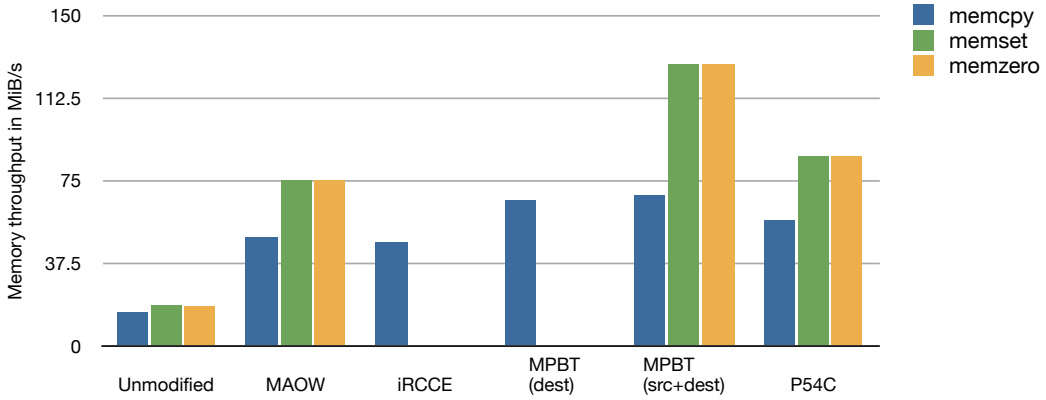


Figure 4.1: DDR memory throughput for different implementations

This result brings message passing through shared DDR memory back into consideration, because using MPBT is not only the fastest solution, but also eliminates the need for an L2 cache flush routine.

4.2 Round-Trip Time and Throughput Measurements

The experimental setup for evaluating the message passing performance, which is described in Section 3.2.2, was used in two variations to measure round-trip time as well as throughput.

As mentioned in Section 2.2, the individual cores on the SCC are arranged in a 6x4 array of two-core tiles. To estimate the impact of the distance of two instances on the chip, the round-trip time was investigated by sending 32 bytes from every core to core 0 and back. This was done 20 times, while the first two results were excluded as warm-up runs. The average of the remaining 18 runs is taken as the round-trip time between the respective pair of cores.

To measure throughput, a large message of 32 MiB was sent in variable packet sizes one-way from every core to core 0. To restrict the focus of the measurements to the mesh performance, the same packet was sent as often as needed for transferring at least 32 MiB rather than sending different data in each packet. Thus, the cache size does not compromise the throughput. The experiment was run 10 times for every core 1 through 47 and packet sizes ranging from 32 to 4096 bytes. After discarding the first result, the remaining 9 values form the average throughput for the measured core and packet size pair. Additionally, all these measurements were taken for both the pull and the push implementation.

Round-trip time and throughput experiments were run in polling as well as interrupt notification method.

It was expected to confirm with the results from these experiments a steady growth in throughput for increasing packet sizes, because sending fewer packets decreases the overhead needed to send the whole message. On the other hand, a farther distance on the

chip (i.e., more mesh network hops) is anticipated to increase round-trip time and thereby decrease throughput.

In an early stage of these experiments, a significant anomaly in the throughput graph was discovered. In a certain implementation, a peak throughput could be seen for a packet size of 1024 bytes. Further investigation of this issue lead to the result that the implementation has a large impact on the achievable performance due to the *non-allocate-on-write* policy described in Section 3.3.1.

