Implementing MINIX on the Single Chip Cloud Computer

Author:
Niek Linnenbank

First Reader:
Andrew S. Tanenbaum

Second Reader:
Dirk Vogt

August 22, 2011

1843370

Abstract

In this work we implemented the MINIX operating system on the Single Chip Cloud Computer, a research prototype manycore computer by Intel. We redesigned MINIX such that it is able to run on the SCC with a fast message passing IPC mechanism. Our implementation supports different modes for message passing which we benchmarked and analyzed for performance. Although future work on MINIX is still required to transform it in a fully mature manycore operating system, we provided a solid basis for further development and experimentation on the SCC platform.

Acknowledgements

First and foremost, I would like to thank Intel for providing us with the SCC hardware. Without Intel's generous donation, this work would never exist. Additionally, great many thanks go to my supervisors: Tomas Hruby, Dirk Vogt and Andrew Tanenbaum. With the excellent guidance of Tomas and Dirk, I was able to create solutions to the many hard challenges we faced in our work. It has been a great pleasure for me to do this work and I am very happy to have learned so much at the Vrije Universiteit Amsterdam.

– Niek Linnenbank, August 22, 2011

Contents

List of Figures

Glossary

2D	Two Dimensional, 27, 28
A.OUT	Executable file format for UNIX systems, 47
ACPI	Advanced Configuration and Power Interface, 43, 45
AHCI	Advanced Host Controller Interface, 24
APIC	Advanced Programmable Interrupt Controller, 25
ATA	Advanced Technology Attachment, 24
BIOS	Basic Input and Output System, 15, 43–45
BMC	Board Management Controller, 31
bug	Programming error in a computer program, 12, 13
CD-ROM	Compact Disc Read Only Memory, 15, 43, 45
CL1INVMB	Clear Level 1 Cache with Message Passing Buffer Type bit instruction, 27
CMOS	Complementary Metal Oxide Semiconductor, 15
CMP	Chip Level Multiprocessing, 10
CPU	Central Processing Unit, 10–13, 15, 16, 21, 24, 27–29, 31, 33–36, 38–42, 44, 47, 48, 54, 57, 65
DDR3	Double Data Rate type three, 27, 29, 31
die	Silicon computer chip, 27
DS	Datastore Server, 15, 22, 39, 44, 47, 48
EXT2FS	Extended 2 File System, 23
FPGA	Field Programmable Gate Array, 27
GB	Gigabyte, 29
GUI	Graphical User Interface, 31
I/O	Input and Output, 10, 12, 13, 21, 23, 28, 43
ID	Identity, 29, 38, 39, 44, 56
INET	Internet Network Server, 23, 45
IOAPIC	Input and Output Advanced Programmable Interrupt Controller, 25, 43

IP	Internet Protocol, 23, 24, 31
IPC	Inter Process Communication, 12, 19, 21, 23, 33, 34, 36, 38, 39, 41, 42, 44, 49, 50, 58
IPI	Inter Processor Interrupt, 25, 28, 42, 43, 50, 54, 57, 58, 60, 67
JMFS	Journaled MINIX File System, 23
KB	Kilobyte, 27
LUT	Lookup Table, 28, 29
MB	Megabyte, 29, 31, 44
MBR	Master Boot Record, 15
MCPC	Management Control Personal Computer, 27, 31, 44, 45
MFS	MINIX File System, 15, 24
MHz	Megahertz, 27, 29, 43
MIU	Mesh Interface Unit, 27, 28
MMU	Memory Management Unit, 29
MPB	Message Passing Buffer, 27, 29, 31, 34, 40–44, 48–50, 54, 57, 58, 60, 67
OS	Operating System, 11, 45
P54C	Intel Pentium Processor, 27, 29, 45, 58
PC	Personal Computer, 15, 16, 21, 27, 43
PCI	Peripheral Component Interconnect, 21, 23, 45
PCI/e	Peripheral Component Interconnect Express, 27, 31
PFS	Pipe File System, 15
PIC	Programmable Interrupt Controller, 21, 43
PID	Process Identity, 21, 35, 36, 38, 44
PIT	Programmable Interval Timer, 16, 43
PM	Process Manager, 15, 23, 34, 35, 44, 56
POSIX	Portable Operating System Interface for UniX, 13
PS2	Personal System/2, 21, 44
RAM	Random Access Memory, 15, 45
RS	Reincarnation Server, 15, 22, 34, 47
SCC	Single Chip Cloud Computer, 10, 11, 23, 26–29, 31, 33, 34, 36, 40, 43–45, 47–50, 56–58, 60, 65–67
SMP	Symmetric Multi Processing, 10, 11, 24, 33, 34, 43, 58, 67
source code	Human readable program text, which may be interpreted or compiled into machine language for execution by the computer, 66
TCP	Transmission Control Protocol, 23, 24, 31
Tera	1 trillion or 1,000,000,000,000, 26

TLB	Translation Lookaside Buffers, 13
TTY	Teletype Writer, 15, 23, 44
UART	Universal Asynchronous Receiver Transmitter, 43, 44
UDP	User Datagram Protocol, 23, 24
VFS	Virtual File System, 15, 23, 24, 34
VGA	Video Graphics Array, 44
VM	Virtual Memory, 34, 44

Chapter 1

Introduction

Since the introduction of microprocessor based computers several decades ago, manufacturers have repeatedly found clever ways to increase the speed of a single CPU. The methods engineers used to speedup a CPU are increasing clock frequencies, cache sizes and instruction level parallelism [1]. For a long time these techniques helped to make computers increasingly faster. However, at the same time it became increasingly harder to continue this trend [2, 3]. First, due to the progressivly smaller chip geometries and higher clock frequencies the transistor leakage current increases, leading to excessive power consumption and heat. Second, over time processor speed grew relatively faster than memory speed [4, 5]. This high memory access latency prevents further performance gains. In an attempt to further increase computing power, hardware designers decided to increase the number of CPUs instead [6]. This led to the Symmetric Multi Processing (SMP) computer systems widely available today. Recent processors from Intel take the level of parallelism even further with Chip Level Multiprocessing (CMP), adding multiple execution cores per CPU. Although current hardware now provides more performance potential due to more parallelism, the operating system software must also change to actually gain higher performance.

