Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 4

Systems Group, Department of Computer Science, ETH Zurich

Support for heterogeneous cores for Barrelfish

by

Dominik Menzi

Supervised by

Adrian Schuepbach
Andrew Baumann
Timothy Roscoe

4. Oct 2010 - 3. Jul 2011

inf | Informatik
Computer Science

Contents

As computer hardware becomes more powerful and cheaper, hardware other than the CPU becomes suitable to run applications. This has been used to offload work to devices that are more suited for the workload, like highly parallelizable computing workloads that are offloaded to graphics cards.

Barrelfish is a "Multikernel", a new way of building operating systems that treats the inside of a machine as a distributed, networked system, and runs a different kernel, or "CPU driver", on every core.

This possibly makes Barrelfish highly suitable to run not only on a homogeneous set of CPUs and cores, but to include processors on peripheral devices, which may be of different architecture.

In this thesis I present two extensions to Barrelfish that allow it to be run on a heterogeneous set of CPU cores, either using x86_64 and x86_32 cores in one machine or using Intel's Single-Chip Cloud Computer to extend the set of cores available in a machine.

1 Introduction

1.1 Motivation

As computer hardware becomes more powerful and cheaper, hardware other than the CPU becomes suitable to run applications. This has been used to offload work to devices that are more suited for the workload, like highly parallelizable computing workloads that are offloaded to graphics cards.

Barrelfish is a "Multikernel" [BBD+09], a new way of building operating systems that treats the inside of a machine as a distributed, networked system, and runs a different kernel, or "CPU driver", on every core. Barrelfish also uses techniques like logic programming and constrained optimization to reason about hardware and devices [SSBR08].

This gives gives an approach to consider not only offloading applications and specific workloads to different devices, but possibly have the operating system span diverse hardware with different architectures, such as CPUs, GPUs, programmable NICs or other devices suitable for running an operating system.

In this thesis, I will demonstrate two versions of barrelfish that are able to run on heterogeneous systems. The first version, described in chapter 3, can run on x86_64 and x86_32 processors simultaneously. The second version, described in chapter 4, can run on a x86_64 core on a host PC and on a x86_32 core on a Intel Single-Chip Cloud Computer (SCC) simultaneously, connected by a Peripheral Component Interconnect Express (PCIe) bus.

2 Background

2.1 Barrelfish

Barrelfish is an operating system built in the Systems Group at ETH Zurich. It is built using a new architecture called "Multikernel" [BBD+09]. The kernel of the operating system is very similar to that of a microkernel in that it only provides the most basic functionality.

The multikernel differs from a microkernel based operating system in the way it works on multicore systems. Instead of having the kernel span all cores and protecting critical datastructures by locks from concurrent access, the multikernel runs a separate kernel on every core in the system.

In Barrelfish, these kernels do not communicate in any way with each other. On top of each kernel runs a service called "monitor", which creates connections to other monitors in the system and provides the basic functionality for applications to create connections to local and remote applications, drivers and other services.

2.1.1 IDC / Flounder

In Barrelfish, applications communicate over the inter-dispatcher communication (IDC) system. At its heart lies Flounder, an interface definition language. Flounder enables application writers to easily define interfaces to their services. A Flounder interface definition mainly consists of a number of definitions of messages.

From the interface definition, Flounder generates stub functions to call in an application and also implementations for each backend that was selected for a given application and architecture. At runtime, a connection is established using the appropriate backend implementation depending on the availability of hardware features such as shared memory, inter-processor interrupts and cache coherence.

After the binding process, the application can bind its own callbacks for receiving messages. When receiving a message, the callback is called with the arguments sent by the other side of the connection without the need of any conversion, extraction from or other manipulation of data structures. While a connection is initiated by a client, connecting to a server that exported the interface, the distinction between client and server disappears when the connection is established.

For Flounder, there exist a number of backend implementations that have different hardware requirements and purposes. For a list of the currently available backends, see table 2.1.

Name	Description	Requirements for connection endpoints
LMP	Local Message Passing	• Endpoints on the same core
UMP	User-level Message Passing	• Endpoints can share memory • Cache coherence for shared memory
UMP_IPI	User-level Message Passing with inter-processor interrupts (IPIs)	• Endpoints can share memory • IPIs between endpoints
BMP	Beehive Message Passing	• Only available on the Beehive processor
SIFMP	SIF Message Passing	• Protocol for communicating between Intel Single-Chip Cloud Computer (SCC) cores and connected host processors • see section 4

Table 2.1: Flounder backends

2.2 SCC

2.2.1 Overview

The Intel SCC, codename Rock Creek, is an experimental chip developed by Intel Labs [1]. Its name comes from the architecture of the SCC, which resembles that of a datacenter. The chip consists of 24 tiles, of which each contains two Intel P54C cores. The tiles are placed in a 6x4 mesh, connected by routers on each tile (see figure 2.1). The mesh is used for communication with other cores, connected hardware and for memory access. The SCC does not provide cache coherence.

The SCC has 4 DDR3 memory controllers, placed on the edges of the mesh, which control up to 16 GB of memory each, resulting in a maximum of 64 GB system memory. Additionally, there is message passing buffer (MPB) memory on each tile, giving each tile 16 kB of fast, local SRAM.

Each tile is connected to the mesh via a Mesh Interface Unit (MIU), that catches accesses to memory and translates *core addresses* to *system addresses*. It acts as a physical-to-physical memory mapping table and converts addresses using a lookup table (LUT). The LUTs are normally pre-configured by the host PC, but entries can also be changed at runtime by the operating system (OS). Entries in the LUT can point to memory regions in any of the 4 memory controllers, local and remote MPB memory, local and remote configuration registers and hardware interfaces.

[1] http://techresearch.intel.com/ProjectDetails.aspx?Id=1

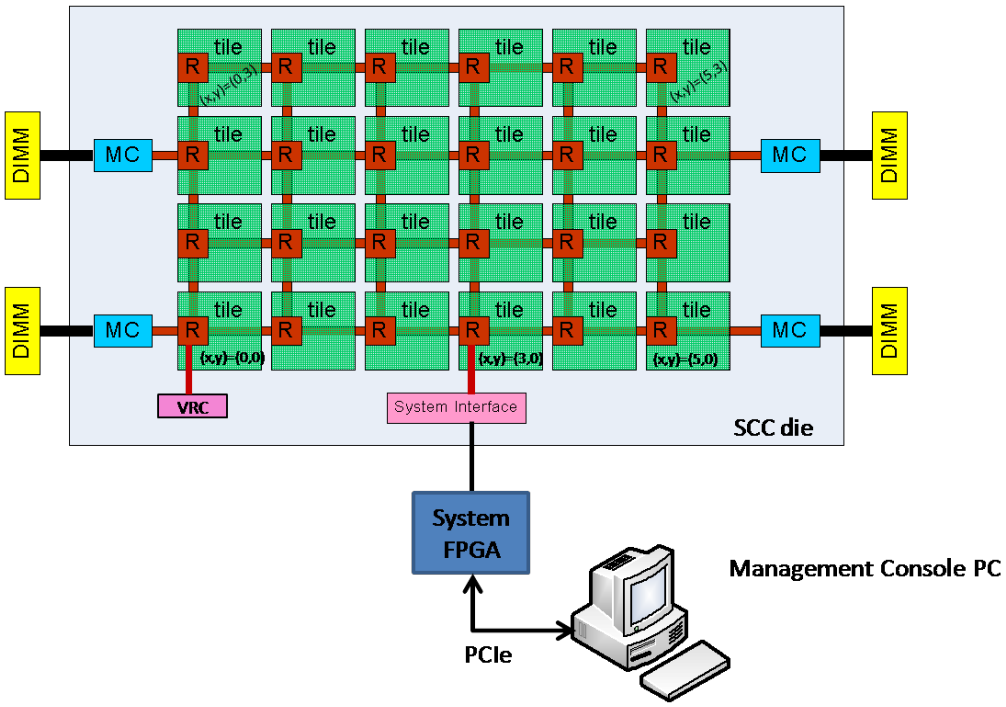

Figure 2.1: SCC Top-Level Architecture

The cores are numbered from 0-47, from bottom left to top right

From SCC External Architecture Specification, Revision 0.94

On memory access, the MIU forges a packet with the help of the LUTs and either sends it to a local destination (MPB, local configuration register) or forwards it to the routher on the tile, which forwards it to the appropriate destination. Destinations consist of the coordinates of a tile and a subdestination number (see table 2.2).

Subdestination	Number	Comment
Core 0	0x0	Not a destination for memory R/W
Core1	0x1	Not a destination for memory R/W
CRB	0x2	Configuration Register
MPB	0x3	Message Passing Buffer
East port	0x4	@(5,0) and @(5,2) is DDR3 MC
South port	0x5	@(3,0) is SIF, @(0,0) is VRC
West port	0x6	@(0,0) and @(0,2) is DDR3 MC
North port	0x7	Nothing is on this port of any edge router

Table 2.2: SCC subdestination values

To facilitate the efficient use of MPB memory, Intel added an additional bit in the page tables (virtual-to-core-physical) which marks memory as message-passing buffer data. Memory which is marked as such will not be cached in the L2 cache and cached as write-back in the L1 cache. The SCC instruction set architecture (ISA) also contains a new instruction for use with this memory type. The instruction CL1INVMB invalidates all cache lines in the L1 cache that are marked as message-passing buffer data. If the instruction is called before modifications to the L1 cache were written to memory, the data is lost.

The SCC is mounted on a field-programmable gate array (FPGA), which adds a management console reachable via telnet, controllers for solid state disks (SSDs), network access and a Peripheral Component Interconnect Express (PCIe) connection to a host PC. The host PC is used to initialize the hardware and to set up the OS to run on the SCC.

The documentation for the SCC is incomplete and only covers the high-level architecture of the chip. Most of the low-level details and functionality can only be reverse engineered from the source code of Intel's tools for the SCC [2]. In the following sections, I will cover what I could figure out and what I used for the implementation of my driver.

2.2.2 Memory controller memory layout

Specified in the Intel SCC External Architecture Specification (EAS) are default memory mappings for setups with 16, 32 and 64 GB of system memory, which are used by the sccKit, the tools package provided by intel for use with the SCC. The

[2] http://marcbug.scc-dc.com/svn/repository/trunk/

current implementation of Barrelfish also assumes the default memory mapping for 16 GB of main memory (see table A.1).

	Position	Port	Private Memory of Core #
0	x=0, y=0	West	0-5, 12-17
1	x=5, y=0	East	6-11, 18-23
2	x=0, y=2	West	24-29, 36-41
3	x=5, y=2	East	30-35, 42-47

Table 2.3: Locations of the four memory controllers

The default memory mapping is used for every core and each core's private memory is mapped to a different system memory region and memory controller (see table A.2).

2.2.3 Write-combine buffer

To reduce write accesses to MPB memory, the SCC uses a write-combine buffer. As the functionality of this buffer is not covered by the Intel documents, it can lead to unexpected results.

To figure out the exact functionality of the write-combine buffer, I did a number of tests with the following setup:

The host PC booted an image of Barrelfish on the SCC and immediately started checking a predefined cacheline for changes by issuing non-cacheable read requests via the FPGA (see section 2.2.4). This way, the host PC was able to monitor the data in main memory without any intervening caches. As soon as it detected a change, it notified me of the changed data.

The image that was booted on the SCC was configured so that it would not use any memory mapped as MPB. The boot process was configured so that it mapped the frame with the predefined memory location and carried out the steps as in table 2.4. The numbers in the table indicate the order of the steps, while the letters denote branches that were carried out separately with a clean setup each.

Results

The results of these tests led to the conclusion that the write-combine buffer works on a complete cacheline as assumed and is activated if and only if the memory is mapped as MPB memory, independent of the actual memory hardware type. The write-combine buffer also acts as a cache for a single cacheline and is unaffected by the CL1INVMB instruction. The buffer is flushed when a write to a different cacheline mapped as MPB memory occurs or when each byte on the cacheline was written at least once.

Step #	Description	Success?
1	write part of the cacheline	
2.a	write the rest of that cacheline	yes
2.b	write another cacheline on the same page	yes
2.c	write to the stack	no
2.d	write to a global variable	no
2.e	write to a second page mapped as MPB	yes
2.f	do a CL1INVMB and write to a second page mapped as MPB	yes
2.g	do a CL1INVMB and write to a second page not mapped as MPB	no
2.h	do a CL1INVMB, write to a second page not mapped as MPB and write to a second cacheline on the first page	yes
2.i	write to a second page not mapped as MPB, that is on a different memory controller	no
2.j	write to a second page mapped as MPB, that is on a different memory controller	yes
2.k	read the whole cache line into a buffer, write the whole cache line from the buffer	yes
2.l	write part of the cacheline repeatedly	no

Table 2.4: Write-combine buffer functionality test steps

2.2.4 FPGA / System Interface

The Rocky Lake or Copperridge FPGA is connected to the host PC as a PCIe device and provides access to the SCC via the chip's System Interface (SIF). The host PC can send messages through the FPGA directly to routers in the mesh and thus to every memory location on the SCC. While the messages used in the mesh, called flits, are of variable length, the message sent to the SIF are 48 bytes long with a header length of 16 bytes, leaving 32 bytes for payload. They can be transmitted using programmed input/output (PIO) or the builtin direct memory access (DMA) engine.

2.2.4.1 Host-SCC messaging

The FPGA has 2 FIFO queues for input and output. For these two queues, there is one 32-bit status register (trnct4), divided into input and output part. Both parts are 16 bits wide and consist of a 15-bit counter and an error bit. The counter indicates the number of 8-byte words in the queue. The maximum number of bytes in the queue can be read from a 32-bit read-only register.