4.2.1 Impact of Implementation on Performance

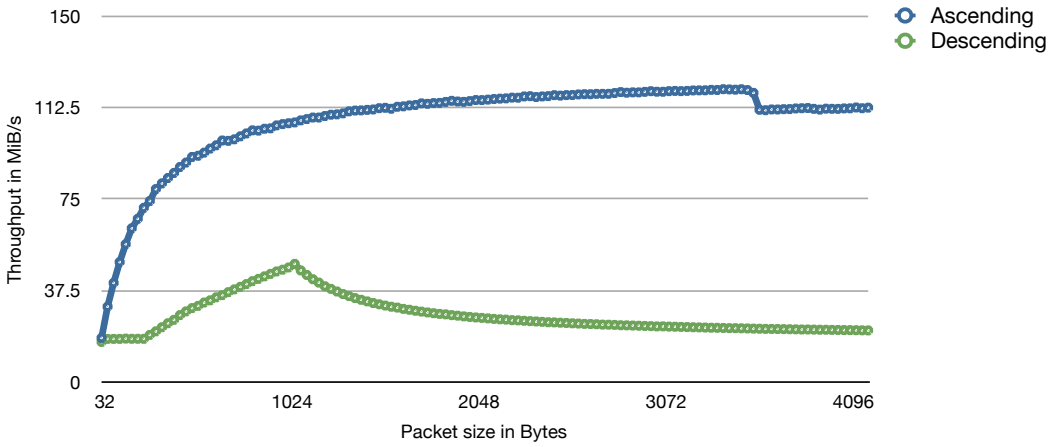


Figure 4.2: Throughput for ascending and descending packet sizes from core 1 to core 0 (polling mode)

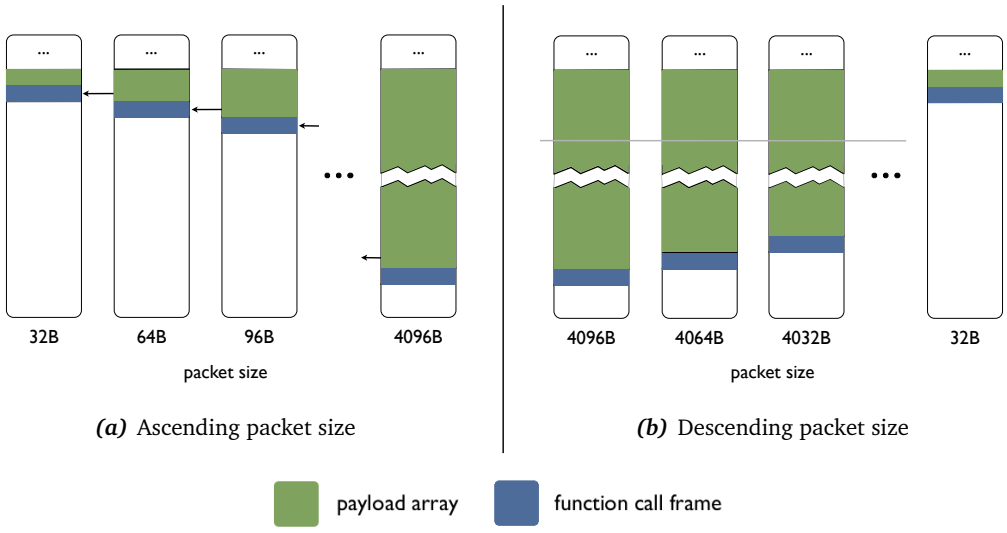
The above anomaly manifested itself as follows:

- When measuring one packet size for one core per benchmark run, a peak throughput emerged for a packet size of 1024 bytes.
- When measuring all packet sizes in a loop from 32 bytes to 4096 bytes for one core per benchmark run, the throughput grew monotonically as expected.

Given these facts for basically the same implementation lead to the hypothesis that the individual measurements influenced each other. To investigate the cause of this problem, the packet size loop was used in ascending as well as descending order. The result of this experiment confirmed the hypothesis: The ascending version produced results significantly differing from the descending loop, as shown in Figure 4.2. Also, the problem only occurred when copying the data from the MPB to some location on the stack. Using a static variable (and thereby forcing the data to be located in the data section) immediately eliminated the anomaly.

These findings led to the assumption that the problem is caused by the non-allocate-on-write cache policy in connection with the measurement implementation. To confirm this, a large `memcpy` operation of 8 MiB between each loop step on the receiver side was introduced in the descending loop using a stack variable. Afterwards, the throughput curve looked almost the same as the original ascending loop curve. The only difference was a generally lower throughput, which can be explained by the cache practically being flushed before each run. This proved that the deviation of the descending results from the ascending ones is caused by cache lines being allocated in the ascending version as opposed to the descending loop.