Most modern operating systems for SMP architectures assume a consistent view of shared memory. Typically all bookkeeping information is in shared memory and locking is used to avoid concurrent updates to the same information. In SMP architectures each CPU has their own private memory cache. The hardware guarantees to the software that these memory caches are kept synchronized when CPUs write to memory. Effectively this gives software developers always a consistent view of memory which makes the software more simple. However, when adding more CPUs to an SMP system, under the hood more synchronization traffic is needed to keep all private memory caches coherent. Although current operating systems attempt to temporarily deal with this limitation by avoiding sharing, it makes further scaling of SMP computer systems difficult.

Intel introduced a new computer system called the Single Chip Cloud Computer (SCC) [7] which removes cache coherence. The SCC is an experimental 48-core computer system for exploring the possibilities of cache incoherent hardware.All cores in the SCC are on a single die, interconnected by a novel mesh network. In addition, the SCC has hardware support for message passing and allows software developers to scale frequencies and voltages on much finer granularity. The internal architecture of the SCC allows hundreds to thousands of cores in future versions, which challenges operating system designers to scale the software accordingly. Although current operating systems can scale up to a few dozen cores [8], scaling to hundreds or thousands of cores with many concurrent applications and I/O requests is still an unresolved issue.

MINIX 3 [9] is an operating system which can potentially be the answer. In contrast to traditional monolithic operating system kernels, MINIX is a microkernel based OS. All well known operating system functionality like device drivers, file systems and memory management are implemented as unprivileged processes in MINIX. The processes communicate by sending small messages to each other using IPC primitives offered by the kernel. On a multicore system these processes can run concurrently on different CPUs. Additionally, since the server processes in MINIX share little memory (if at all), overhead due to locking is minimal. Moreover if the number of CPUs is greater than the number of processes, context switching may not be needed. Therefore MINIX potentially achieves high scalability on multicore.

In this work we modified MINIX to run on the SCC. Since MINIX was originally designed for uniprocessor and SMP computer systems, we introduced design changes which allow MINIX to function in a cache incoherent environment. Additionally, we changed several core components in MINIX which depend on hardware devices unavailable on the SCC. Especially the startup procedure of MINIX required a complete rewrite for the SCC. Finally, we extended the message passing code in MINIX to support the SCC hardware based messaging and evaluated different implementations for message passing.

The rest of this thesis is structured as follows. Chapter 2 describes the MINIX operating system and the SCC in detail. In Chapter 3 we discuss the design choices we made and what alternative designs we considered. Chapter 4 is a detailed description of the implementation of MINIX on the SCC. Chapter 5 evaluates the the MINIX SCC implementation including performance results. In Chapters 6 and 7 we discuss future and related work, respectively. We conclude in Chapter 8.

Chapter 2

Background

In this Chapter we describe relevant background information which is needed to understand the design and implementation of our work. First, we introduce the microkernel operating system architecture in Section 2.1, which is relevant for understanding MINIX in Section 2.2. Second, we give a technical overview and details of the SCC in Section 2.3.

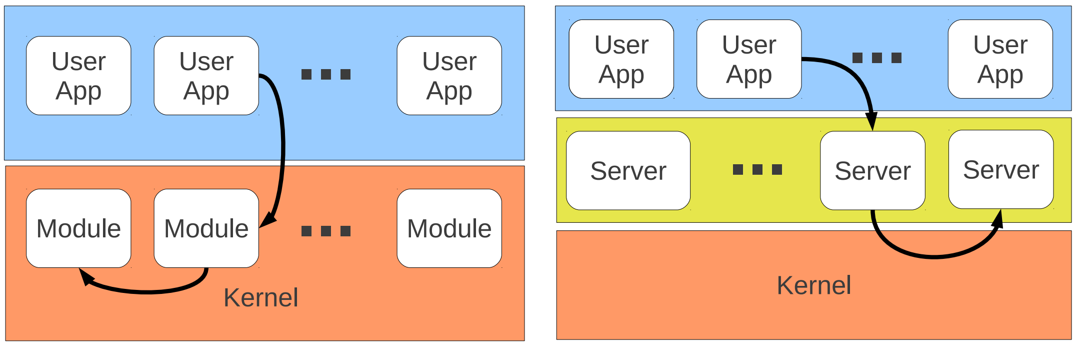
2.1 Microkernels

When creating an operating system, software developers must choose a *kernel* design. The kernel is a software component which runs in priviledged mode. When user applications need to access hardware, they must invoke the kernel to perform the required action on their behalf. A fundemental design choice of a kernel is to determine what functionality it should have.

Figure 2.1 illustrates two different ways to design the kernel. As shown in Figure 2.1a, the *monolithic kernel* implements most of the operating system functionality, such as file systems, device drivers and memory management, inside the kernel. For example, if a user application needs to read a file, it asks the kernel to retrieve the corresponding data blocks from disk by doing a *kernel trap*. A kernel trap switches the CPU mode from unprivileged to privileged and begins executing kernel code. Inside the kernel, the file system module calls the disk module to read the data blocks from disk. Finally, the kernel switches back to the user application code.

The *microkernel* design in Figure 2.1b implements the majority of OS functionality in server processes, which are scheduled by the microkernel. The server processes communicate by message passing, with help from the kernel. For example, whenever a user application needs to read a file, it asks the kernel to deliver a message to the corresponding file server process using a kernel trap. On receipt, the file server process in turn sends a message to the disk driver server to read the data blocks. Finally, the file server replies back to the user application.