To send a message, using either PIO or DMA, the input counter has to be checked first to ensure that the queue is not full. When the output counter has a value greater than zero, a message is pending and should be received by polling by the host PC.

SIF messages are sent to the appropriate destination by the FPGA using the mesh on the chip through the SIF after the messages have been converted to flits. Response flits are sent back to the SIF and converted by the FPGA to the 48-byte message format.

Most of the information in this chapter is extracted and reverse-engineered from the sources of Intel's sccKit, the set of tools provided by Intel for use with the SCC, since the host-SCC messaging is not documented in any way.

Message format The message format for all messages sent and received over the SIF is the same. All messages are 48 bytes long, with a payload size of 32 bytes and a header size of 16 bytes. Most messages don't use the full 32 bytes for payload.

All message types in use work with an address alignment of 8 bytes. Not aligned addresses in messages are truncated along the way and changed to the next lower 8-byte alignment border. The only exception from this is the WBI command, which uses a 32-byte address alignment.

Read operations (NCRD command) do not use the payload part of the message at all and their responses (DATACMP command) use only the first 8 bytes of payload and bytes 8-15 for some additional data from the same cacheline. This additional data, however, is not used by Intel's sccKit, thus can not be deemed safe to use.

Write operations (NCWR command) use only the first 8 bytes to transmit data. The NCWR command is the only operation in use that uses the *byteenable* header field to determine which bytes should be written. Write operations do not generate a response.

	Size	Description
	32 bytes	Payload
byteenable	8 bits	For write operations, each bit can be set individually to indicate whether the byte should be written. A pattern of 0*1+0* is expected.
tx ID	8 bits	unused
source ID	8 bits	The ID of the sender of the message. When sending from the host PC, this is to be set to 0.
destination ID	8 bits	The ID of the receiver of the message. When sending to the SCC, this is to be set to 1.
address	34 bits	The address field has to contain the *system address* of the memory location to read or write.
command	12 bits	See table 2.6.
rck ID	8 bits	The rck ID is of the form (y:4 x:4), where (x, y) are the coordinates of the destination tile.
rck sub-ID	3 bits	The sub-ID denotes the destination port of the router (see table 2.2).
reserved	5 bytes	

Table 2.5: Host-SCC message format fields

For a schematic view of the message format, see table A.3

Cacheline write operations (WBI command) use the full 32 bytes payload to write a whole cacheline on the SCC.

For a description of all the header fields, see table 2.5.

For all message types and their description, see table 2.6.

PIO The FPGA provides an easy to use but slow method of transferring messages with PIO. For message input, the 32-bit registers *trnct2* and *trnct3* have to be written in turn, to effectively write 8-byte chunks of messages. Receiving messages is done in a similar way, by reading the registers *trnct0* and *trnct1* in turn.

PIO has no setup time, thus no overhead when a single message is transmitted, but it is not fast enough to transmit large amounts of data.

DMA engine The SCC board has a built-in Xilinx DMA engine that can be used to read and write messages on the FPGA.

A DMA transfer is initiated by first resetting the DMA engine, configuring it by writing configuration values to read or write configuration registers and starting it by writing the correct bits to the control register.

Name	Command #	Description
DATACMP	0x04f	Response with data
NCDATACMP	0x04e	SysIF specific: NCRD Response with data
MEMCMP	0x049	Only Ack no data
ABORT_MESH	0x044	Abort
RDO	0x008	Burst line fill (Read for ownership)
WCWR	0x122	Memory write (1,2,4 8 bytes)
WBI	0x02C	Cacheline write / Write back invalidate
NCRD	0x007	Non-cacheable read (1,2,4,8 bytes)
NCWR	0x022	Non-cacheable write (1,2,4,8 bytes)
NCIORD	0x006	I/O Read (sent by SCC cores)
NCIOWR	0x023	I/O Write (sent by SCC cores)

Table 2.6: Host-SCC message format fields

Whether the transfer is completed can be checked in the control register, or by waiting for an interrupt from the PCIe device. In the current implementation of the *sif* driver (see section 4.2.3), the completion of the DMA transfer is checked with the control register, doing a busy waiting until the transfer is finished. The driver does not use interrupts at the moment, because the interrupts do not arrive reliably (see paragraph DMA Interrupts).

For a complete set of steps for a DMA transfer, see tables 2.7, 2.8.

The information about the DMA engine was extracted from Intel's sccKit and from *Xilinx XAPP1052 Bus Master DMA Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions*, which seems to discuss the DMA engine used on the SCC FPGA or a device that is very similar.

DMA Interrupts The Xilinx DMA engine on the Rocky Lake board can be configured to send an interrupt when a DMA transfer is finished. Intel's sccKit tools use it in a very simple way to avoid busy waiting for the end of the transfer:

Before the DMA transfer is started, a variable is set to *false* and when the PCIe device is initialized, an interrupt handler is registered that sets the variable to *true*. After the initiation of the transfer, a loop is entered that is exited as soon as the value in the variable is *true*.

I implemented the same pattern for the *sif* driver, but apparently some part in the system could not keep up with the number of interrupts and only the first interrupt was received by the driver. However, seemingly random added instructions to make the driver slower, like printf or loops, could resolve the issue.

A small test showed that the kernel did get all the interrupts and attempted to send them to the driver: Instead of waiting for the value of the variable to become *true*, I modified the driver to do busy waiting on the control register until the transfer was complete and not attempt to handle any interrupts. With this modification,

Step	Operation	Register Operation	Value
1	Assert initiator reset	Write DCR1	0x00000001
2	De-assert initiator reset	Write DCR1	0x00000000
3	Write hardware memory address	Write RDMATLPA	address
4	Write TLP size	Write RDMATLPS	TLP size
5	Write TLP count	Write RDMATLPC	TLP count
6	Write DMA start	Write DCR2	0x00000001
7	Wait for interrupt (Check for completion bit)	(Read DCR2)	0x00000100

Table 2.7: DMA transfer read steps

Step	Operation	Register Operation	Value
1	Assert initiator reset	Write DCR1	0x00000001
2	De-assert initiator reset	Write DCR1	0x00000000
3	Write hardware memory address	Write WDMATLPA	address
4	Write TLP size	Write WDMATLPS	TLP size
5	Write TLP count	Write WDMATLPC	TLP count
6	Write DMA start	Write DCR2	0x00010000
7	Wait for interrupt (Check for completion bit)	(Read DCR2)	0x01000000

Table 2.8: DMA transfer write steps

the driver did receive all the interrupts, but after all the DMA transfers had been completed.

As I could not get it to run without adding extra instructions in the DMA code, I changed the driver to use busy waiting on the control register by default, but it can be configured to use interrupts instead.

2.2.4.2 Benchmarks

I did some benchmarks on the SCC to test the speed of the PCIe connection to SCC memory. I tested the time needed to read chunks of memory from SCC memory from all memory controllers to see how transmission times increase with bigger chunk sizes and to see if there are any differences between the connections to the memory controllers. I also tested the time needed to read from MPB memory of different cores to see if there are any differences between memory close to the SIF and memory far away from the SIF.

All the benchmarks were done using DMA, PIO was not used.

Memory Controllers The benchmark of the memory controllers consisted of bulk read procedures of 4, 8, 16, ... 1024, 2048 bytes. Each test was completed with a 99.9% confidence interval for the mean of less than $\pm 1\%$.

From the benchmarks I can conclude that the read operations over the PCIe bus have a part that requires a constant amount of time and a part that is linear to the size of the chunk read. From the data I can see that the constant part of the transmission times takes about 50000 host clock cycles. For small chunk sizes (< 64 bytes) the constant part of the transmission time dominates the overall transmission time and transmission times do not vary much (see figure 2.2).

From the benchmarks I can also conclude that reading from different memory controllers does not change the time needed to read data.

MPB memory The benchmark of the MPB memory consists of read operations of 4 and 4096 bytes from MPB memory of the cores 0, 4 and 46. All tests were completed with a 99.9% confidence interval for the mean of less than $\pm 1\%$.

Core 0 is the core that is booted first by default. Core 4 is the core that is closest to the SIF and core 46 is one of the cores that is the furthest from the SIF.

The benchmarks show that the time for reading from MPB memory on different cores does not differ or the difference is too small to be of any significance (see figure 2.3).