To understand this behaviour, it is helpful to have a closer look to the stack in the process of receiving a message. When using ascending packet sizes, the stack evolves as illustrated in Figure 4.3(a). Before copying the received data, an array of the message size is allocated. Afterwards, the function `readPayload()` is called to copy the message content into the array. When calling a function in C, the stack is filled with the function parameters, the return address and previous stack pointer, as well as local variables of the function itself (*source!*). When returning from the function, the return address and the saved stack pointer are read and removed from the stack, which at the latest then causes the corresponding cache line to be allocated. In the next step of the loop, the message data array is 32 bytes larger than before. Thus, at least the first 32 bytes of the message are written into the cache and are not affected by the *non-allocate-on-write* policy. Another step in the loop adds another 32 bytes to the data array and on its part allocates a new cache line, and so on.



(a) Ascending packet size

(b) Descending packet size

Figure 4.3: Stack configurations inside `readPayload()` function

In other words, every packet size run arranges for the following one to write the message data into cache lines and therefore reaching a maximum throughput. This is not the case for descending packet sizes. As illustrated in Figure 4.3(b), in each run, the cache line caused to be allocated by the previous run is not used anymore, because the data array size

decreased. Thus, all runs are affected by the write around behaviour. The curve progression can then be explained by taking into account that function calls prior to the actual message retrieval already caused some cache lines to be allocated for stack locations, qualitatively indicated by the gray line in Figure 4.3(b). By reducing the packet size, the percentage of memory locations not present in the cache recedes, causing the throughput to increase until the overhead needed to send the packets becomes too high and the curve follows the anticipated trend.

This anomaly was eliminated by the `memcpy` operation copying 8 MiB from one array into another after each loop cycle. Doing so basically flushed the caches and prevents experiment runs from influencing each other. Although the source of the issue can be considered as a software bug, it shows how important it is to keep in mind the *non-allocate-on-write* policy when implementing the message passing, as the deviation in throughput compared to the expectations can be very high.

4.2.2 Mesh Network Investigation

In order to assess the performance degradation when moving communication partners farther away from each other, a set of benchmarks were run where the sent data was only read, not copied into private DDR memory. By doing so, an evaluation of the mesh network and MPB performance is possible. Additionally, the Round-trip time was measured in this context, as it also characterises the network performance.

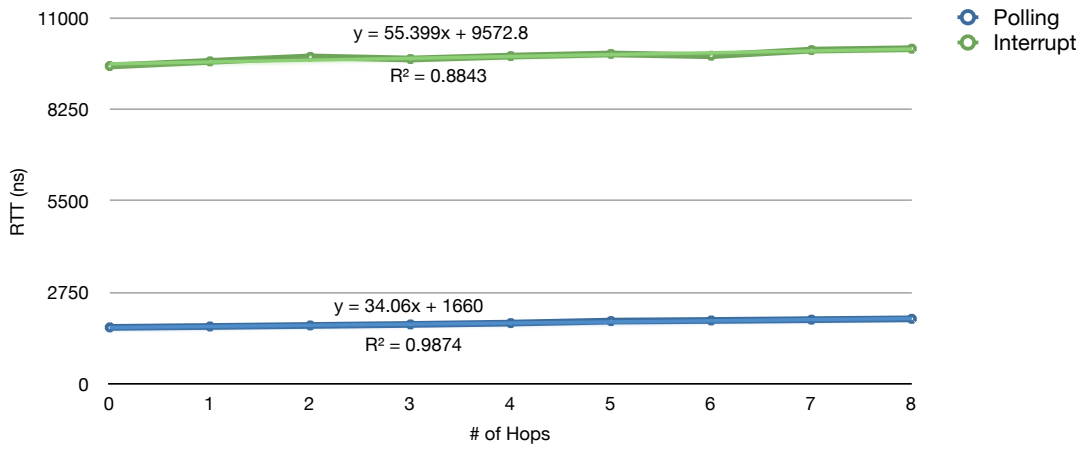


Figure 4.4: Round-trip time of a 32 Byte message depending on mesh network hops

The round-trip time was expected to grow linearly as the number of hops in the mesh network increases. Similar to experiments using RCCE [24], the growth should be in proximity to 30 ns. The router latency of 4 mesh cycles, multiplied by 3 packets (1 header, 2 data) in each direction, results in a total of 24 mesh cycles, being 30 ns for a mesh clock of 800 MHz. The results for round-trip times (RTT), averaged over the times for the same number of network hops, are shown in Figure 4.4. Values range from 1698 ns to 1955 ns for