One of the advantages of the microkernel design is that the kernel itself contains minimum functionality. Microkernels only offer three major functions: scheduling processes, Inter Process Communication (IPC) and I/O handling. In contrast to monolithic kernels, the small amount of code in microkernels reduces the risk of kernel bugs which are often fatal [10]. Additionally, bugs in server processes not neccessarily affect other running processes. Since each server in the microkernel design runs in a separate address space, it cannot overwrite memory belonging to other processes. Even after a server process crashes, it can be restarted. Of course, care must be taken to avoid loss of state if a server process crashes.



(a) Monolithic Kernel (b) Microkernel

Figure 2.1: Monolithic versus Microkernel operating system design.

Due to the componentized design of a multiserver operating system, it is easier to update components [11] while the system is running. For example, if a software developer creates a new version of a network driver to correct a bug, he can replace the currently running network driver without disrupting other servers, drivers or active network connections. Although it is also possible to live update monolithic kernels [12], it is easier to update server processes in a microkernel as they are well isolated. Moreover, in contrast to monolithic kernels, if newly updated components crash it does not bring down the entire system, which makes updates less risky. Additionally, an older version can be restored.

Although microkernels are more reliable than monolithic kernels, the design does not come for free. Scheduling server processes requires *context switches*. A context switch means changing currently active memory mappings and CPU registers to the last saved state of a process. Monolithic kernels do not need context switches when a user application performs a kernel trap. Therefore, microkernels are conceptually slower than monolithic kernels. However, the L4 microkernel [13] proved that microkernels can implement fast IPC mechanisms and that context switching overhead may not be significant. Additionally, some hardware architectures such as MIPS [14] offer better support for fast context switching using software managed *Translation Lookaside Buffers* (TLB). Moreover, with the uprise of multicore systems the context switching overhead can be spread among the cores. If the number of cores is higher than the number of processes, context switching can be completely avoided.

2.2 MINIX

MINIX 3 is a reliabable secure operating system created at the Vrije Universiteit Amsterdam for research, education and small embedded devices. From the start MINIX 3 is designed with a number of core principles in mind [15]. First, the MINIX operating system incorporates *isolation* of components. This means MINIX is split up in separate components which cannot influence each other in case of malfunction. Second, components in MINIX receive only the *least privilege* required to do their job, reducing damage when they behave inappropriately. Third, MINIX is a *fault tolerant* [16] operating system which can recover from crashing components. In MINIX a software bug in operating system components is not necessarily fatal nor can it bring down the entire system. Moreover components in MINIX can be selected for *dynamic update*, which allows performing updates to components without system downtime. Finally, MINIX aims to provide *standard compliance* to users, including the well known POSIX [17] software interface.

MINIX 3 is designed as a microkernel based operating system. The majority of operating system functionality such as device drivers, file systems and memory management are implemented in server processes. Servers are isolated, unprivileged processes with their own private memory and minimal (hardware) resources, which means they cannot in any way get access to resources which they do not own. In order to implement a fully working operating system, server processes can send messages to each other to request a particular service.

As shown in Figure 2.2, MINIX is structured in four different layers. The first layer is for regular processes created for programs ran by the user. These user processes may send messages to the server processes in the second layer below it for invoking operating system services, such as reading or writing files, opening network connections or submitting content to the printer. Under the server processes in the third layer are device drivers, which are responsible for interfacing with hardware devices. The lowest software layer in MINIX is the kernel. The kernel is the only privileged component in MINIX with unprotected access to all resources in the system including memory, and offers three major functions. First, the kernel is responsible for scheduling processes for execution on the CPU. This involves keeping track of all (in)active processes in the system and periodically switching processes to let each run a fair amount of time. Second, the kernel deals with direct input, output and interrupt handling of hardware devices. Since the driver processes in MINIX are unprivileged they cannot directly read and write data from devices. Therefore the kernel performs the I/O on their behalf if they have permission to do so. Finally, the kernel offers inter process communication primitives by exporting an interface which allows processes to send and receive messages.

Processes in a layer in MINIX can only send messages to processes directly in the layer below it. For example, user applications can send messages to all server processes, but never to device drivers. This restriction ensures that user applications cannot bypass security checks in the servers, such as file permissions, by directly sending messages to the drivers. Additionally, user applications and drivers are untrusted in MINIX. They may malfunction at any time, such as crashing, deadlocking, returning bad messages and more. The other trusted layers, servers and the kernel, do not suffer from failing applications and drivers in MINIX.

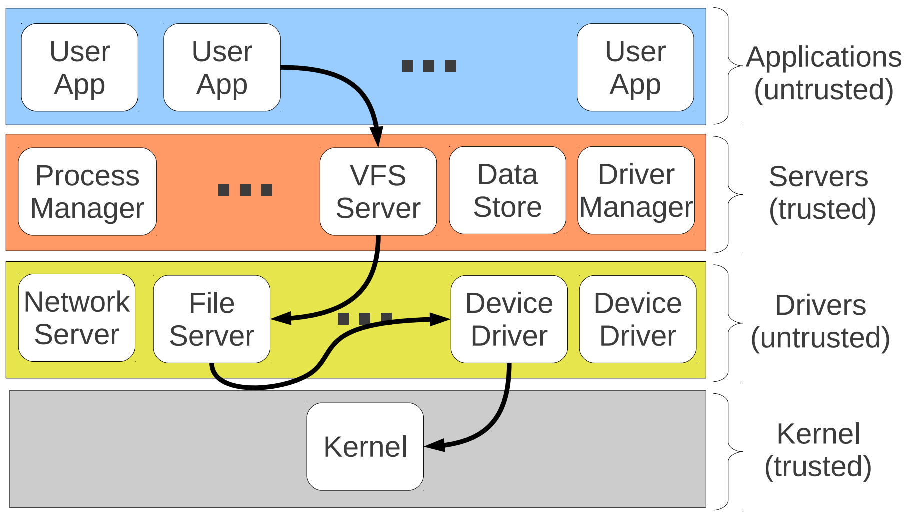


Figure 2.2: Overview of the MINIX 3 architecture.