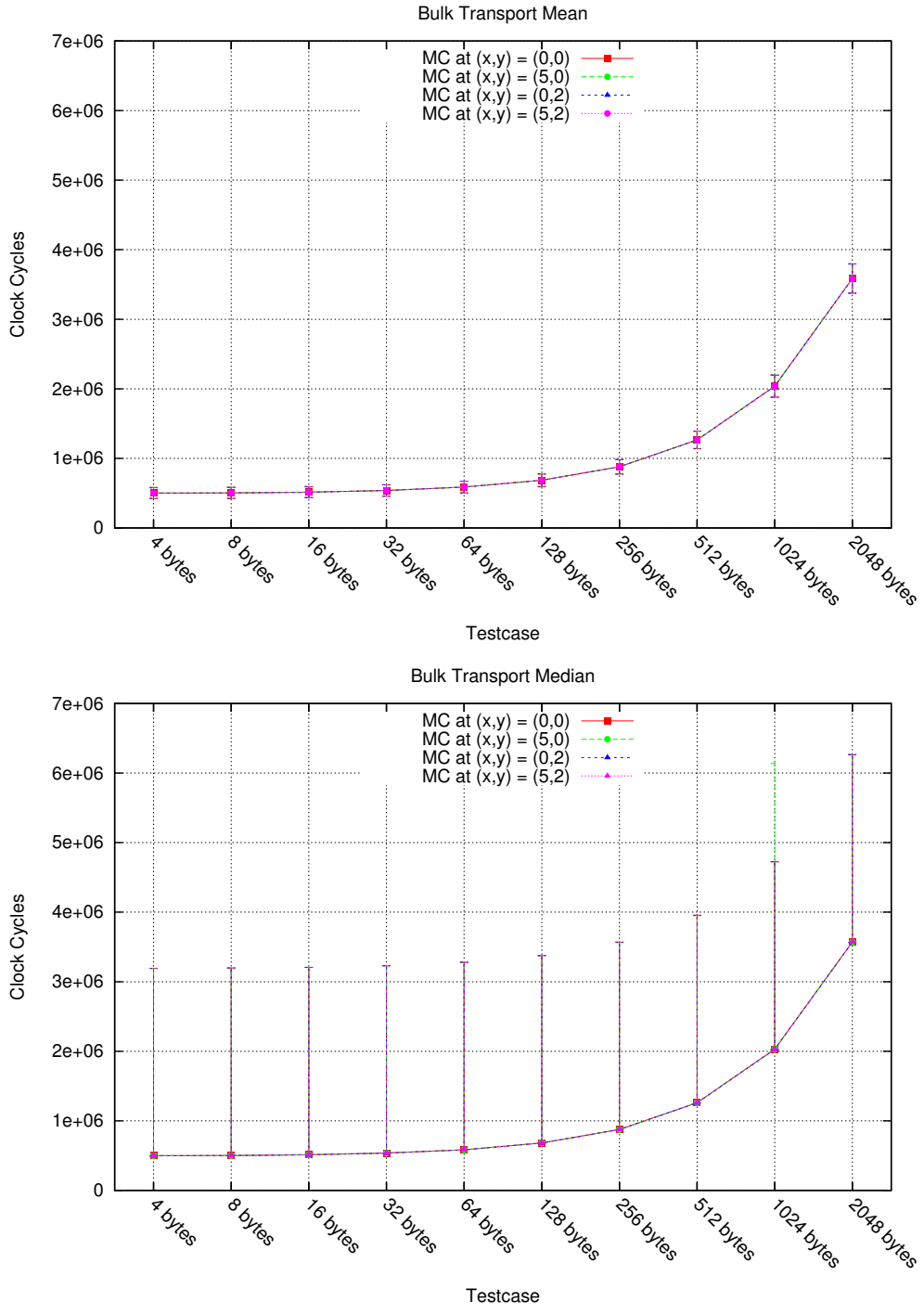

Figure 2.2: SCC main memory bulk transport read times

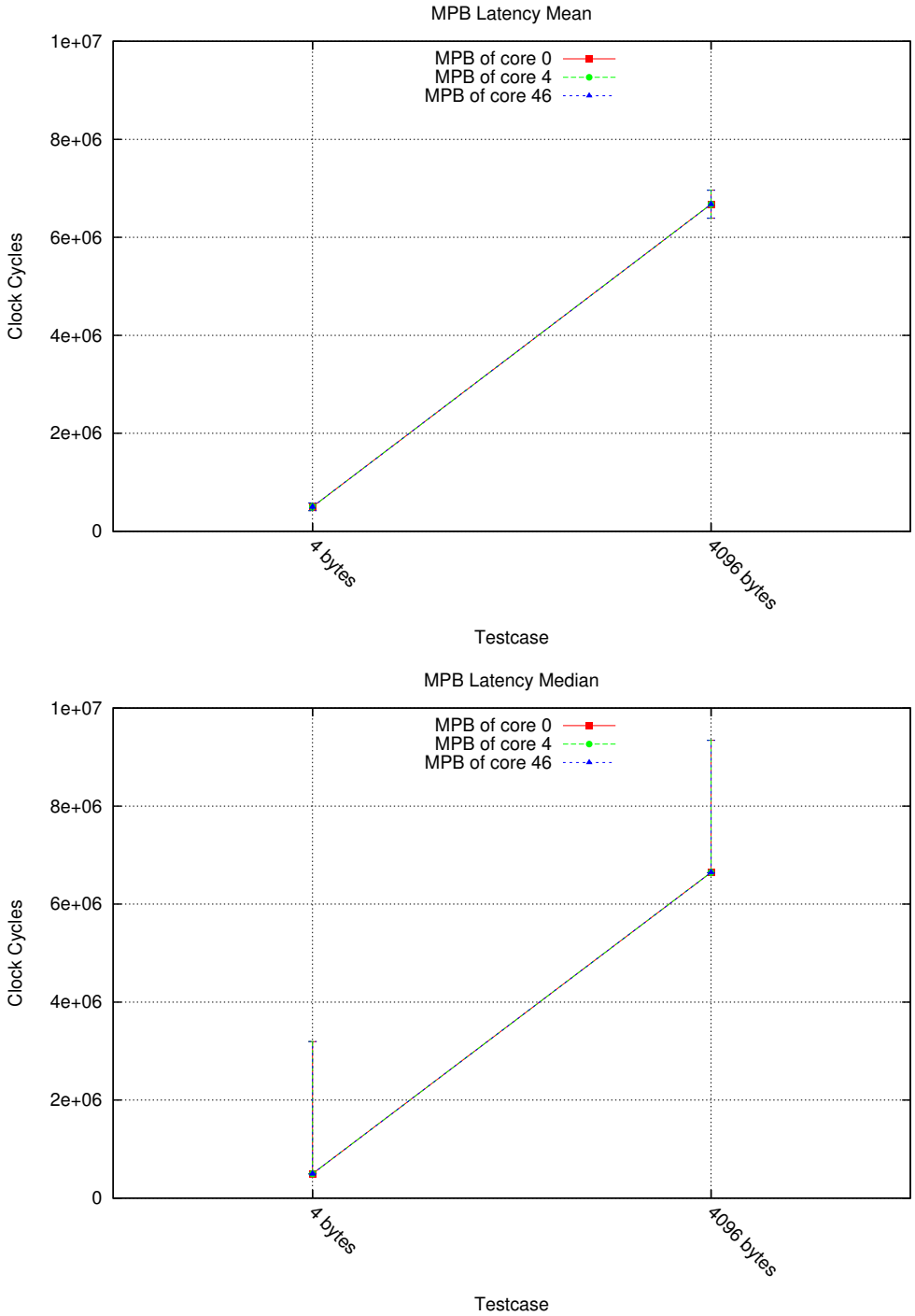

Figure 2.3: SCC MPB memory read times

3 Heterogeneous Support

3.1 Overview

To create an OS that is able to span different architectures, a few things are required:

- The OS must be able to start cores of different architecture.
- There must be a communication protocol that is independent of architecture or performs the required conversions.
- The system must be able to start applications on cores of different architecture.

3.1.1 Boot Process

In Barrelfish, the booting of all cores that should run a cpu driver happens during the boot process of the bootstrap core. Cores can at the moment not be added to the system at runtime and so the boot process is important in creating a heterogeneous system. For this reason, I will first describe the startup process of Barrelfish in a homogeneous system.

1. On core 0 the bootloader starts and loads all modules that are necessary for the system to start over the network into memory. After that, the bootloader starts the kernel with command line arguments that indicate to the kernel that it runs on the bootstrap core. Part of the command line arguments is the memory address of a data structure containing records of all modules that were loaded by the bootloader.
2. The kernel initializes the cpu and relocates itself to a different virtual memory region.
3. Following that, the kernel starts *init*. Init starts the *monitor* and *mem_serv* and helps them connect to each other.
4. When the monitor on the bootstrap core starts, it first connects to mem_serv and initializes its rpc service. It then spawns the nameservice, *chips*, followed by all the modules that have the "boot" command-line argument specified in the bootloader's config file.

5. One of the "boot"-modules is *spawnd*. Spawnd provides the service to start processes and is therefore essential for the system if programs should be spawned after the boot process is over. On the bootstrap core, spawnd also starts the boot process on other cores. For each core it decides to boot (specified either by command-line arguments or detected automatically), spawnd sends a boot request to its local monitor and waits until a spawnd process has started on the new core and registered itself with the nameservice.

6. When the bootstrap monitor receives a boot request, it copies the Executable and Linkable Format (ELF) file of the kernel to a free memory region and initializes the binary. It allocates a new frame for a UMP connection and passes its memory address to the new core as argument.

On the cores booted by spawnd the bootprocess is slightly different:

1. After the kernel is booted, it does not spawn init, but directly the monitor.

2. Before the monitor can provide its services, it needs a connection to the bootstrap monitor. It does that by creating a UMP connection with a frame whose address it was given as a command-line argument. The frame is also known to the bootstrap monitor, which creates the other endpoint of the UMP connection.

3. Once the connection is established, the monitor on the app core can continue its initialization by connecting to mem_serv first.

4. To start its own local spawnd, the monitor requests the image of spawnd from the bootstrap monitor.

3.1.2 Communication

When a connection between two cores is made in Barrelfish, the Flounder backend used is UMP. Because connections spanning multiple cores do not have the possibility to pass message arguments in registers, UMP uses a shared frame to transmit the messages.

When a process sends a message over a UMP connection, the process calls the function for the specific message. This function copies the message arguments to a connection-specific struct and calls the marshalling function. The marshalling function writes message packages into a ringbuffer that is located in the shared frame. It takes each argument and writes it in an array of pointer-sized words or splits and combines arguments to make best use of the available space. This array fills the complete space available for payload in a message package. On the other side of the connection, the unmarshalling function takes these words and builds the message arguments out of them.

Capabilities are handled separately. The reference to a capability is sent to the monitor so that the monitor can send the actual capability to the monitor of the target processor and the other endpoint can receive the reference to its local capability from the monitor.

3.1.3 Application Start

To start a new process, regardless of the target core, the starter process uses the *libbarrelfish*. This library provides convenience functions to communicate with *spawnd*, the spawn daemon. By using these functions, the starter process connects to spawnd on the target core and sends a message to load and run a certain binary. Part of this command is the *env* array, which contains all the environment variables set in the starter process, like *PATH*. The PATH variable is also used by the library to search for the binary of the given name, so that it can send spawnd the path to it.

Spawnd then connects to *ramfs* to look for a binary that matches the given path, loads it using either a shared frame for bulk transport or Flounder messages, and runs the process on its core.

3.2 Implementation

3.2.1 Cross-Architecture Boot

In the architecture of Barrelfish, the points where the implementation depends on the architecture are few. They are mainly the locations where a binary is selected, transmitted or installed or the actual hardware is used and configured. This greatly helped in developing a version of Barrelfish that is able to span x86_64 and x86_32 cores.

To get a heterogeneous OS, I had to change the interface of the monitor, so that spawnd can not only select which cores to boot, but also tell the monitor of which architecture they are. Similarly, the monitor on the new core needs to be able to tell the bootstrap monitor which binary of spawnd it requires, which I could also accomplish with a change of the monitor's interface.

3.2.2 Cross-Architecture Communication Protocol

The marshalling process works very well if both endpoints are of the same architecture, but if they are different, this ceases working, because the words in the payload array might not have the same size due to size differences of pointers and thus the arguments may be grouped differently. Also, arguments may themselves be of different size, like Barrelfish's error values, which are pointer-sized.

For this reason, I decided that when communicating between cores of different architecture, UMP should use the same word size for grouping message arguments and the message arguments should be converted to architecture-independent types.

As I was working with x86_32 and x86_64, the obvious choices for payload array word sizes were 64 bit and 32 bit. I benchmarked versions of UMP with 32 bit word sizes and 64 bit word sizes and compared their speed to the original implementation. Additionally I created a version of UMP that splits, combines and groups words as if it would write 32 bit words, but combines them if necessary to write pointer-sized words, i.e. on x86_64 it combines two 32 bit words and writes a 64 bit word to memory, because I wanted to be able to use the optimized 64 bit capabilities of the x86_64 architecture.

As can be seen in figures 3.1 and 3.2, the implementation that uses pointer-sized words to write the marshalled arguments into memory, but groups them as 32 bit words in any case, is the fastest and so chosen for my heterogeneous implementation.

The greatly increased transmission times of the 64 bit version on x86_32 for the *payload32_16* testcase, I can't explain. The message uses two message packages and should, extrapolated from the *payload32_8* testcase, not be significantly slower than the original implementation.

Also, the increased transmission times of the 64 bit version on x86_32 for the *empty* testcase is unexpected, because apart from the word sizes and the resulting change in the marshalling functions, nothing was changed and this testcase should not do any marshalling.

The increased transmission times of the 64 bit version on x86_32 for the *buffer* testcase can be explained by the functioning of the buffer marshalling function. This function reads individual bytes from the buffer, ORs them together in a register and writes this word into the payload array. With the 32 bit sized registers available on x86_32, these operations might take more time when a 64 bit word is manipulated. However, I did not test if this is the true source of the increased transmission time.

To be able to correctly transmit architecture-dependent types, every message argument is converted to its architecture- independent counterpart and saved in a struct that is very similar to the usual struct UMP uses for storing the message arguments, but stores architecture-independent types. The marshalling that is called is not the regular marshalling function, but one that is aware of the possibly different sizes of the new types and marshalls the message arguments accordingly. On the receiver side, I did the same, so a different unmarshalling function is used and the arguments are converted to the architecture-specific types before the registered callbacks are called.

3.2.3 Cross-Architecture Application Start

When a new process should be started on a core of different architecture, spawnd on the target core has to load the binary of the correct architecture. If the starter process calls the libbarrelfish, the library uses the PATH variable of the starter process to look for the binary. When the architecture of the source ant target cores is not the same, libbarrelfish looks in directories with binaries of the wrong architecture type. It may fina a binary with the correct name and instruct spawnd to run it. This of course fails and the starter process receives an error message.