polling and from 9561 ns to 10082 ns for interrupt notifications. The embedded linear regression verifies the anticipated linear correlation of the number of hops and the round-trip time with a slope of 34.06 ns for polling mode. For interrupt mode, the slope is higher, because triggering the interrupt is also involved, which adds the latency of another 2 packets per direction. The gap of on average 8019.5 ns between polling and interrupt notification can be explained by the interrupt latency due to user-level interrupt handling. In supplementary measurements, the time between entering the kernel interrupt handler and exiting the wait function of the receiving thread was discovered to be around 3940 ns. This latency occurs twice in the round-trip time experiment, thus the overall latency due to interrupts is at around 7880 ns. The remaining time is needed for the interrupt to be actually triggered and delivered to the processor.

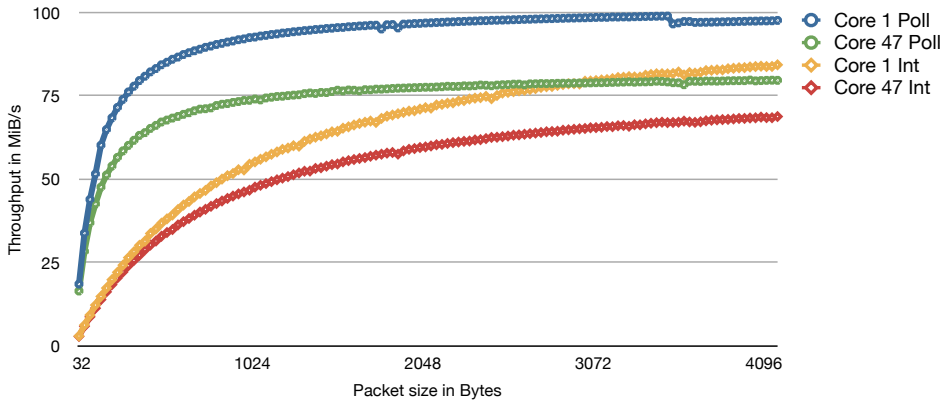
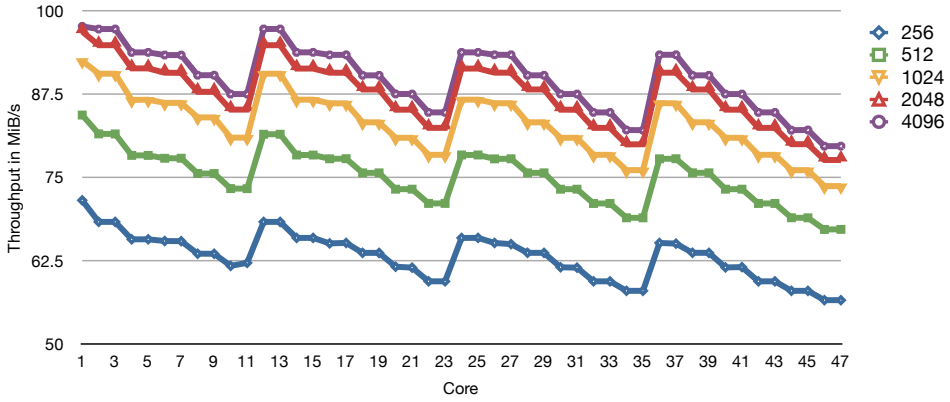