2.2.1 Booting

In our work we needed to completely rewrite the MINIX startup procedure for running on the SCC. In this Section we briefly discuss how MINIX boots on a standard PC. Booting an operating system on a PC begins as soon as the PC is turned on. Special software called a bootloader is responsible for loading the operating system kernel from selected storage device, typically a local disk, CD-ROM or floppy, into memory. Once execution of a monolithic kernel begins, additional (user) programs can be loaded from disk using the internal disk driver inside the kernel.

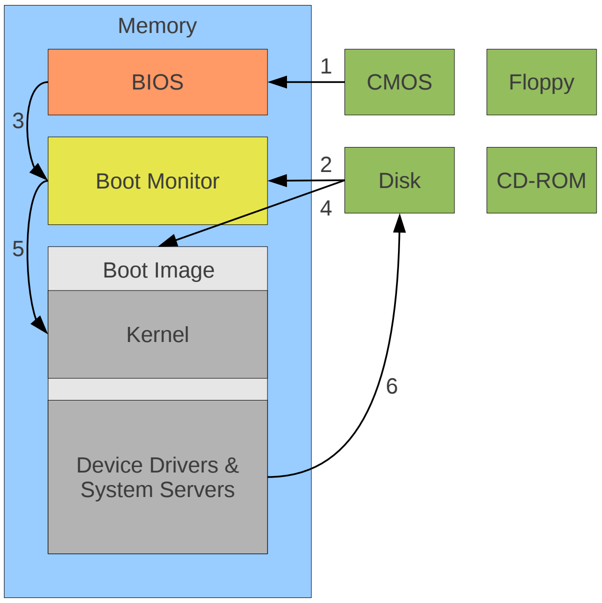


Figure 2.3: Booting MINIX on an standard Intel PC

Figure 2.3 illustrates the the steps needed to fully start MINIX. The first steps begins as soon as power is turned on. A special program called the Basic Input and Output System (BIOS) is the first code executed by the CPU. The BIOS is normally stored on the CMOS, a very low power static memory component. From the BIOS the user can choose which storage device to boot from. In the regular case this is a local disk, but CD-ROM, floppy or even network boot are often supported by the BIOS. Once the BIOS has chosen a storage device, either by default or by manual user override, it loads the first 512 bytes (the Master Boot Record) from the device into memory and continues execution there.

In the case of MINIX, the MBR contains a small program which in turn loads another bigger program called the *boot monitor* from disk into memory. The boot monitor is responsible for presenting the user with an environment for loading MINIX kernels. It currently gives the user a text console prompt in which the user can manually enter commands. For example, the user can select a specific MINIX image to be loaded and pass parameters for the boot sequence.

Since device drivers and file systems are separate programs in MINIX, loading only the kernel into memory is not enough to start the system. The MINIX kernel does not know anything about any devices including disks. Therefore the kernel alone cannot load more servers and drivers. To solve this problem, the boot monitor additionally needs to load a *boot image* from disk in memory, and pass its address to the kernel. As shown in Figure 2.4, the boot image contains all the required device drivers and servers to bring the operating system fully online. On startup, the kernel locates the boot image in memory and creates process entries for each of them in its internal process table

Component	**Description**
kernel	Kernel Code and Data
pm	Process Manager
vfs	Virtual File System
rs	Reincarnation Server
memory	Memory Driver (RAM disk)
atwini	ATA Disk/CD-ROM Driver
floppy	Floppy Driver
log	Log Device Driver
tty	Terminal Device Driver
ds	Data Store Server
mfs	MINIX File System Server
vm	Virtual Memory Server
pfs	Pipe File System Server
sched	Userspace Scheduling Server
init	Init Process

Figure 2.4: Programs contained in the MINIX 3 boot image

so they can be scheduled. Once the basic system is running, additional device drivers and servers can be loaded from the local disk, such as the network and audio drivers.

The MINIX kernel expects to be loaded in a specific state by the boot monitor, before it can be executed. First, it is assumed that the boot monitor loads the components from the boot image into memory such that they do not overlap. Second, the components must be loaded *page aligned*, which is 4K on Intel machines. This is important when later on the Virtual Memory server wants to turn on *virtual memory* [18]. Loading the components at page aligned addresses ensures that when paging is turned on, the virtual memory pages point to the correct memory sections and not partly overlap with others. Third, the boot monitor must initialize the component's memory. Every process in MINIX has three different types of memory associated with it: code, data and heap. The code section contains the program's machine instructions for execution by the CPU. The data section has static data, such as integer constants and character strings. Finally, the heap section is used by the program for storing the values of dynamic variables, such as counters or other internal state variables. It is assumed that the heap section initially contains only zero bytes. Therefore the boot monitor must allocate enough space for the heap section when loading the component into memory and make sure to zero all bytes in the heap. Last but not least the boot monitor must pass all relevant information needed to the kernel, including the addresses of the boot image components in memory and any configured boot monitor parameters.

2.2.2 Scheduling

The kernel is the lowest level software in the MINIX operating system and runs with all privileges enabled. One of the core tasks of the kernel is to schedule processes. On a basic PC there are only one or two CPUs in the system. This means that only one process can run at a time on a CPU. Since sometimes user programs can take quite long to finish, it is often desired to pre-empt a running process such that other processes can get a chance to run on the CPU as well. This principle is called *multitasking* or *multiprogramming*. Therefore the kernel uses clock hardware to context switch between processes whenever it receives an interrupt from the clock. In most Intel systems there is a clock device called the *Programmable Interval Timer* (PIT) [19] which can generate a hardware interrupt at regular intervals. The MINIX kernel uses the PIT for scheduling processes on Intel systems. By default the PIT is configured to generate an interrupt 60 times a second in MINIX.