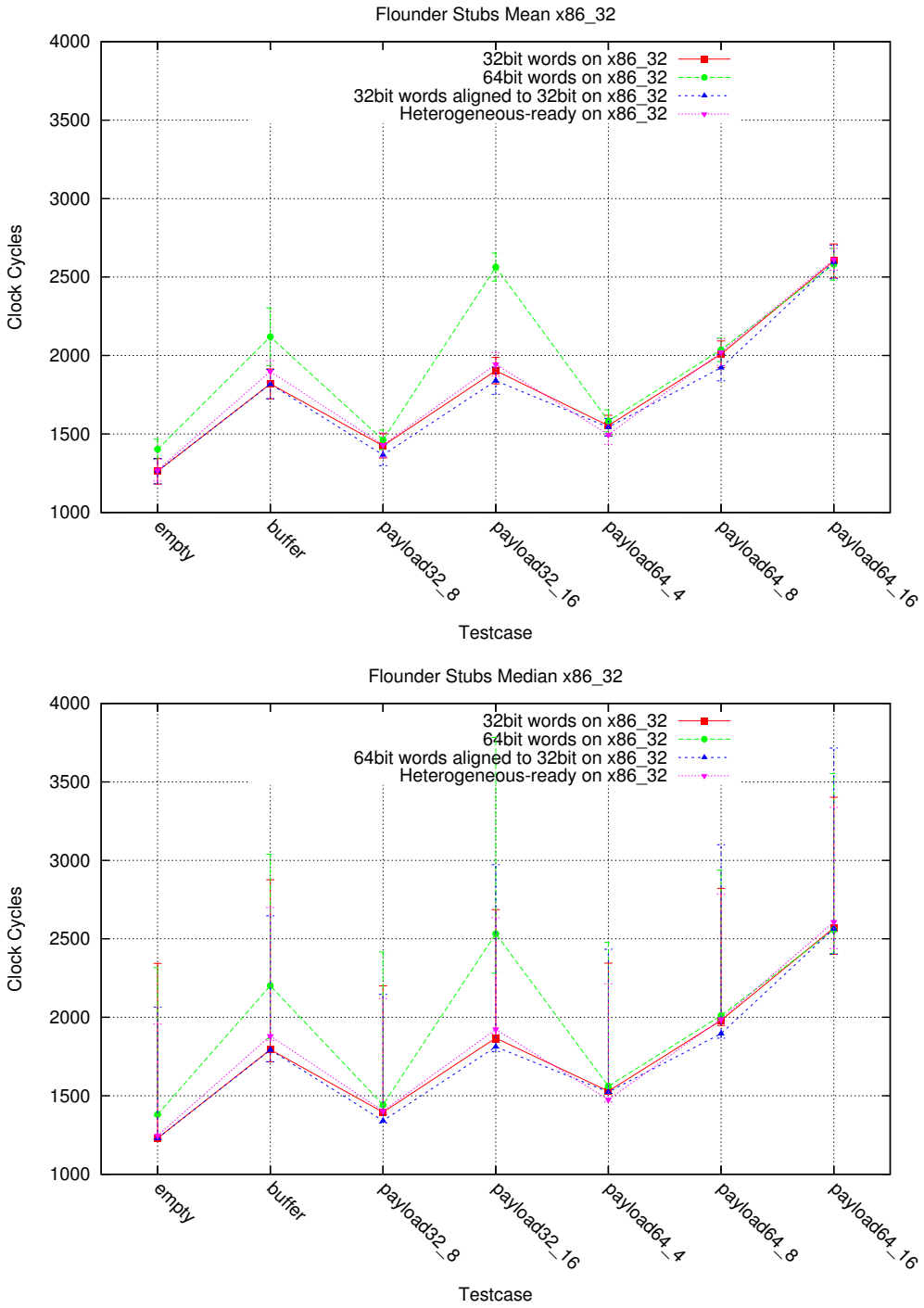

Figure 3.1: Transmission times for different payload array word sizes on x86_32

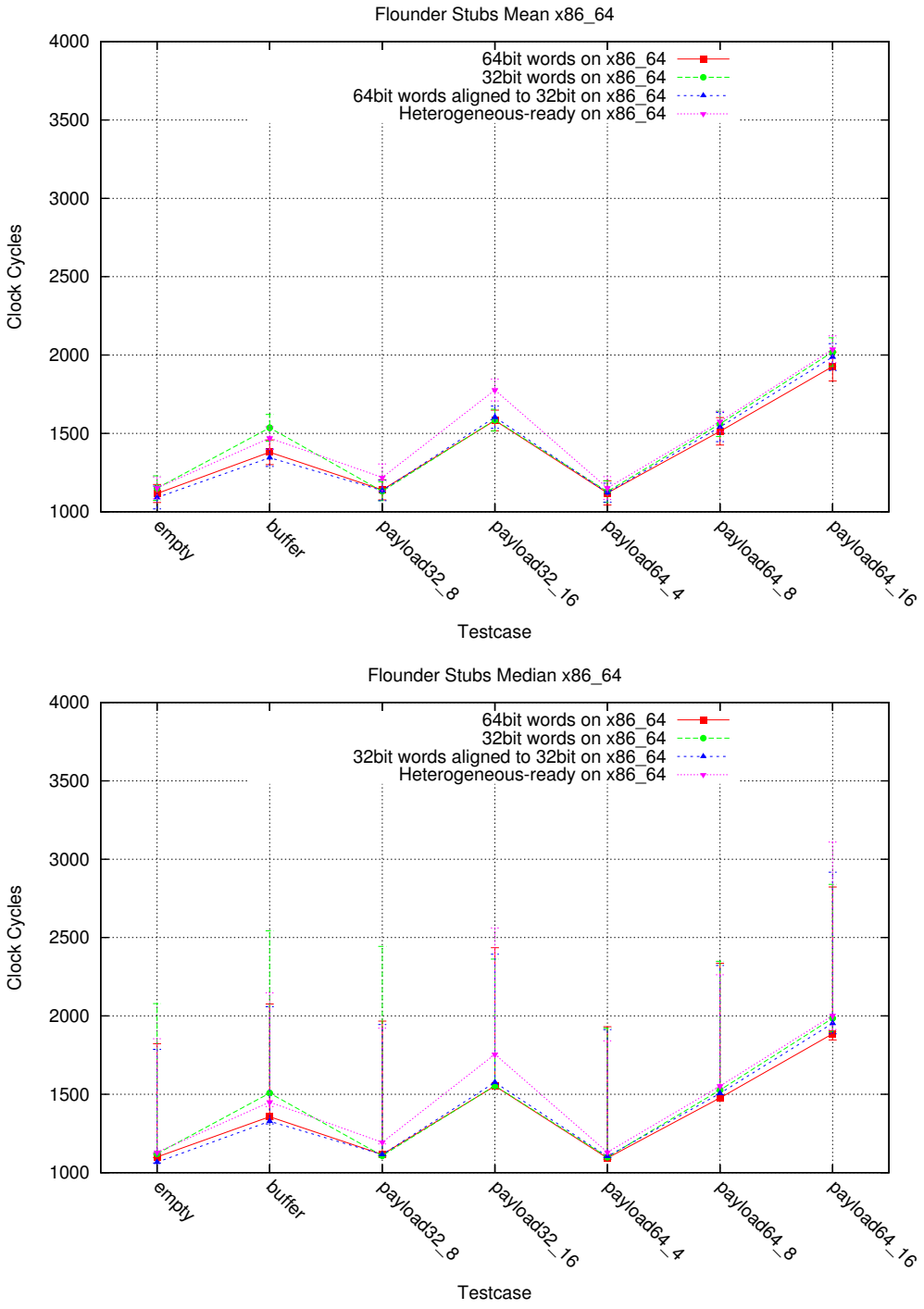

Figure 3.2: Transmission times for different payload array word sizes on x86_64

A workaround for this is to supply libbarrelfish with the full path to the binary instead of only its name. This way, it ignores the PATH variable of the starter process and loads the correct binary.

This is only a temporary solution and if heterogeneous systems should become a normal part of Barrelfish, a more robust and flexible solution is needed.

A possible solution would be to add paths to the PATH variable that contain binaries for other architectures. The libbarrelfish would then be able to find a binary that matches the name as well as the architecture. The downside of this would be that the starter process needs to know the architecture of the target core and be able to search for the binary, but this may not be possible if the binaries are stored on a file system only accessible by the target core.

To avoid this problem, it may be possible to move the searching from the libbarrelfish to spawnd and let the target core decide whether a binary matches the core's architecture.

Another possibility would be to standardize binary paths. At the moment all binaries for a certain architecture are in the directory `/<arch>/sbin/`. A naming scheme that is more flexible could be created so that searching for a binary would become faster, because less directories have to be checked and less binaries have to be checked for matching architectures.

3.3 Evaluation

The heterogeneous implementation of UMP is slower than a pure UMP implementation, as can be seen in figures 3.1 and 3.2. This is expected, as the heterogeneous-ready implementation has an overhead with converting and correctly aligning message arguments to an architecture-independent format. The overhead in terms of transmission speed is about 1%-4% for messages transmitted between x86_32 cores and about 5%- 12% for messages transmitted between x86_64.

The transmission times of heterogeneous connections between x86_64 and x86_32 cores are higher than the times of homogeneous connections between two x86_64 cores but lower than the times measured for homogeneous connections between two x86_32 cores. It is expected that the heterogeneous connections are not as fast as connections between x86_64 cores, as those connections do not perform conversion, but that is not the main contributor to the great difference between those two connections. As half of the work for the heterogeneous connection is done on the x86_32 core, which does not have the same possibilities in terms of instructions and number of registers as the x86_64 core and which therefore is slower, the heterogeneous connection speed is between the two homogeneous connection speeds (see figure 3.3).

The speed could probably still be increased slightly with an optimized marshalling process that writes a 64 bit chunk of a message argument as a 64 bit word directly instead of splitting it into two 32 bit words and combining them again for writing into the payload array.

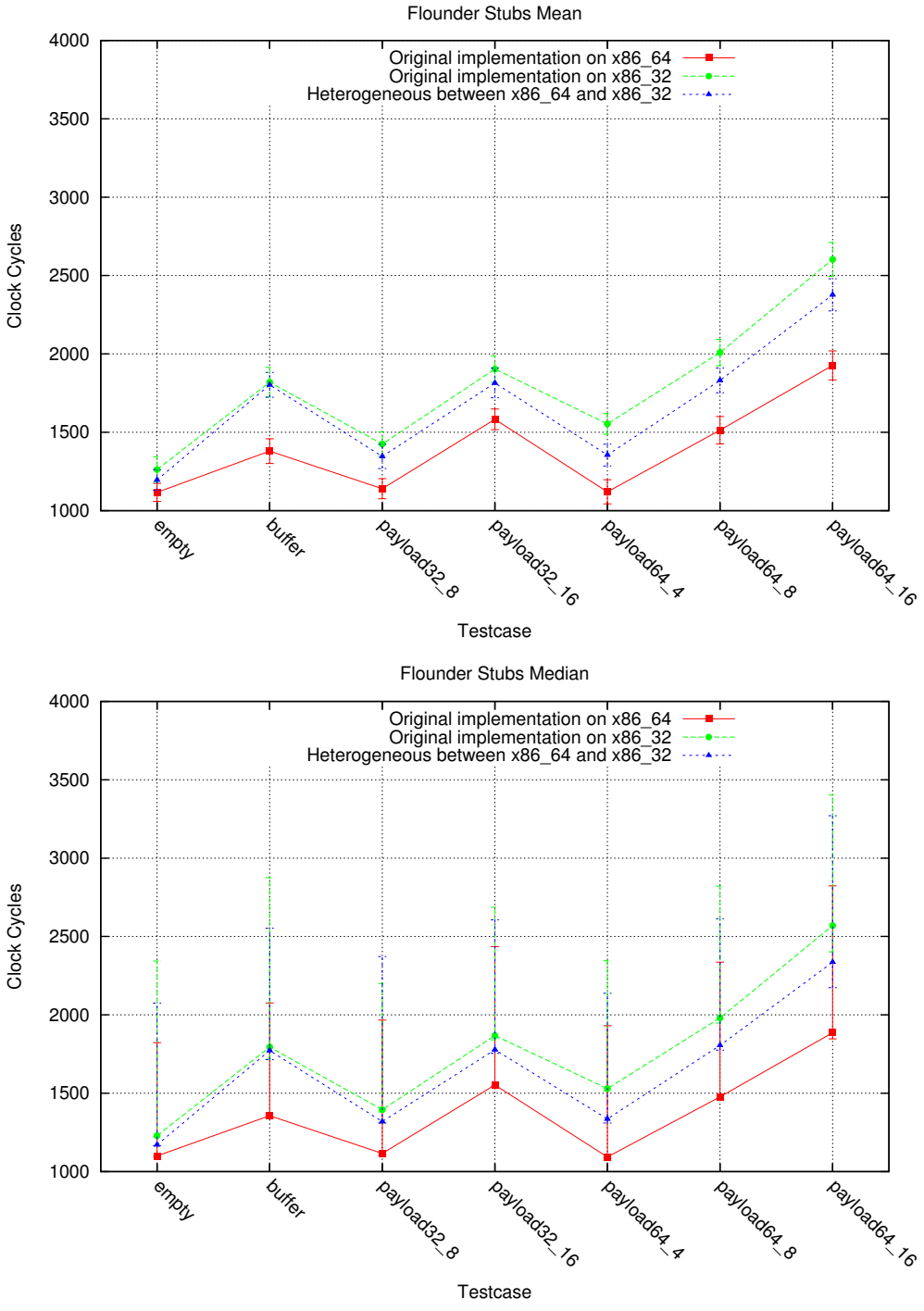

Figure 3.3: Transmission times of a heterogeneous compared to homogeneous connections

3.4 Future Work

3.4.1 Endianness

The heterogeneous version of UMP is not completely ready for connections between different endiands yet. It is at the moment optimized and tested for little endian and may also need some adaptions for connections between big endian architectures only.

In any case, when a new architecture is to be supported by heterogeneous UMP, regardless of the architecture's endianness, new conversion functions have to be added so that variables on the new architecture can be converted to architecture-independent types.

3.4.2 Capabilities

Capabilities are at the moment not converted in any way when transmitted between cores of different architecture. The conversion of capabilities would have to be done in the monitor as that is the only place in the connection where the actual capability is available rather than a reference to it. Between x86_64 and x86_32, capabilities do not require any conversion, because the datastructures have the same types on both architectures and also have the same representation in memory. This is also the reason why I did not concentrate on that topic any further, but decided to begin working on a connection to the SCC.

However, even if capabilities can be transmitted without conversion, they are not necessarily valid on another core. Frame capabilities from a x86_64 core for example are not valid on an x86_32 core if the frame contains memory locations above 4GB. This would probably have to be resolved with the help of the *skb*.

4 SIFMP - SIF message passing

The Intel SCC is a research chip developed to give researchers the possibility to work with an architecture that might be the future of all processor chips. Specifically the unavailability of cache coherence and the non-shared memory impose restrictions on OSs and applications that run on this kind of processor. (See chapter 2.2 for more information on the SCC)

As Barrelfish already treats each core in a system as separate entities rather than a multi-threaded environment, Barrelfish should be perfectly suited to run on such a kind of chip. A version of Barrelfish that runs as one OS, spanning the whole chip already exists, so it seems natural to try and connect the SCC and the host PC to create an OS that spans both.

In this chapter I will describe the functioning of a communication protocol that enables processes to talk to each other even if they are on opposite sides of the PCIe bus that connects the SCC to the host PC.

4.1 Overview

For the communication protocol, there were a few requirements:

1. The SCC is attached to the host as a PCIe device, so it needs a device driver to access and configure it. The communication necessarily involves the driver.

2. The communication protocol should be integrated into the existing IDC framework, i.e. an extension to Flounder.

3. As the communication between the host and SCC cores is part of the boot process, a communication channel has to be opened without an external communication channel or manual configuration.

4.2 Implementation

The protocol used to connect processes on the host PC to processes on the SCC is called *SIFMP*, for *SIF message passing*. *sif* is the name of the driver for the SCC FPGA and stands for *System Interface*, the port on the SCC that connects it to peripheral devices.

4.2.1 Possible Communication Schemes

For communication protocol across a PCIe bus, a number of different approaches seemed possible, which I will describe in this section.

4.2.1.1 Double Proxy

The *Double Proxy* version of SIFMP uses a driver on both sides of the PCIe bus.

Processes that want to create a connection to another process on the other side of the bus allocate a channel from the driver and register the connection to the driver as the notification channel.

Part of the allocated channel is a frame the allocating process gets from the driver, that is used as ringbuffer as with UMP. When the process sends a message, it writes the message in the ringbuffer and sends a notification to the driver, which copies the message to a similar frame on the other side of the PCIe bus and forwards the notification to the receiver of the message.

The process allocating a channel receives a channel identification number, which is sent to the other endpoint of the SIFMP connection so the other process can request its local frame for the ringbuffer and register a connection to the driver as notification channel.

This version is very symmetric from the processes' point of view, but uses two frames and two non-SIFMP connections for every SIFMP connection.

4.2.1.2 Double Proxy, Notificationless

This version of a communication scheme is very similar to the *Double Proxy* version, but does not use notifications.

When a new SIFMP channel is created, it is normally registered on the default waitset. A channel registered on a waitset can either be idle, polled or pending.

If a channel is in idle state, there are no pending messages on that channel and the channel will not be checked when a message handler loop for that waitset is entered.

In pending state, a channel does have pending messages, which will be delivered as soon as the waitset is processed.

Channels marked as polled may have channels pending, but they are not marked as pending, because the channel state is not updated when a message arrives. When checking the waitset for pending messages, each channel marked as polled has to be checked individually.