Figure 4.5: Throughput for a 32 MiB message depending on the packet size

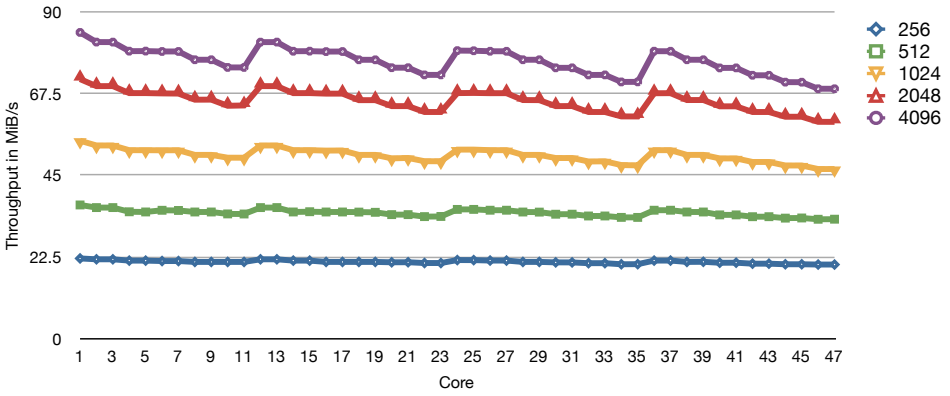
Figure 4.6 compares the measured throughput from every core to core 0, for interrupt and polling notification method. In each graph the curves for the packet sizes 256 B, 512 B, 1024 B, 2048 B and 4096 B are shown. Not surprisingly, larger packets yield a higher performance. However, the increase flattens faster for polling than for interrupt notification, which can be explained by the higher impact of the interrupt latency. As illustrated by Figure 4.5 for cores 1 and 47 as communication partners, the curves for polling and interrupt tend to roughly the same value, but using interrupts, the latency slows down the throughput increase. Clearly, if one could use a packet size of 32 MiB, the two curves would only differ non-significantly.

The gathered data reflect exactly the expected behaviour. With each network hop, the throughput decreases by roughly the same amount. The deviations where the decrease is smaller, to be seen for cores 4-7 as well as 14-17 and 24-27, are neglected. They appeared consistently for various measurements at different points in time and could be caused by hardware inconsistencies.

The curves can be viewed as a traversal over the chip, beginning with core 1 at the lower left corner, moving to the right. At the end of a row in the array of tiles, the traversal jumps



(a) Polling



(b) Interrupt

Figure 4.6: Throughput for a 32 MiB message sent in 4096 B packets depending on the position on the chip

to the left of the next higher row. These jumps can be seen in the graphs, when for instance by moving from core 11 to core 12 the throughput increases to the value of cores 2 and 3. This is expected because these four cores have the same network distance to core 0.

For polling notification, the measured values range from a maximum of 97.6 MiB/s to a minimum of 79.7 MiB/s. In the interrupt case, the performance drops to values between 84.3 MiB/s and 68.8 MiB/s. The differences of 18 MiB/s and 15.5 MiB/s, respectively, show that the small increase in the round-trip time leads to a significant decrease in throughput.

4.2.3 Realistic Message Passing Measurements

In reality, only reading the received data is not very likely. Instead, the receiver would copy the data to some private DDR memory location in order to be able to process the data and to receive the next message. As already discovered in Section 4.2.1, the measurements showed various difficulties in response to implementation details. The experiment was thus modified to use the MAOW implementation together with a software cache flush to eliminate all impacts by the caches. It is expected that this solution does not reach a throughput as high as in the read-only case, as a dependence on the memory bandwidth is introduced. Additionally, MPBT tagged DDR memory instead of MAOW could still reach a higher performance, but this is a memory type not yet supported by Fiasco.OC and was therefore not used.

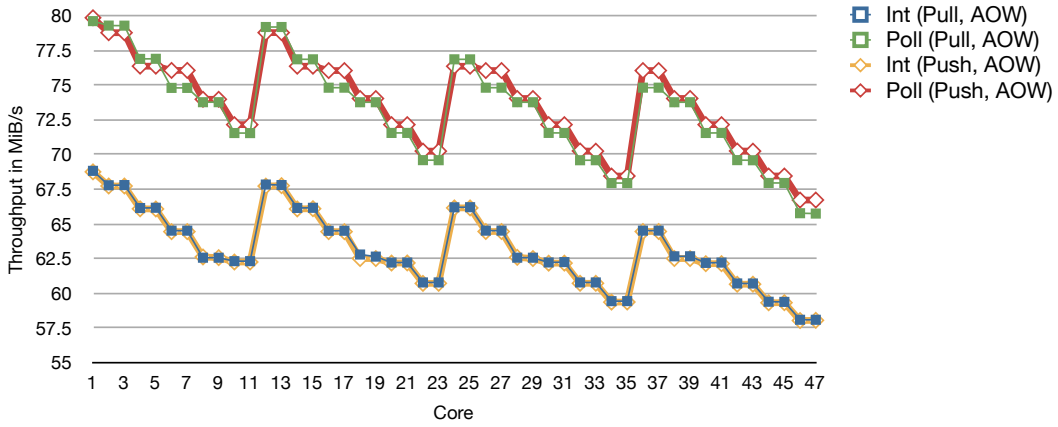


Figure 4.7: Throughput for a 32 MiB message sent in 4096 B packets depending on the position on the chip