Since version 3.1.8, MINIX supports *userspace scheduling* [20]. This means that it is possible for a special userspace scheduling server to make the scheduling decisions on behalf of the kernel. This reduces the functionality needed in the kernel itself even further and allows changing scheduling policies dynamically by sending a message to the scheduling server. On the other hand a constant overhead will be added since the scheduling server needs to run to be able to make the scheduling decisions.

2.2.3 Synchronous IPC

The majority of functionality in the MINIX operating system is implemented in user space processes which communicate by *message passing.* Figure 2.5 shows the structure of messages in MINIX. First, messages contain a source field indicating from which process the message came. Second, every message has a type field. The type field distinguishes different types of messages and can be filled arbitrarily by processes. Finally, every message has a payload with a maximum of 28 bytes. In total a MINIX message is 36 bytes.

```
typedef struct {
  endpoint_t m_source;   /* who sent the message */
  int m_type;            /* what kind of message is it */
  union {
        mess_1 m_m1;     /* message payload types are a set of integers and  */
        mess_2 m_m2;     /* character (pointers) with a maximum of 28 bytes. */
        mess_3 m_m3;
        mess_4 m_m4;
        mess_5 m_m5;
        mess_7 m_m7;
        mess_8 m_m8;
        mess_6 m_m6;
        mess_9 m_m9;
  } m_u;
} message;
```

Figure 2.5: The MINIX message format

MINIX offers several synchronous message passing primitives to deliver messages. After calling any of the synchronous primitives, the message has either been successfully delivered or an error occured. MINIX supports the following synchronous message passing primitives, which are implemented as kernel calls:

- **int send(endpoint_t dest, message *m_ptr)**
  Sends the message at *m_ptr* to the destination process *dest.* This function will block until either the message is delivered to the destination process or an error has occurred.

- **int receive(endpoint_t src, message *m_ptr, int *status_ptr)**
  Receive a message in *m_ptr* from source process *src.* The source might be a specific process or the special *ANY* keyword to indicate any source. Additionally, the *status_ptr* variable may contain optional status flags about the message. This function blocks until either a message has arrived or an error has occurred.

- **int sendrec(endpoint_t src_dest, message *m_ptr)** Sends the message at *m_ptr* to the destination process *src_dest* and in the same call receives a message from the destination in the *m_ptr* variable. This function is both a send and receive in the same call and will block until it has both send and received a message from the destination or until an error has occurred.