In the *Double Proxy* version, SIFMP channels are registered on a waitset as idle, because they are triggered by notifications sent by the *sif* driver. A version without the notifications needs to register the channels as polled. On the host PC this simply results in a protocol that is more similar to UMP than to UMP_IPI. On the SCC however, UMP does not work, because there is no cache coherence for shared memory, so the ringbuffer frames either have to be mapped as uncached or a method to cause a cache flush has to be added, which is very similar to a notification system.

Also, there is an additional overhead for at least the host part of the *sif* driver, as it needs to check all channels manually for new messages. In the *Double Proxy* version, the driver has to poll at most one channel per process that uses SIFMP connections. For system services that may have a big number of SIFMP connections such as *chips*, this difference could become significant.

4.2.1.3 Double Proxy, IPI Notifications

This version of SIFMP is very similar to the *Double Proxy* version, but instead of using Flounder channels and messages to notify applications of pending messages on SIFMP channels, it uses IPIs to notify channels of pending messages, much like UMP_IPI does.

This results in a protocol that is largely the same as the *Double Proxy* version, but with more restrictions on the hardware. SIFMP may be faster when using IPIs, so this is a tradeoff between fewer requirements and speed.

It would probably be relatively easy to add IPIs as an optional notification mechanism to the *Double Proxy* version.

4.2.1.4 Single Proxy

While the three previously discussed versions of SIFMP use a driver on both sides of the PCIe bus, this possible implementation uses only one driver part on the host.

In this version, the host driver does not only trigger one channel on the SCC with a message in the MPB and an IPI, which it does in the *Double Proxy* version, but by using that mechanism for every channel individually. The channel number for the MPB message and the core number to trigger the kernel on the right core have to be passed to the host endpoint as part of the binding process via the monitor.

The host part of the driver is still responsible for copying new messages and updating the ringbuffers, but updates to ringbuffers, i.e. new messages can only be detected when the driver checks every one of them individually, which results in a huge overhead in the form of a lot more read accesses over the PCIe bus.

4.2.1.5 Encapsulation

A completely different approach is to use the Flounder channel of an application to the *sif* driver as a transport medium and encapsulate and encode SIFMP messages in other messages, e.g. LMP or UMP messages.

When sending a message, the SIFMP implementation encodes the message and sends it to the driver, along with a SIFMP channel identifier. The driver then sends the encoded message to the other part of the driver, which forwards it to the application. The SIFMP implementation at the destination decodes the message and delivers it to the application.

This idea is independent of the the notification mechanism and the number of proxies used.

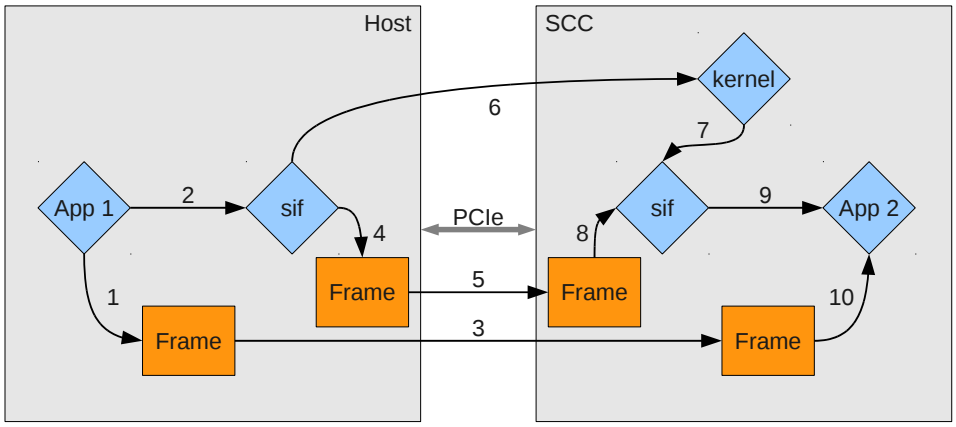

Figure 4.1: Sending of a message from host to SCC

The encapsulation of messages into SIFMP messages requires the use of a message header inside the payload to identify the channel and the arguments, which reduces the available payload space for actual arguments. But such an encapsulation scheme uses much less space than the mechanism described in the previous versions, as each application needs at most one frame for SIFMP messaging, namely for the connection to the driver.

(This version has some similarities with a routing scheme that I will describe in section 4.4.5.)

4.2.2 SIFMP Communication

Of the versions described previously, the *Double Proxy* implementation seemed to be the one that is most practical for a first implementation of SIFMP. For a communication protocol that crosses a PCIe bus and possibly has high round-trip times, notificationless communication and the accompanying busy waiting did not seem appropriate. The version with IPIs seemed too restrictive for this protocol. The *Single Proxy* version would have been much more complicated, because it is not as symmetric as the *Double Proxy* version, which unnecessarily complicates the library for SIFMP messaging. The *Encapsulation* strategy would have used up parts of the payload for additional headers, which I thought undesirable, especially when large amounts of data are transmitted.

With the *Double Proxy* version the SIFMP protocol is very similar to UMP_IPI. It uses frames as ring buffers and uses a separate channel to notify the receiver of a pending message. The separate channel is an IDC connection to the *sif* driver on the host and SCC and a connection between the two driver parts.

Sending a SIFMP from host to SCC involves a number of steps (see figure 4.1):

1. The application on the host writes a message into a ringbuffer, a frame shared

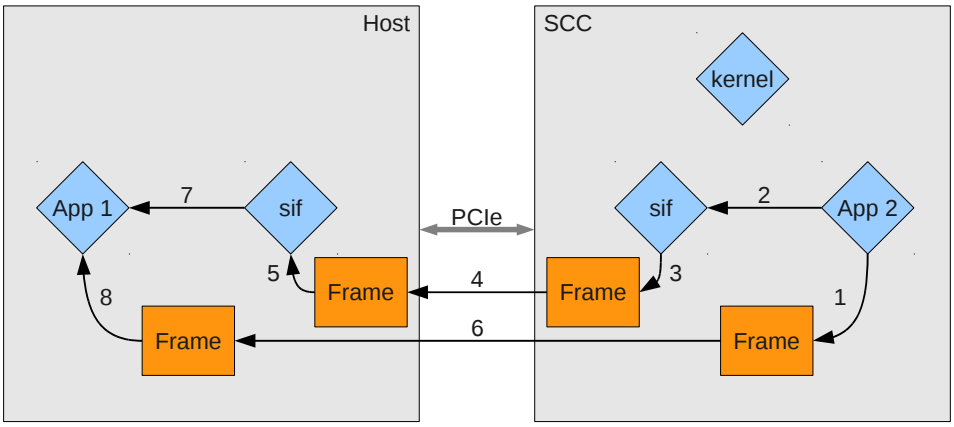

Figure 4.2: Sending of a message from SCC to host

with the host part of the *sif* driver.

2. The application on the host sends a notification to the *sif* driver, using any protocol available between the two processes.

3. The host part of the *sif* driver receives the notification and updates the ringbuffer belonging to the same channel on the SCC with the new messages.

4. The host part of the driver writes a notification into a ringbuffer, a frame on the host.

5. The host part of the driver updates a ringbuffer in a frame on the SCC, known to the SCC part of the driver, with the notification.

6. The host part of the driver writes a notification into the MPB of core 0 on the SCC, protected by the test-and-set lock of that core and triggers an IPI on core 0.

7. The kernel on core 0 sends a LMP message to *sif* to notify it of a new message on its channel to the host and to trigger the receive function of that channel.

8. The SCC part of the driver receives the notification from the host part ...

9. ... and sends a notification to the application on the SCC.

10. The application checks the ringbuffer and receives any pending messages on the channel.

Sending of a SIFMP message from SCC to host is different internally, but works exactly the same from an application's perspective (see figure 4.2):

1. The application on the SCC writes a message into a ringbuffer, a frame shared with the SCC part of the driver.
2. The application on the SCC sends a notification to the driver, using any protocol available.
3. The SCC part of the driver receives the notification from the application and writes a notification into a ring buffer, a frame in SCC memory that is known to the host part of the driver.
4. The host part of the *sif* driver periodically checks for new notifications in the ringbuffer from the SCC part and copies the notification to a ringbuffer in a frame on the host.
5. The host part of the driver receives the notification.
6. The driver updates the ringbuffer in the frame on the host belonging to the channel with the new message.
7. The host part of the driver sends a notification to the application on the host.
8. The application on the host receives the message from the application on the SCC.

4.2.3 SIF

SIFMP uses a two part driver to set up communication channels, to transmit the messages and to notify channel endpoints of new messages. The driver, called *sif*, consists of a host part, running on one of the host PC's cores and a SCC part, running on core 0 (tile 0,0) on the SCC. The host part is responsible for setting up and booting the SCC, transmitting messages, sending notifications to endpoints on the host and receiving notifications for endpoints on the SCC. The SCC part is mainly for sending notifications on its side of the PCIe bus and forwarding notifications for the host.

The two parts are connected by a manually set up SIFMP channel, which they use to coordinate and to forward notifications.

sif implements two interfaces, *sif* and *sif_mgmt*. *sif* is the interface that most applications use. It provides the following services:

create_and_register_chan: This call creates a new SIFMP channel and registers the caller as the endpoint for future notifications on this channel. In the reply, the caller receives channel identification numbers for both endpoints and a frame that is to be used for the ringbuffer.

get_and_register_chan: With this call an application can get the frame for the ringbuffer of a SIFMP channel that was already created by the other endpoint of the channel. The caller is also registered as the endpoint for notifications on this channel.

register_chan: By calling register_chan, an application can register itself as the endpoint of a channel of which it has already received the identifier and the ringbuffer frame capability, like the bootstrap monitor on the host and the monitor on core 0 on the SCC for the bootstrap channel.

notify_chan: This call is used by the channel endpoints to notify the driver of a new message and also by the driver to notify the endpoint.

sif_mgmt is the interface that is used by the monitors to boot the SCC and set up the connection between bootstrap monitor and the monitor on core 0 on the SCC. It provides the following calls:

map_default_lut: This function call causes the driver to set the LUTs entries to the default mapping (see appendix A). This call is only available on the host.

boot_scc: By calling this function, the bootstrap monitor can issue a complete reboot of the SCC. All the cores' configurations are reset and the cores halted. A new image is then copied to the SCC and core 0 started. This call is only available on the host.

get_intermon_chan: With get_intermon_chan, the bootstrap monitor and the monitor on core 0 on the SCC get the channel between them. This channel is created as part of the driver's initialization procedure.

dump_videomem: This call dumps the contents of core 0's log memory region into a file. It is mainly used for debugging purposes.

scc_bind_notification: This call is used to configure the driver and register an endpoint for the IPI that the host part of the driver sends to notify the SCC part. This call is only available on the SCC.

4.2.3.1 Host Part

The host part of the driver serves as the service endpoint for applications on the host. It implements the *sif* and *sif_mgmt* interfaces.

The host part is mainly responsible for allocating frames to be used as ringbuffers in main memory and to copy messages to and from SCC memory. Also during the boot process it is responsible for configuring the hardware and copying the binary image into SCC memory.

To check for new messages on the channel from the other part of the driver, it uses a separate thread in which the ringbuffer is monitored for new messages. If a new message was detected, the channel is triggered and the main thread can receive and handle the message.

4.2.3.2 SCC Part

The SCC part of the driver is mainly responsible for allocating frames to be used as ringbuffers in SCC memory. It allocates the frames from the shared memory region on memory controller 3.

Whenever a UMP_IPI connection is created on the SCC, the originator of the connection allocates a frame of its core's shared memory. The four times 16 MB of shared memory available with the standard mapping is divided into chunks of 1 MB size. Each core can allocate frames in its 1 MB share of shared memory. With 48 cores, this leaves 16 MB of shared memory unused, which is now used by the *sif* driver for connections to the host.

4.2.4 Bootstrapping

Since channels between cores, their monitors and spawnd are part of the boot process, and channels between host cores and SCC cores require the *sif* driver, the driver is part of the boot process and has to be tightly integrated.

4.2.4.1 SIF

On the host, *sif* is started by the monitor, but is otherwise a normal process, because it is started only on request when it is needed, which is when spawnd sends a boot request. This can only happen when system services like *chips* are already available.

On the SCC however, the driver is one of the first processes to be started. When *sif* is started, there is no connection to the host yet. When Barrelfish applications are started, normally *libbarrelfish* is initialized. Part of the initialization procedure of libbarrelfish is to connect to various services like the nameservice and *serial*. With no connection to the host, this is not possible and so *sif* has to start with an uninitialized version of the library and do the initialization of needed parts like the IDC system and a connection to the memory service themselves.

4.2.4.2 Monitor

Oh the SCC, the boot process is a little different from the boot process on "classical" x86 cores. Because the cores all have private physical address space, each core needs to start its own instance of a memory service. To do that, *init* is always started as if it runs on the bootstrap core, even if it doesn't. The distinction between bootstrap core and app core is made only in the *monitor*. The bootstrap core initializes the system and starts vital system services. When the SCC is started as an extension of the host, the first core to be started is neither bootstrap core nor application core. It behaves like an application core in terms of starting system services, but it does not connect to another SCC core. As the boot process is largely the same as for an application core, I decided that the first core should also be started as an application core, but with a slightly different boot procedure. I then added a flag to

the arguments that are given to the monitor on the SCC that allows it to determine whether it was started by the host and if so, choose a different initialization path.