The main purpose of these measurements was to assess again the impact of the mesh network, but also compare the *push* and *pull* models. In Figure 4.7, the two models are plotted against each other for both polling and interrupt notification. The yellow and blue curves (interrupt-driven push and pull model) are, except for negligible deviations, completely identical. Only little higher, but still non-significant discrepancy can be seen for the red and green curves (polling push and pull model). Again, the curves follow the number of network hops, while interrupt notifications cause a lower throughput, but also less decrease per hop.

The comparability of push and pull can be seen again in Figure 4.8. The curves are following the same trends as in the read-only case and there is no relevant difference between the two models.

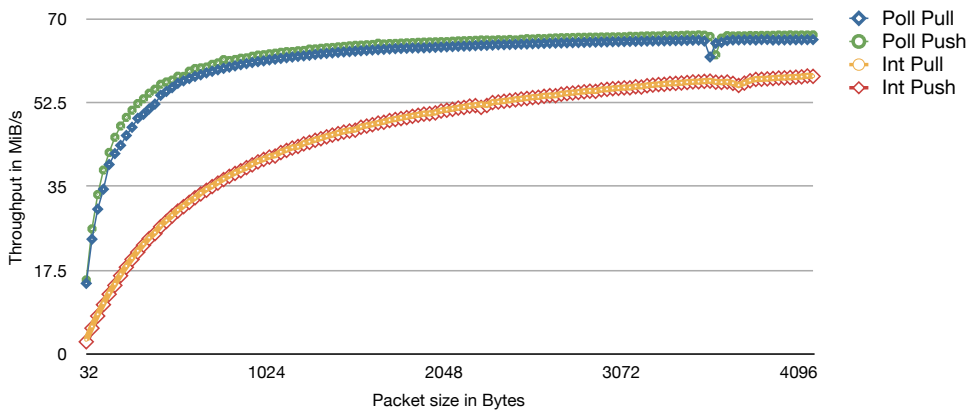


Figure 4.8: Throughput for a 32 MiB message depending on the packet size

4.3 Software Evaluation

The software developed in order to perform memory and message passing experiments proved to be a more complicated exercise than expected.

The current version of *sccLoad* is able to setup, monitor and control one or two instances of an arbitrary ELF image and could be easily extended to support up to 48 instances. However, it still suffers from boot sequence interruptions due to I/O errors that are not yet completely investigated. In a future version of the program, it would be better to separate the task of setting up the boot environment from the monitoring and controlling action and also allowing different ELF images to be loaded onto different cores.

The message passing package is by no means a product prototype, but rather an experiment to reveal the pitfalls and assess the possibilities and consequences of design choices when implementing a real message passing library. It was able to point out the importance of the *non-allocate-on-write* policy as well as the impact of mesh network hops on the performance.

5 Future Work

The problems described in Section 3.1.4.1 suggest thinking about alternatives to the input and output handling implemented in this work. The unreliable I/O port handling leading to aborted booting sequences is not a sufficient environment to conduct excessive experiments. Because also other research groups faced some I/O related problems [21], developing something not relying as much on I/O ports might be a more promising approach. For output, the virtual text buffer can be used like it is done in *sccGui*.

For interacting with the instances, SSH access is a desired goal. To achieve this, also a network driver needs to be developed that uses the on-board devices for communication with the MCPC and thus with the internet.

Besides the environment, also the message passing research can be continued. Choices include a combination of a message passing library and an L4Re server multiplexing the MPBs as well as a RCCE port. Given the issues discovered while implementing the basic message passing application, both possibilities are an interesting, yet time-consuming field of investigation. Building a sound message passing environment from scratch needs to be thoroughly planned in order to incorporate the results of this work, but would be at the same time a promising solution with regard to performance and integration into a microkernel system. But also the comparison to the port of a well-tested library (i.e., RCCE) is a serious possibility.