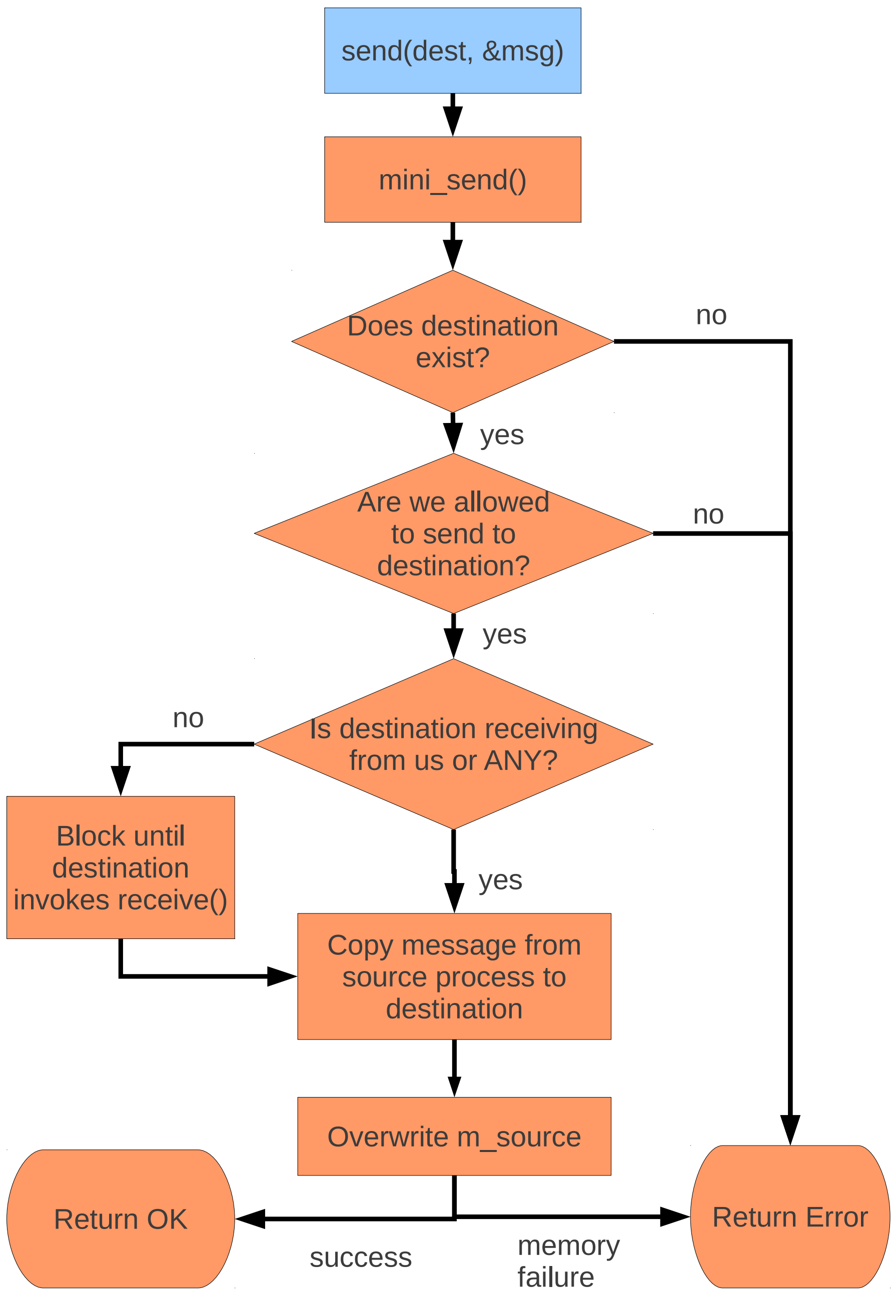

send(dest, &msg)
mini_send()
Does destination exist?
no
yes
Are we allowed to send to destination?
no
yes
Is destination receiving from us or ANY?
no
Block until destination invokes receive()
yes
Copy message from source process to destination
Overwrite m_source
Return OK
success
memory failure
Return Error

Figure 2.6: Flowchart of sending a message in MINIX

All of the message passing primitives in MINIX invoke the kernel, which is responsible for delivering messages at the correct destination. Calling synchronous message passing primitives may change the scheduling flags of a process. For example, when a process invokes send() but the destination process receives() only from another specific process, the sending process will not be scheduled until the message is delivered. User applications in MINIX can only invoke sendrec(). This ensures that user applications cannot create IPC deadlocks. If a user application would invoke send() but never receive(), the server process cannot deliver a reply message to the application.

Figure 2.6 illustrates the basic flow of execution when sending a message in MINIX. The process first invokes the send() function with the destination and message as arguments. Then the send() function will perform a kernel trap operation, which changes the execution context to the kernel. The kernel has a handler function *mini_send()* which is called when a process wishes to send a message to another process. The kernel is responsible for doing sanity checks on the requested destination process, such as whether it actually exists and if the calling process is allowed to send a message to it. For example, in MINIX there are various device drivers which take care of interaction with hardware devices, such as the disk driver. A regular user process should not be able to send messages to the disk driver as otherwise it could directly read data from the disk and avoid security checks in the file system server. Therefore, each process in MINIX has subset of allowed IPC destinations to which it is allowed to send messages to. Next, the mini_send() function can only deliver a message to the destination if it is actually waiting for a new message. If the destination did not call *receive()*, the calling process is flagged unschedulable to block it from executing. Later on when the destination process calls receive() the kernel can continue with delivering the message. At that point the last step is to copy the message from the calling process's memory to the destination process's memory. Additionally, the kernel overwrites the source field of the message, to prevent forgery. If no memory errors occur, the message is considered successfully delivered.

2.2.4 Asynchronous IPC

In addition to the synchronous message passing primitives, MINIX also offers asynchronous message passing functionality to processes. The asynchronous primitives are neccessary in some cases to avoid deadlock scenarios. For example, when two processes occasionally need to send a message to each other they would deadlock if both processes try to send a message at the same time. By using asynchronous message passing the sending call does not block and thus the deadlock can be avoided. The MINIX kernel offers the following asynchronous, non blocking message passing primitives to processes:

- **int sendnb(endpoint_t dest, message *m_ptr)**
  Attempt to send the message *m_ptr* to destination process *endpoint_t*. In contrast to send(), this function always returns immediately. If the destination process is unable to receive the message the function returns with an error.

- **int senda(asynmsg_t *table, size_t count)**
  Attempt to send one or more asynchronous messages in *table* with at least *count* entries. The kernel attempts to deliver these messages while the application can keep running.

- **int notify(endpoint_t dest)** Notifies the destination process *dest*. The destination process will receive an empty message indicating it has been notified.

The sendnb() function is useful in cases where the destination process is untrusted and may not have called receive() yet due to a crash or deadlock. Sendnb() only delivers the message if the other process is receiving and never blocks. In case the message cannot be delivered the caller must either temporarily store the message and try again later or discart it.

In some cases messages temporarily storing messages makes the software too complex and discarting messages is not an option. The senda() primitive is an alternative to sendnb() which solves these issues. It allows programs to submit a message table to the kernel containing messages for transit. As shown in Figure 2.7, while MINIX is running the kernel will occasionally peek in the asynchronous messaging tables during execution and attempt to deliver.