During the boot process of an application core, the monitor requests the image of spawnd from the bootstrap monitor. Instead of transferring the whole image via messages, the bootstrap monitor sends the size and memory location of the image as a reply and the monitor on the application core can copy it to its own memory region where it should be executed. If the cores on the SCC request the image of spawnd from the bootstrap monitor, they get replies that are meaningless for them, so I had to reroute the request to a loopback function. Now the requests for spawnd images are answered by the monitor on core 0 on the SCC. The image location returned by this core is also valid on all other cores, because the same image is loaded for all cores initially.

I added a similar loopback for the UMP binding functions, because in the current version of Barrelfish, there are no checks whether UMP connections between cores are possible and the monitors would support the binding attempt using UMP. The loopback for the binding functions simply return an error to indicate that a connection with UMP is not possible and the application tries the next protocol.

4.2.4.3 Inter-SIF

The connection between the two driver parts is created in a way that is very similar to the way the inter-monitor UMP channels are created. The host part of the driver, like the bootstrap monitor, passes the memory location of the frame for the ringbuffer to the kernel of the new core as a parameter. The kernel then passes the information on to the monitor, which forwards it to *sif*. The driver on the SCC then allocates the specified memory region and initializes the connection.

With UMP, this is enough to fully establish a connection, but the SIFMP connection is at this point one-sided. Messages can be sent from the SCC to the host, but not in the other direction. The driver part on the host first needs to know the channel number to send with the IPI.

The monitor on the SCC sends this channel number, together with the capability for it to the driver using the *sif_mgmt* interface. The driver sends back a capability for the endpoint so the monitor can register it as the endpoint for the IPI channel. The first message that is sent between the two driver parts goes from the SCC to the host and it is part of a three-way handshake. Part of the first message is the channel number for the IPI. The messages are also used to set up the inter-monitor channel between the bootstrap monitor and the monitor on core 0 on the SCC.

When the handshake is done, the two driver parts can communicate over the PCIe bus. At this point, they export their *sif* service.

4.2.5 RPC Clients

When two messages in a Flounder interface belong together as call and response, they can be grouped as a remote procedure call (RPC). Flounder still generates two

separate messages and they can be used separately with the same semantics as a call and a response, but Flounder is also able to generate an additional connection interface, the RPC client. RPC clients are a way to use connections conveniently in Barrelfish. RPC clients group the call and the receiving of the request together as one function for the application to call, so the application does not have to care about asynchronous messages.

RPC clients are a meta-interface to Flounder connections. They wrap existing connections and place them in their own waitset. When an application invokes a RPC function on the RPC client, the client sends the call message through the wrapped connection and waits on the waitset of the RPC client for the response. When the response is received, the RPC client can return the values sent in the response.

The extra waitset allows the RPC client to block until the response is received, without the possibility to trigger other message callbacks. This makes RPC clients much easier to use in applications where a deadlock could occur.

For SIFMP connections, the possibility to receive messages while waiting for a response is crucial, since message notifications are sent through Flounder connections. This means that the connection to *sif* must always be in the same waitset as the SIFMP connection itself. For this reason, every SIFMP connection has its own connection to *sif*, which is always placed in the same waitset as the connection.

4.3 Evaluation

The resources needed by a SIFMP connection can not completely be described with an absolute number, because some of the resources are used by more than one connection or even more than one application.

Nevertheless, the usage of resources can be compared to UMP/UMP_IPI, which has a very similar concept.

UMP/UMP_IPI	SIFMP
1 frame	2 frames (1 on the host, 1 on the SCC)
1 IPI channel (only UMP_IPI)	1 connection to *sif*
1 waitset entry	1 waitset entry
	1 process (driver)

4.3.1 Benchmarks

I did some benchmarking of the SIFMP by using 7 testcases from the *bench/flounder_stubs* collection of messaging benchmarks in the Barrelfish repository. All the benchmarks were completed with a 99.9% confidence interval for the mean of ±1% or less. The following benchmarks were executed:

empty tests the speed of a null-message

buffer tests the speed of a 1-byte long buffer transmission

payload32_(1,2,4,8,16) test the speed of transmitting 1, 2, 4, 8 and 16 32bit integers

As is to be expected from a communication channel that crosses the PCIe bus, SIFMP is much slower than UMP (see figure 4.3). For SIFMP, the transmission time of a buffer is the same as the transmission time of the empty message or a message which can be transmitted in one package. With UMP, this is not the case and I can conclude that the marshalling time of SIFMP messages is a minor factor in the speed of SIFMP (see figure 4.4).

It can also be seen that the transmission time of messages that use more than one package to transmit is not proportional to the number of packages. This comes from the functioning of the buffer updating function in the driver, which becomes active after each received notification, but copies as many messages as available each time, so some messages are copied before the corresponding notification has arrived at the driver.

4.4 Future Work

4.4.1 Multicore SCC

My version of SIFMP was only tested with a single core running on the SCC, as support for multiple cores on the SCC did not work at the time of my work.

When multiple cores should be started, great care has to be taken in creating inter-monitor connections. If further SCC-cores should be started by the host, the boot command should be forwarded to the first core on the SCC and executed there or new cores need their own endpoint of *sif*.

4.4.2 Data Prefetch

The number of read accesses can be reduced by reading greater chunks of memory from the SCC. This creates a tradeoff between the number of read accesses and the time each read operation takes. The optimal size for each read access is something that has yet to be determined.

4.4.3 SCC-Host messaging

Apparently, a way to actively send a message from the SCC to the host exists. The Linux-tools and -drivers provided by Intel for the SCC make use of a mailbox that appears to be a specific memory region in the SIF on the SCC. The Linux-driver writes data into it, which then apparently appears as pending message in the PCIe device visible on the host.

This mechanism could be used to reduce the number of needed read operations over the PCIe bus by changing the periodic check for new messages in SCC memory

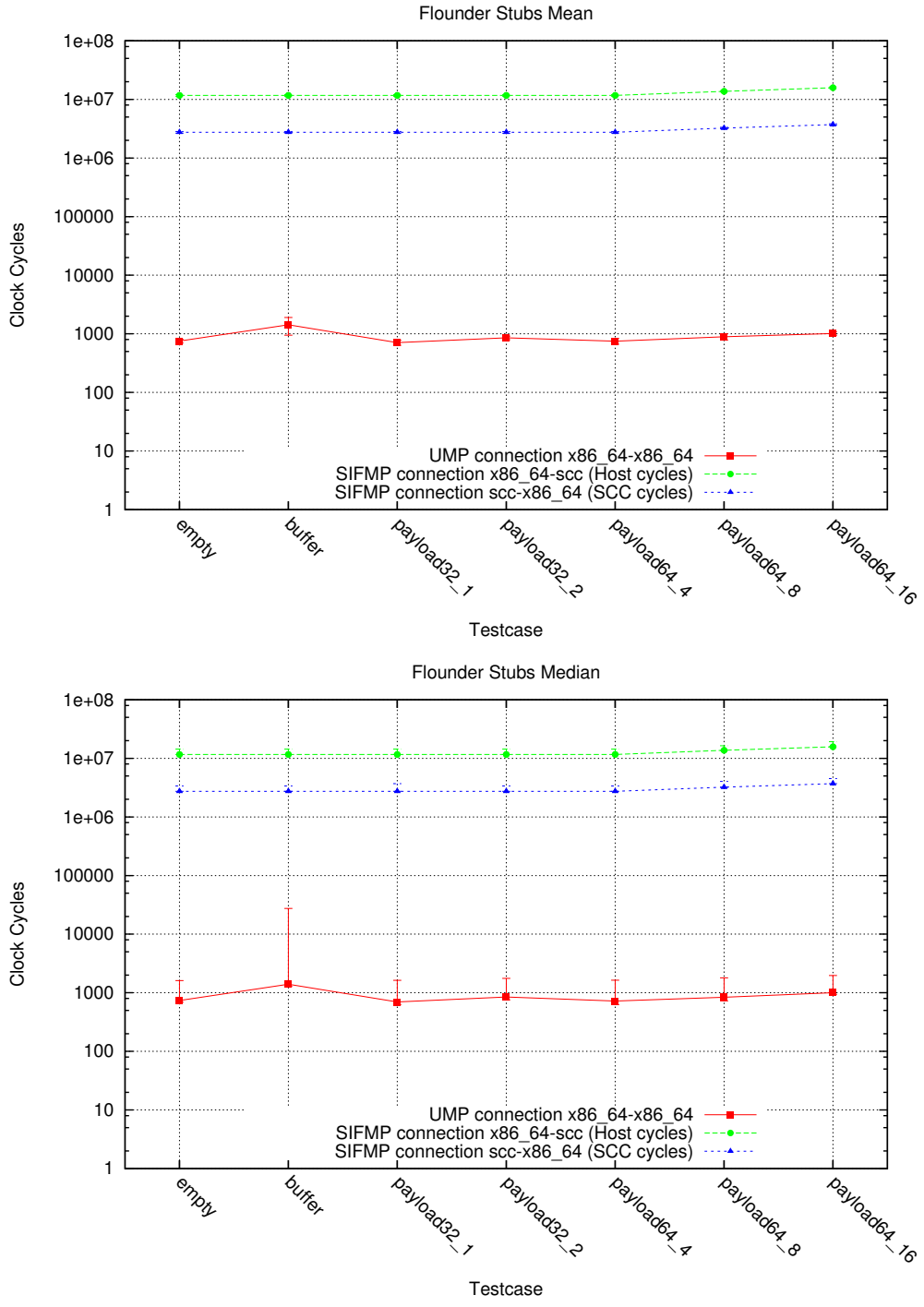

Figure 4.3: Comparison of SIFMP with host UMP connection (logarithmic scale)

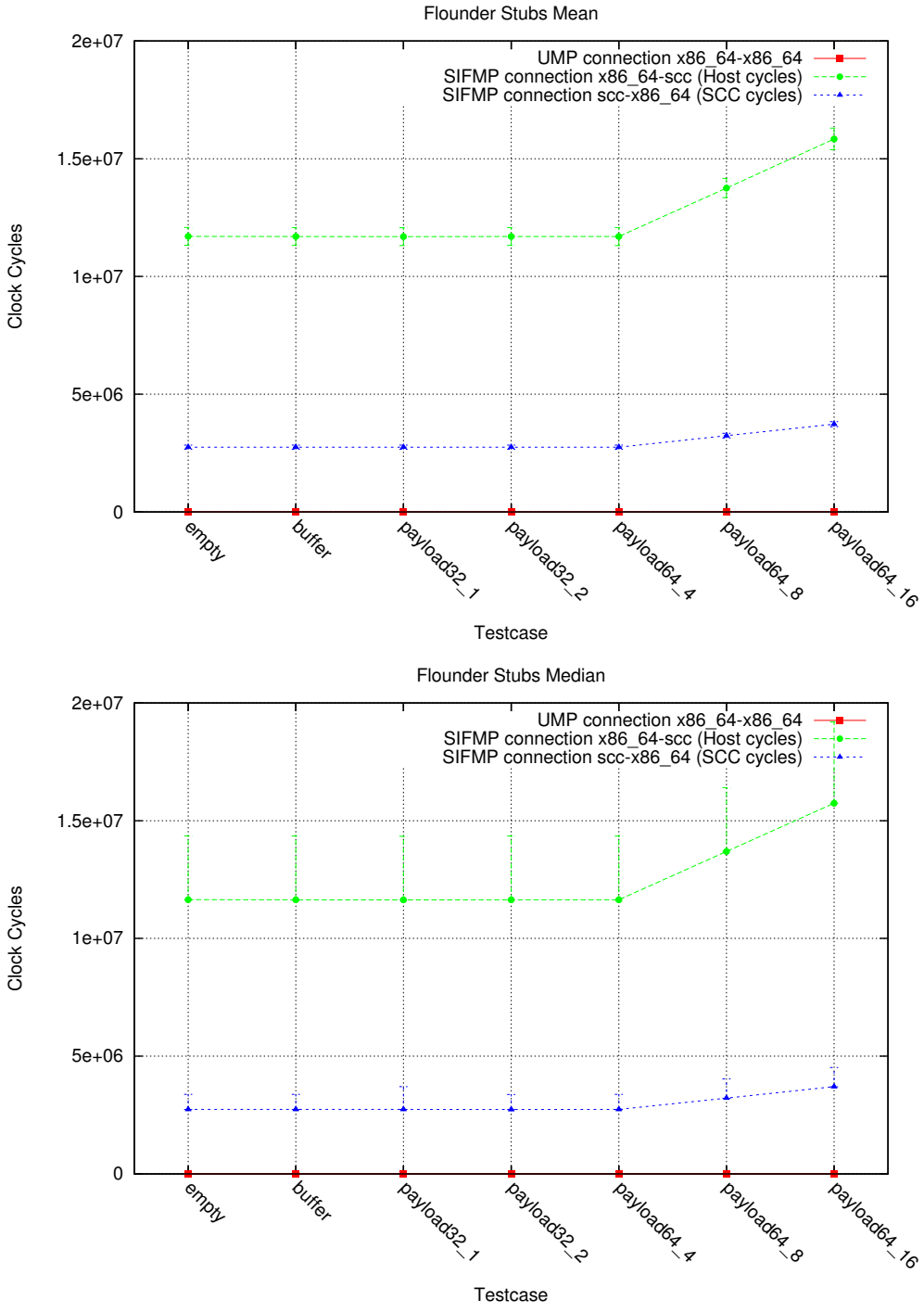

Figure 4.4: Comparison of SIFMP with host UMP connection (linear scale)

to a check for pending messages in the PCIe device or even by completely replacing the SCC-host inter-*sif* channel with the mailbox. However, this functionality is undocumented and I did not have time to test it.

4.4.4 IPI

Like UMP and UMP_IPI, SIFMP could be extended by the possibility to use IPIs instead of the connection to the *sif* driver to deliver notifications.