Independent of the chosen software, a hybrid notification method, where an interrupt is sent to the receiver prior to finishing the message transfer, can be investigated. The receiver would then start polling for the message to arrive completely. This way, the disadvantages of both methods can be mitigated.

Also related to message passing is the performance issue discussed in Section 4.2.1, leading to the question how to compensate for this shortcoming of the cache architecture and where to implement the improvement. An important step in this process is to integrate the MPBT memory type in the memory allocation mechanisms of Fiasco.OC and L4Re.

Another topic in relation to communication is the inherent security issue when allowing all cores to access every memory location with full rights. In the standard layout, each core can read and modify all configuration registers, MPBs and DDR memory. By modifying its own LUT entries it can basically control the entire system. When the MPB is used for communication, there is no means to prevent cores from tampering with MPB data once they have access to the MPB locations.

Exploring ways to establish security measures could be an interesting topic. For example, one could think of sandboxing cores by not giving them access to configuration registers. This, of course, would render inter-core interrupts, which are implemented using those registers, impossible. A slightly weaker restriction, allowing access to all configuration registers but the cores' own one, would enable interrupts again but also gives the cores the ability to mess with other cores.

Instead of the configuration registers, also the functionality of a global interrupt controller that was recently added to *sccKit* could be used along with the other additional features (global timestamp counter, global atomic increment counter), which were not implemented in the course of this thesis.

In addition to covering performance issues, we the SCC is a good platform to research energy-related issues, because it provides regulators allowing to modify tiles' frequency and voltage settings. These regulators also allow for controlled *undervolting*, which may lead to energy savings as well as failures with yet unknown characteristics. An evaluation of the concrete manifestations of such failures and tradeoffs between power saving and fault probability in future experiments pose a challenging research topic.

6 Conclusion

Although porting Fiasco.OC to the SCC seemed straight-forward, several problems occurred that made the port more difficult than expected, such as bugs in the sccKit and cache misbehaviour. Nevertheless a working environment for experiments with microkernels on many-core systems could be established in order to investigate particularities and problems that can occur on such systems, as well as possible approaches to solve them.

After building an environment to boot Fiasco.OC with as little modifications as possible and extending the existing functionality to harness the SCC characteristics, experiments could be conducted to evaluate memory as well as message passing performance.

The performance anomalies resulting from the *non-allocate-on-write* caches could be addresses by separating the experiments from each other through cache flushes and by implementing either MPBT tagged DDR memory or a modified `memcpy` featuring MAOW.

The results of the message passing experiments show all in all the expected behaviour regarding the mesh network performance and yielded valuable background information for future work on the SCC.

However, although the results were consistent with the expectations, the process of obtaining them was derailed by the SCC's particularities and quirks at several stages. For instance, the hardware bug concerning uncacheable MPBT memory and mostly the instability of I/O posed a major obstacle to the work. Given these caveats and the relatively poor performance of some components (e.g., memory) as well as the quite aged feature set of the P54C architecture, it is not clear whether or not to continue the work on this particular chip.

Acronyms

AOW	Allocate On Write
API	Application Programming Interface
BIOS	Basic Input Output System
ELF	Executable and Linkable Format
EOI	End Of Interrupt
FPGA	Field-Programmable Gate Array
IPI	Inter Processor Interrupt
Local APIC	Local Advanced Programmable Interrupt Controller
LVT	Local Vector Table
LUT	Lookup Table
MAOW	Manual Allocate On Write
MARC	Many-Core Applications Research Community
MCPC	Management Console PC
MIU	Mesh Interface Unit
MPB	Message-Passing Buffer
MPBT	Message-Passing Buffer Type
OS	Operating System
PCI(e)	Peripheral Component Interconnect (Express)
PIC	Programmable Interrupt Controller
PIT	Periodic Interrupt Timer
PSE	Page Size Extension
RTC	Real-Time Clock
RTT	Round-Trip Time
SCC	Single-Chip Cloud Computer
SIF	External System Interface Controller
SMP	Symmetric Multiprocessor System
SSH	Secure Shell
TSC	Timestamp Counter
UC	Uncacheable
WCB	Write-Combine Buffer

Bibliography

[1] Tool Interface Standard - Executable and Linkable Format. `www.skyfree.org/linux/references/ELF_Format.pdf`, 1998.