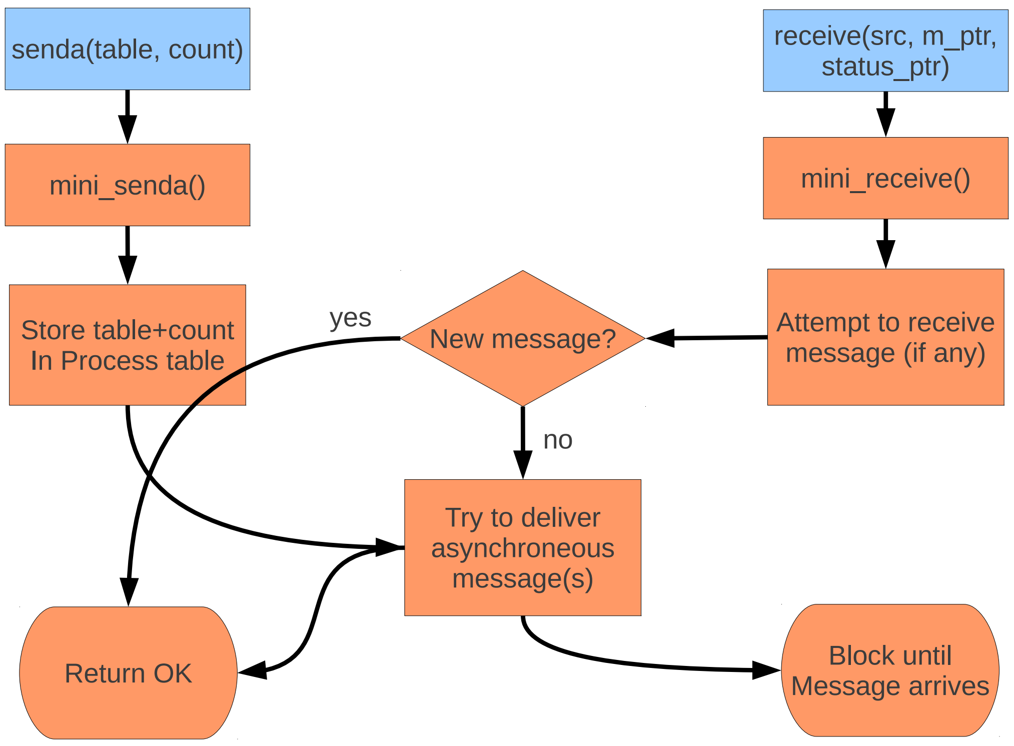

senda(table, count)
mini_senda()
Store table+count In Process table
receive(src, m_ptr, status_ptr)
mini_receive()
Attempt to receive message (if any)
New message?
yes
no
Try to deliver asynchroneous message(s)
Return OK
Block until Message arrives

Figure 2.7: Flowchart of sending an asynchronous message in MINIX

One disadvantage of senda() is that the messages table may become full. Fortunately user applications do not have to manage the asynchronous messages table themselves. Instead a library function called *asynsend()* takes care of managing the slots in the asynchronous messaging table and calling the senda() function appropriately:

- **int asynsend(endpoint_t dst, message *msg)**
  Sends the message *msg* to destination process *dst* without blocking. Internally this function calls senda(). This function returns OK if the message is being delivered or an error code otherwise.

MINIX provides yet another asynchronous messaging primitive which is notify(). This function serves as a way for processes to send interrupt signals to each other. For example, when one or more special events occur in a driver process and it wishes to inform a server process, it can use notify(). In contrast to the other primitives, notify() has the same semantics as hardware interrupts. This means that whenever a driver process calls notify() multiple times on the same destination process, but it has not yet processed any of them yet, only one notify message will eventually arrive at the destination.

2.2.5 IPC Endpoints

Processes in MINIX are able to communicate by sending and receiving messages to and from *endpoints*. An endpoint is a 32-bit unique number selected by the kernel. Every process in MINIX has a single endpoint number, which never changes. The MINIX kernel chooses a new endpoint number when creating a new process. This ensures that whenever a process crashes and is restarted, it receives a different endpoint number. That way other processes which communicate with the recently crashed process are aware of the crash since the old endpoint no longer works.

Endpoints in MINIX contain two types of information. First, endpoints store the *Process Slot*. The MINIX kernel keeps track of processes using its process table and the slot of a process describes the index of the corresponding entry in the process table. Second, an endpoint has a *generation number*. The generation number of a process begins with zero and is incremented by one every time it forks. Therefore, when a process is recreated with fork() after a crash, it receives a different endpoint from the kernel. Generation numbers also makes sure that when a process forks multiple times, each child will receive a different endpoint every time, even if the previous child has died already. The following formula calculates an endpoint number:

*Endpoint = (Generation * Max Generation) + Slot*

The generation size is the maximum number a generation can be and is higher than the maximum number of processes. With an endpoint number it is easy to calculate the corresponding process slot by dividing it modulus the generation size:

Slot = ((Endpoint + Max Processes) % Max Generation) - Max Processes

As discussed in Section 2.2.1, MINIX is started using a boot image. Every process in the boot image has a static endpoint number. Having a well known endpoint number for components in the boot image makes bootstrapping MINIX easier. There are also a few other special endpoint numbers in MINIX. First, there is the ANY endpoint which indicates 'any process' for use in the receive() IPC primitive. Second, the special endpoint NONE indicates 'no process at all'. This special endpoint can be used to initialize internal structures containing endpoint number(s) with a default value. Finally, a process can use the special endpoint SELF to indicate itself.

2.2.6 Input, Output and Interrupts

Since the kernel is the only component in MINIX which runs with all privileges enabled, it is also the only component which can interact directly with hardware devices. In Intel based systems, hardware devices can present operating system programmers with two ways for input and output: I/O ports and memory mapped I/O. An I/O port is a 16-bit number and can be assigned to only one hardware device. Programmers can read and write from I/O ports using special CPU instructions. For example, INB and OUTB are instructions for reading and writing a single byte from and to an I/O port, respectively. These instructions can only be executed in privileged kernel mode in Intel based systems.

As shown in Figure 2.8, MINIX device drivers need to ask the kernel to do any I/O port operation on their behalf. The kernel will verify any attempt to do I/O port operations such that unauthorized user processes cannot interact with the hardware this way. In Intel based systems, hardware devices can also present programmers with a special memory area for doing I/O operations. If a device driver in MINIX needs to access such memory mapped I/O region, it can ask the PCI server for permission to map the I/O region in its process address space. Occasionally some hardware devices need to interrupt the CPU when a special event occurs. For example, the PS2 controller on the keyboard needs to interrupt the CPU when a key is pressed so it can be captured and processed by the terminal driver. On most architectures including Intel PCs, if an interrupt is triggered the CPU will enter a special code section in kernel mode called the *interrupt handler*. As illustrated by Figure 2.8, the MINIX kernel constructs a new message and delivers it to the corresponding driver process when it receives a notify from a hardware device. Once the driver process receives the message generated by the interrupt, it can use I/O ports or memory mapped I/O to see what has changed for the hardware device. Intel based systems have a special *Programmable Interrupt Controller* (PIC) [21] hardware component which regulates delivering interrupts to the CPU. The PIC configures from which hardware devices the operating system wants to receive interrupts. When a driver is started in MINIX, it may need to enable receiving interrupts and the kernel configures the PIC on its behalf.