With this, the resources needed by each SIFMP connection could be reduced, as instead of using a whole UMP_IPI channel, only one IPI channel would be needed for a SIFMP notification connection.

Additionally, SIFMP connections might become faster when no additional connection and its unmarshalling process is involed.

4.4.5 Routing

SIFMP in its current state provides the possibility to connect processes on the host with processes on the SCC. If the capabilities of Barrelfish are enhanced to also span other peripheral devices like graphic cards and network cards, Barrelfish will probably need a routing infrastructure that not only allows the creation of connections with fixed protocols but also with "concatenations" of connections and protocols.

When Barrelfish is enhanced in such a way, the *sif* driver can be changed to provide routing capabilities. The driver would then become a gateway to the SCC (and to the host from the SCC).

5 Protocol Selection

During my work on SIFMP, I saw the need to change the way Flounder backends (protocols) are selected when a new connection is created in Barrelfish. In this chapter I will describe the problems with the old system and a first implementation of an improvement.

5.1 Overview

When a new connection is created in Barrelfish, the initiator of the connection has to chose the Flounder backend for this connection. In the old version of the initializer code generated by Flounder, the initiator would try each protocol in a trial and error method. The first connection attempt would be made with LMP. If that fails, it would try UMP_IPI if available on the architecture, then UMP, BMP and SIFMP.

This works when the number of Flounder backends is small and the backends can be partially ordered with respect to their requirements and unordered pairs are mutually exclusive on any single system. This is the case with the current range of Flounder backends available.

As soon as the number of backends increases, this trial and error method may not be feasible anymore, because each backend may need to allocate resources at the beginning of the connection attempt and release them again after it has failed, a process which takes time and creates unnecessary workload for the processes guarding the resources.

Also, when connection types become more complex, such as routed connections with multiple possible gateways, a more sophisticated way of choosing the Flounder backend and its parameters is needed.

5.2 Implementation

My implementation of a protocol selection method introduces an additional step in the connection creation process. Before a connection attempt is made, a process sends a message to its monitor, asking for the type of connection that should be used for communication between that core and the target core.

The monitor may be the bootstrap monitor, in which case the *skb* is running on the same core. The monitor queries the skb for the preferred connection type and sends it back to the process.

Backend	Requirements for connections between cores A, B
LMP	• A = B
UMP_IPI	• A, B have shared memory • (A, B have cache coherence for shared memory) • A can send an IPI to B • B can send an IPI to A
UMP	• A, B have shared memory • A, B have cache coherence for shared memory
SIFMP	• A ≠ B • A, C have shared memory (C ≠ B) • B, D have shared memory (D ≠ C, D ≠ A • C, D are connected by *sif* • there exists a backend for the connection (A, C) • there exists a backend for the connection (B, D)
BMP	• A, B cores on a Beehive processor

Table 5.1: Requirements table for Flounder backends

If the monitor is not the bootstrap monitor, it forwards the message to the monitor that booted it through the inter-monitor connection. This way the message will eventually reach the bootstrap monitor, which then sends back the reply along the same path.

When the reply reaches the process, the process can decide which Flounder backend to use and will start a connection attempt with that backend. Should the connection attempt fail, it does not try any other connection types, but returns an error to the callback function.

If a request for a connection type can't be forwarded to the skb, the process receives an error indicating this. The process will then use a fallback function to use the trial and error method. This will happen if the system was started without an skb or when the skb has not been started yet.

The decision which Flounder backend should be used is made by a skb module that takes available hardware information and compares them to requirements for the backends (see table 5.1).

5.3 Evaluation

The backend selection implementation avoids the trial and error method that was previously used in Barrelfish as much as possible without adding heuristics which can be used without contacting the skb.

The implementation uses the connection to the monitor of a process to get the information. This seems reasonable, since the monitor is already responsible for

helping processes to create connections.

The implementation uses only the skb, the monitor and the inter-monitor connections to query for the preferred Flounder backend. As the monitor and the inter-monitor connections should be available before any other services, this method has very low requirements to work.

5.4 Future Work

The implementation of the protocol selection is very rudimentary and does not yet have the means to provide the connection initiating process with more information about the preferred protocol, such as ringbuffer size or routing information. With more complex architectures, which will probably be created in the future, this may become a requirement.

6 Related Work

Heterogeneity

A lot of work has been done on how to make use of spare computing power on devices in computer systems to move work off the host CPU. In this section I am going to describe some of the work that has been done in this area.

Helios

In [NHM+09] Nightingale et al. present the Helios operating system, which is designed to run on a heterogeneous platform. Helios is based on Singularity [HL04], an operating written almost entirely in Sing#, which is an extension to the C# programming language.

Helios extends the Singularity operating system with satellite kernels. Satellite kernels are microkernels that don't require a lot of hardware primitives to run. The Helios satellite kernels require the device on which it runs to have a timer, an interrupt controller and the ability to use traps. With this, Helios satellite kernels are able to run on general purpose CPUs with or without NUMA domains, network cards or future graphics cards such as the Intel Larrabee.

Because applications written for Helios and Singularity, for that matter, are compiled to byte-code of the .NET platform and compiled at runtime, no changes have to be made for applications to run on any satellite kernel, on any architecture.

Helios extends the messaging system in Singularity by a remote message passing mechanism, so that applications located on different satellite kernels can communicate. This remote message passing mechanism is integrated transparently into the existing messaging system, so that applications can use it without changes to the code.

Hera-JVM

The Hera-JVM [MS09] is an implementation of a Java Virtual Machine based on the JikesRVM that runs on the IBM Cell processor [1], a processor with two different architecture types on a single die, i.e. a heterogeneous system.

The IBM Cell processor consists of 9 processors of two types: a single PowerPC processor called *Power Processing Element* (PPE) and 8 *Synergistic Processing Elements* (SPE). The PPE is a general purpose processor, while the 8 SPEs are

[1] http://www.research.ibm.com/cell/

co-processors, designed to do most of the work, but which can not run on their own and require the help of the PPE to do any useful work.

A Java thread can be run entirely on the PPE if that is the developer's intent or it can switch core type on any method invocation. Therefore, and because the Hera-JVM uses a JIT-compiler only, Java applications are compiled on a per-method basis. Since most methods only ever run on one processor type, most of the Java byte-code has to be compiled only once, either for the PPE or SPEs. The compiler inserts code to trap to migration support code automatically if a method should be run on a particular processor. As soon as amethod that was to be executed on another core type returns, the thread is migrated back to the originating processor where it resumes execution.

To cope with the SPEs' non memory-coherent local cache, the Hera-JVM uses a software cache mechanism to cache entire Java-objects and methods. Before accessing fields of an object or calling a method, the compiler adds inline code to check whether the object or method in question is already in the cache and if not traps to a handler that copies it from main memory to the software cache.

Harmony

In [DY08] Diamos et al. propose a system to effectively create applications for heterogeneous systems in the HPC area called Harmony.

Harmony consists of a programming model and the Harmony runtime. Harmony applications consist of a code segment invoking the execution of compute kernels and control decisions. Compute kernels can be compared to functions in imperative programming languages. They can be invoked, take input arguments and define a set of output arguments. The order of execution of compute kernels can be restricted by control decisions by specifying dependencies of kernels on other kernels. Compute kernels can be shipped with multiple implementations for different architectures, but all of them have to produce the same output for a set of input arguments, i.e. they must conform to the same specification.

As a Harmony program executes, compute kernels are invoked and scheduled by the runtime on any device for which an implementation of that compute kernel exists, and if possible on the device that is best suited for the task at hand.

Since dependencies could slow down computation if they impose a sequential execution of kernels, the Harmony runtime is able to speculatively execute kernels akin to branch prediction in imperative programming languages and recover in the case of a misspeculation.

Hydra

Weinsberg et al. present a system for offloading parts of applications to devices better suited for the task in [WDA+08]. The proposed system is called Hydra and consists of a programming model, host runtime support and runtime support for target devices.

In Hydra, applications are split into components that may either be implemented as conventional parts of the application running on the host processor or as Offcodes. Offcodes are specified by an *Offcode Description File* (ODF) and can implement one or more interfaces. Offcodes can express multiple kinds of dependencies on other Offcodes, such as the requirement to load an Offcode on the same device or to load it on its preferred device. Also, preferred device-classes for the Offcode can be specified, such as NICs or graphics cards.

When an offload-aware application (OA) is started, it may invoke the Hydra runtime to create and deploy an Offcode. The runtime creates a mapping of Offcodes to devices that satisfies any constraints given by the ODFs and invokes an appropriate compiler or linker and does the actual offloading. Dynamic loading of Offcodes can also be done on either the host CPU or on-demand by the target devices.

Once an Offcode is deployed on a target device, the OA has a default channel to the Offcode and with it the possibility to create specialised channels for specific purposes or to call functions of interfaces implemented by the Offcode.

SPINE

In [FMOB98] Fiuczynski et al. present SPINE, a system to offload parts of applications onto I/O devices.

The SPINE runtime consists of a host component called the *SPINE kernel runtime* and a I/O device component called *SPINE I/O runtime*. SPINE extensions, which are written in Modula-3, have access to the standard Modula-3 interfaces, means of communication with other extensions and the host component as well as access to hardware such as DMA and network send/receive engines.

The host application and SPINE extensions communicate by using a messaging mechanism. The messages can either be small or bulk messages and messages are routed to either the host or I/O devices by a dispatcher.

GPU offloading

GPU computing was made easier with the introduction of CUDA [NVI10] and more recently with OpenCL [SGS10].

CUDA and OpenCL enable the use of a subset of C to write kernels for offloading on GPUs and other heterogeneous platforms such as the IBM Cell processor.

Fast IPC

L3

Liedtke describes in [Lie93] a modification of the L3-kernel, a microkernel of the second generation. Since it is widely believed that microkernels can not perform well because IPC is too slow, and former attempts to reduce the time needed for an

IPC weren't very successful, they tried to make IPC faster by completely redesigning the kernel. They were able to drastically reduce the time needed to complete an IPC-call by focusing the design of the whole kernel on fast IPC. The system avoids superfluous copying on calls, tries to reduce cache misses and reduces TLB misses by careful placement of important structures in memory. The resulting time taken to execute an IPC call is only twice the minimum time imposed by the hardware.

Today, the L3 microkernel has mostly been replaced by its successor, the L4 microkernel [Lie95], and its different variations.

Batched Futures

Bogle and Liskov present a system called *Batched Futures* [BL94] that is capable of substantially reducing the number of cross domain calls done when interacting with a object oriented database. Often when accessing objects in an object-oriented database, the calls don't do much work on the server side, but take a long time to complete nevertheless, because the cross domain calls are so expensive. With Batched Futures, calls to the database which only return handles to objects are not executed when the call is made by the application, but only when the handle is actually used to retrieve or store a value or otherwise access a feature of the object. The system then sends all buffered calls to the database in a single call and return the value requested by the application.

MPICH(-G)

In [GLDS96], a message passing library called MPICH, which implements the MPI standard, is presented. MPICH aims to provide a high degree of portability and high performance. The system is heavily layered and allows for architecture- specific optimizations at different layers of the library. In [FK98] the library is extended to run on heterogeneous grid environments. Although the libraries run on top of existing systems and are not part of the operating system itself, the aim for portability and high performance are the same as in the message-passing system in Barrelfish, which also uses multiple layers to ease the use of the message-passing infrastructure.

SCC

SCC programmer's manual

The Intel Single-Chip Cloud Computer (SCC) platform [Int10] is a research processor developed by Intel. It features a tile-oriented architecture wich 24 tiles on a chip with 2 cores on each tile with a total of 48 cores on a single chip. The tiles are connected by routers in a grid-like network. The SCC features 4 memory controllers, each capable of managing up to 16GB of DDR3 memory, making a total

of 64GB of memory accessible from the SCC cores. The memory in the SCC is divided into different types, namely Message Passing Buffer memory, core-specific memory and shared memory. The Message Passing Buffer memory is located on the tiles and has special semantics that allow them to be used for message-passing between the cores, while the core-specific memory and shared memory is located in the main memory managed by the memory controllers. Memory access on the SCC is not covered by cache-coherence, which makes it a great platform for systems like Barrelfish.

7 Conclusion

As a conclusion I can say that I was able to produce two extensions for Barrelfish, which both enable it to span a heterogeneous set of processors. With that I have achieved my goal to create a version of Barrelfish with support for heterogeneous cores.

Both extensions work reliably and integrate well into the existing communication infrastructure in Barrelfish.

The extension that allows Barrelfish to have connections from x86_64 cores to x86_32 cores is fast and has the potential to be extended to include more architectures as it uses architecture-independent messages that could be written or read on any architecture that also supports these types of messages.

SIFMP is a first attempt to create connections between a host and an attached PCIe device in a way that applications are not concerned with the difficulties in creating such connections. SIFMP works reliable, but is still slower than it could be. SIFMP may also be too focused on connections between a host and a SCC and does not support connections to other devices.

I think Barrelfish is very well suited as a basis for a OS that spans cores, CPUs and devices of different architecture, as the aspect of separated but connected computing units is part of the multicore architecture of Barrelfish. However, much work like routing and capability transfer remains to be done.