[2] Andrea Bartolini, MohammadSadegh Sadri, Francesco Beneventi, Matteo Cacciari, Andrea Tilli, and Luca Benini. SCC Thermal Sensor Characterization and Calibration. In *Proceedings of the 3rd MARC Symposium, KIT Scientific Publishing*, Ettlingen, Germany, July 2011.

[3] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Recent Advances and Future Prospects in iRCCE and SCC-MPICH – Poster Abstract. In *Proceedings of the 3rd MARC Symposium, KIT Scientific Publishing*, Ettlingen, Germany, July 2011.

[4] Carsten Clauss, Stefan Lankes, Thomas Bemmerl, Jacek Galowicz, and Simon Pickartz. iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer. `http://www.lfbs.rwth-aachen.de/publications/files/iRCCE.pdf`, November 2011.

[5] Marco Fais and Francesco Iorio. Fast fluid dynamics on the single-chip cloud computer. In *Proceedings of the 3rd MARC Symposium, KIT Scientific Publishing*, Ettlingen, Germany, July 2011.

[6] Bryan Ford and Erich Stefan Boleyn. Multiboot Specification version 0.6.96. `https://www.gnu.org/software/grub/manual/multiboot/multiboot.html`.

[7] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Symposium on Operating Systems Principles*, pages 66–77, Saint Malo, France, 1997.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Elsevier, 5, illustrated edition, 2011.

[9] Intel Corporation. *Pentium Processor Family Developer's Manual*, volume 3: architecture and programming manual edition, 1995.

[10] Intel Corporation. *Datasheet for Intel Pentium processor 75/90/100/120/133/150/166/200*, June 1997.

[11] Intel Corporation. Excerpts from A Conversation with Gordon Moore: Moore's Law. `ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf`, 2005.

[12] Intel Corporation. Single-chip Cloud Computer. http://communities.intel.com/servlet/JiveServlet/downloadBody/5031-102-1-8024/SCC-Overview.pdf, 2009.

[13] Intel Corporation. *SCC External Architecture Specification (EAS)*, revision 1.1 edition, November 2010.

[14] Intel Corporation. *The SccKit 1.4.x User's Guide*, revision 1.0 edition, October 2011.

[15] Ted Kubaska and Michiel van Tol. Are you using the new l2 cache flush routine? http://communities.intel.com/message/121371, April 2011.

[16] Adam Lackorzynski and Alexander Warg. Taming Subsystems: Capabilities as Universal Resource Access Control in L4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pages 25–30, Nuremberg, Germany, 2009. ACM.

[17] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, California, USA, June 2007.

[18] M. Papamarcos and J. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. 11th ISCA*, 1984.

[19] Markus Partheymüller. Different performance on same tile? http://communities.intel.com/message/138140.

[20] Markus Partheymüller. Strange behaviour when reading MPB. http://communities.intel.com/message/133047, July 2011.

[21] Jan Arne Sobania. Connection to SCC breaks if I/O load is high. http://communities.intel.com/message/105177.

[22] TU Dresden OS Group. Fiasco.OC microkernel. http://os.inf.tu-dresden.de/fiasco/, 2010.

[23] TU Dresden OS Group. L4 runtime environment. http://os.inf.tu-dresden.de/l4re/, 2010.

[24] Rob van der Wijngaart and Tim Mattson. Getting the Most from Your SCC: An interactive exploration of SCC for SCC users. http://communities.intel.com/servlet/JiveServlet/downloadBody/6257-102-1-9488/how_to_get_the_most_from_your_scc_intel_mattson_vanderwijingaart.pdf, March 2011.

[25] Michiel W. van Tol, Roy Bakker, Merijn Verstraaten, Clemens Grelck, and Chris R. Jesshope. Efficient Memory Copy Operations on the 48-core Intel SCC Processor. In *Proceedings of the 3rd MARC Symposium, KIT Scientific Publishing*, Ettlingen, Germany, July 2011.