A SCC

Default memory mapping

LUT #	Physical Address	
255	FFFFFFFF - FF000000	Private (Boot Page)
254	FEFFFFFF - FE000000	
253	FDFFFFFF - FD000000	
252	FCFFFFFF - FC000000	
251	FBFFFFFF - FB000000	VRC
250	FAFFFFFF - FA000000	Management Console TCP/IP Interface
249	F9FFFFFF - F9000000	Route to Host for eMAC access
248	F8FFFFFF - F8000000	Core's own System Configuration Register
247	F7FFFFFF - F7000000	System Configuration Register – Tile 23
246	F6FFFFFF - F6000000	System Configuration Register – Tile 22
245	F5FFFFFF - F5000000	System Configuration Register – Tile 21
244	F4FFFFFF - F4000000	System Configuration Register – Tile 20
243	F3FFFFFF - F3000000	System Configuration Register – Tile 19
242	F2FFFFFF - F2000000	System Configuration Register – Tile 18
241	F1FFFFFF - F1000000	System Configuration Register – Tile 17
240	F0FFFFFF - F0000000	System Configuration Register – Tile 16
239	EFFFFFFF - EF000000	System Configuration Register – Tile 15
238	EEFFFFFF - EE000000	System Configuration Register – Tile 14
237	EDFFFFFF - ED000000	System Configuration Register – Tile 13
236	ECFFFFFF - EC000000	System Configuration Register – Tile 12
235	EBFFFFFF - EB000000	System Configuration Register – Tile 11
234	EAFFFFFF - EA000000	System Configuration Register – Tile 10
233	E9FFFFFF - E9000000	System Configuration Register – Tile 09
232	E8FFFFFF - E8000000	System Configuration Register – Tile 08
231	E7FFFFFF - E7000000	System Configuration Register – Tile 07
230	E6FFFFFF - E6000000	System Configuration Register – Tile 06
229	E5FFFFFF - E5000000	System Configuration Register – Tile 05
228	E4FFFFFF - E4000000	System Configuration Register – Tile 04
227	E3FFFFFF - E3000000	System Configuration Register – Tile 03
226	E2FFFFFF - E2000000	System Configuration Register – Tile 02
225	E1FFFFFF - E1000000	System Configuration Register – Tile 01

Table A.1: Default memory mapping for 16 GB of main memory

LUT #	Physical Address	
224	E0FFFFFF - E0000000	System Configuration Register – Tile 00
223	DFFFFFFF - DF000000	
:	:	:
217	D9FFFFFF - D9000000	
216	D8FFFFFF - D8000000	Core's own MPB
215	D7FFFFFF - D7000000	MPB in Tile (x=5,y=3)
214	D6FFFFFF - D6000000	MPB in Tile (x=4,y=3)
213	D5FFFFFF - D5000000	MPB in Tile (x=3,y=3)
212	D4FFFFFF - D4000000	MPB in Tile (x=2,y=3)
211	D3FFFFFF - D3000000	MPB in Tile (x=1,y=3)
210	D2FFFFFF - D2000000	MPB in Tile (x=0,y=2)
209	D1FFFFFF - D1000000	MPB in Tile (x=5,y=2)
208	D0FFFFFF - D0000000	MPB in Tile (x=4,y=2)
207	CFFFFFFF - CF000000	MPB in Tile (x=3,y=2)
206	CEFFFFFF - CE000000	MPB in Tile (x=2,y=2)
205	CDFFFFFF - CD000000	MPB in Tile (x=1,y=2)
204	CCFFFFFF - CC000000	MPB in Tile (x=0,y=2)
203	CBFFFFFF - CB000000	MPB in Tile (x=5,y=1)
202	CAFFFFFF - CA000000	MPB in Tile (x=4,y=1)
201	C9FFFFFF - C9000000	MPB in Tile (x=3,y=1)
200	C8FFFFFF - C8000000	MPB in Tile (x=2,y=1)
199	C7FFFFFF - C7000000	MPB in Tile (x=1,y=1)
198	C6FFFFFF - C6000000	MPB in Tile (x=0,y=1)
197	C5FFFFFF - C5000000	MPB in Tile (x=5,y=0)
196	C4FFFFFF - C4000000	MPB in Tile (x=4,y=0)
195	C3FFFFFF - C3000000	MPB in Tile (x=3,y=0)
194	C2FFFFFF - C2000000	MPB in Tile (x=2,y=0)
193	C1FFFFFF - C1000000	MPB in Tile (x=1,y=0)
192	C0FFFFFF - C0000000	MPB in Tile (x=0,y=0)
191	BFFFFFFF - BF000000	
:	:	:
132	84FFFFFF - 84000000	
131	83FFFFFF - 83000000	Shared MCH3 - 4MB
130	82FFFFFF - 82000000	Shared MCH2 - 4MB
129	81FFFFFF - 81000000	Shared MCH1 - 4MB
128	80FFFFFF - 80000000	Shared MCH0 - 4MB
127	7FFFFFFF - 7F000000	
:	:	:
21	15FFFFFF - 15000000	
20	14FFFFFF - 14000000	

Table A.1: Default memory mapping for 16 GB of main memory

LUT #	Physical Address	
19	13FFFFFF - 13000000	Private
18	12FFFFFF - 12000000	Private
:	:	:
1	01FFFFFF - 01000000	Private
0	00FFFFFF - 00000000	Private

Table A.1: Default memory mapping for 16 GB of main memory

Memory controller memory layout

Physical Address		
FFFFFFFF - FF000000	Boot Page core 17	Boot pages of cores 0-5 and 12-17. These are placed at the end of the core's physical memory
FEFFFFFF - FE000000	Boot Page core 16	
:	:	
FAFFFFFF - FA000000	Boot Page core 12	
F9FFFFFF - F9000000	Boot Page core 5	
:	:	
F5FFFFFF - F5000000	Boot Page core 1	
F4FFFFFF - F4000000	Boot Page core 0	
F3FFFFFF - F3000000		This memory region is unused
F2FFFFFF - F2000000		
F1FFFFFF - F1000000	Default Memory	This memory region is mapped as default if no other memory region should be mapped in a LUT entry
F0FFFFFF - F0000000	Shared Memory	
EFFFFFFF - EF000000	Private Memory of core 17	Private pages of cores 0-5 and 12-17 without their last pages (boot pages)
EEFFFFFF - EE000000	Private Memory of core 17	
:	:	
DCFFFFFF - DC000000	Private Memory of core 17	
DBFFFFFF - DB000000	Private Memory of core 16	
:	:	
79FFFFFF - 79000000	Private Memory of core 12	
78FFFFFF - 78000000	Private Memory of core 12	
77FFFFFF - 77000000	Private Memory of core 5	
76FFFFFF - 76000000	Private Memory of core 5	
:	:	
64FFFFFF - 64000000	Private Memory of core 5	
63FFFFFF - 63000000	Private Memory of core 4	
:	:	
01FFFFFF - 01000000	Private Memory of core 0	

Table A.2: Memory layout for memory controller at (0,0)

Physical Address		
00FFFFFF - 00000000	Private Memory of core 0	

Table A.2: Memory layout for memory controller at (0,0)

SCC FPGA message format

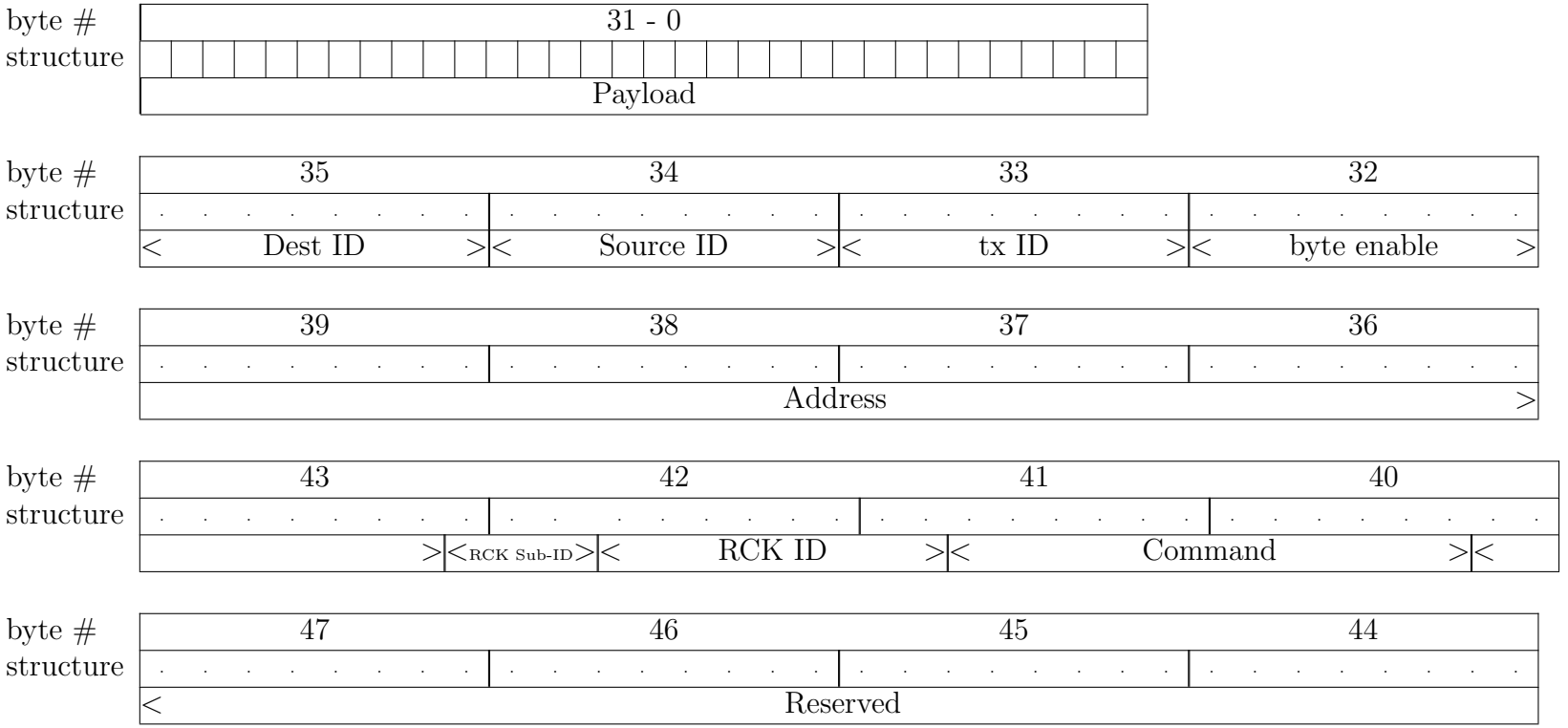

Table A.3: Message format for the SCC FPGA

B Benchmarks

For each benchmark, there are two graphs in this document, one depicting the mean execution times and one for the median execution times. In the graphs for the mean times, the ranges around the data points show the standard deviation of the test results. In the graphs for the median times, the ranges below the data points show the smallest measured value and the ranges above the data points show the greatest measured value for that experiment.

All the benchmarks were completed with a 99.9% confidence interval for the mean of at most ±2%.

For all the benchmarks a minimal set of Barrelfish services was loaded, consisting of the following processes:

- cpu
- init
- mem_serv
- monitor
- chips
- ramfsd
- skb
- pci
- spawnd
- sif (only for benchmarks involving SIFMP)
- flounder_stubs_***_bench (only for message protocol benchmarks)
- sif_bench (only for SCC hardware benchmarks)

All benchmarks involving the SCC were done on the machine *tomme1* or the identical *tomme2*.

All benchmarks for the heterogeneous communication between x86_64 and x86_32 were done on the machine *nos5*.

tomme1/2

CPU: Intel Gainestown (Nehalem-EP) (Intel Xeon L5520)
/proc/cpuinfo under Linux says:

family	6
model	26
stepping	5
clock speed	2266.755MHz
cache size	8192kB
clflush size	64
cache_alignment	64

Motherboard: SUN FIRE X2270

nos5

CPU: AMD Santa Rosa (Opteron 2200)
/proc/cpuinfo under Linux says:

family	15
model	65
stepping	3
clock speed	2800MHz
cache	1024kB
TLB size	1024 4K pages
clflush size	64
cache_alignment	64

Motherboard: Tyan Thunder n6650W (S2915)

Bibliography

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[BL94] Phillip Bogle and Barbara Liskov. Reducing cross domain call overhead using batched futures. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, OOPSLA '94, pages 341–354, New York, NY, USA, 1994. ACM.

[DY08] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. ACM.

[FK98] Ian Foster and Nicholas T. Karonis. A grid-enabled mpi: message passing in heterogeneous distributed computing systems. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–11, Washington, DC, USA, 1998. IEEE Computer Society.

[FMOB98] Marc E. Fiuczynski, Richard P. Martin, Tsutomu Owa, and Brian N. Bershad. Spine: a safe programmable and integrated network environment. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, EW 8, pages 7–12, New York, NY, USA, 1998. ACM.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789 – 828, 1996.

[HL04] Galen C. Hunt and James R. Larus. Singularity tech report 1: Singularity design motivation. Technical report, Microsoft Research, 2004.

[Int10] Intel Labs. *Single Chip Cloud Computing (SCC) Platform Overview*, rev. 0.7 edition, May 2010. http://techresearch.intel.com/spaw2/uploads/files/SCC_Platform_Overview.pdf.

[Lie93] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[Lie95] J. Liedtke. On micro-kernel construction. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[MS09] Ross McIlroy and Joe Sventek. Hera-jvm: abstracting processor heterogeneity behind a virtual machine. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 15–15, Berkeley, CA, USA, 2009. USENIX Association.

[NHM+09] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM.

[NVI10] NVIDIA. *NVIDIA CUDA C Programming Guide*, 9 2010. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[SGS10] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12:66–73, May 2010.

[SSBR08] A. Schupbach, Peter Simon, Andrew Baumann, and Timothy Roscoe. Embracing diversity in the Barrelfish manycore operating system. 2008.

[WDA+08] Yaron Weinsberg, Danny Dolev, Tal Anker, Muli Ben-Yehuda, and Pete Wyckoff. Tapping into the fountain of cpus: on operating system support for programmable devices. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 179–188, New York, NY, USA, 2008. ACM.