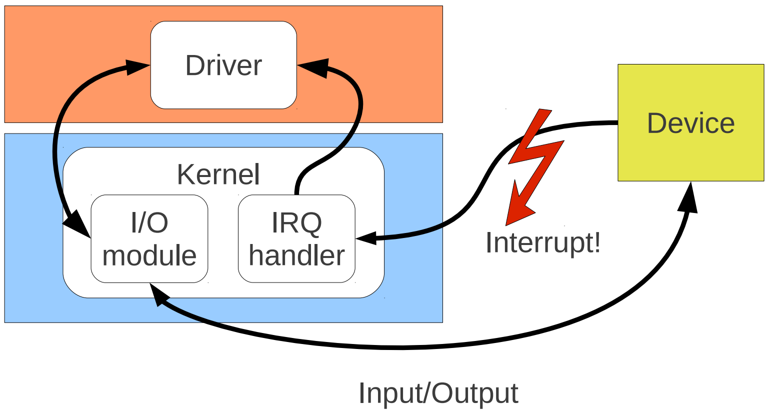


Figure 2.8: Handling input, output and interrupt requests in the MINIX kernel.

2.2.7 Servers

Servers are the basic building block of the MINIX operating system. All servers have a clear purpose. Servers are trusted system components which means it is assumed they do not show any malicious behaviour. Additionally, servers are the only type of processes which can send messages to device drivers. Together, these servers form the core of the MINIX operating system:

Reincarnation Server (RS) RS is a special component in MINIX. RS the root process and is responsible for the reliability of the entire operating system. If a driver crashes, for example because of memory corruption errors, RS receives a signal message from the kernel. Then RS typically attempts to restart the driver in the hope it can recover from the error. Additionally, the RS server sends a keep alive message to every component at regular time intervals. RS expects processes to always send a reply back, indicating it is still functioning correctly. If RS does not receive a reply back, the component likely is in a deadlock, infinite loop or other errorneous state. In that case RS also restarts the faulty component. Of course, if RS itself crashes or there is a fatal hardware failure such as power loss, MINIX cannot recover from the error. Finally, since RS is the root process, it is also responsible for starting all other servers.

Datastore Server (DS) There is another system server process which is involved in the reliability of the MINIX operating system. DS provides a persistent storage of server state in memory. When a component is running it can, from time to time, save the values of important variables in the DS server's memory. Should the component crash and be restarted by RS, it can retrieve the previous values of the stored variable at the DS server. The DS server supports saving several types of data using a unique key, including integers, character strings and binary data and allows lookup of variables using the key. Additionally, the DS server also functions as a naming server in MINIX. When RS creates a new server, it publishes the endpoint number of the new server to DS using the server name as the key. When another process needs to send a message to the new server, it can ask DS for the endpoint using the name of the server.

Virtual Memory server (VM) VM is responsible for managing both virtual and physical memory mappings. It keeps a list of available and allocated memory pages and is the only process in MINIX which has access to all system memory. Whenever another component in MINIX needs more memory or access to memory mapped I/O, it must send a message to VM.

Process Management server (PM) PM is responsible for creating, destroying and managing processes in MINIX. Just like the kernel, PM has an internal process table. PM implements functionality for POSIX [17] compliance in MINIX. PM works closely together with the VFS system server, especially when creating a new process. When creating processes the PM server needs the help of VFS to obtain the program executable code.

Virtual Filesystem Server (VFS) Another core function of an operating system is to provide an abstraction for managing data storage. This is usually done by file systems. In MINIX, VFS is responsible for providing a unified interface to all mounted file systems in the system. It acts as a man in the middle between user processes and file system server implementations, such as the Journaled MINIX File System (JMFS) [22] and the Extended 2 file system (EXT2FS). User processes can only send IPC messages to VFS and never directly interact with file system servers nor (storage) device drivers.

Peripheral Component Interconnect Server (PCI) The PCI server allows device drivers to access devices on the PCI bus [23]. Device drivers can send a message to the PCI server to claim device resources, such that it receives permission for I/O.

Internet Network Server (INET) MINIX supports a wide range of standard networking protocols, including TCP/IP [24] and Ethernet [25]. The component in MINIX responsible for the implementation of such network protocols is INET. It keeps track of all established TCP/IP connections, open TCP and UDP ports and knows about all network device drivers in the system. Whenever a regular user process needs to send data over a network connection, the INET server will generate the correct network packets and sends IPC messages to the network device drivers to transmit and receive the appropriate packets.

2.2.8 Drivers

Another type of processes in MINIX are device drivers. Unlike servers, device drivers are untrusted. In MINIX there are no assumptions about the current state or implementation quality of device drivers. If a driver crashes, becomes unresponsive or returns incorrect messages, it never significantly harms the rest of the system. Drivers in MINIX are also stateless. This means all important state information, such as opened files and network connections, are kept at the servers. For example, the disk driver provides functions for reading and writing disk blocks only. It does not know anything about files. In this case, VFS is the only process which carries state information about files. Some important device drivers, which were also needed or modified in the implementation of MINIX on the SCC, include:

Terminal Driver (TTY) TTY is responsible for the operation of the system console. It has support for three different types of system consoles:

Keyboard/Screen The most standard system console is a keyboard and screen display. The user types commands on the keyboard and result output is displayed on the screen. Currently the TTY driver supports AT/PS2 keyboards, dozens of different keymappings and standard VGA display screens.

Serial The TTY driver supports a serial port as the system console for debugging purposes. In such scenario users typically have two systems connected by a serial cable. One system runs MINIX and the other system has another operating system with an application running to read data from the serial port.

Pseudo When connecting to the OpenSSH [26] server on MINIX, each connection receives a special pseudo terminal. Pseudo terminals are not directly attached to a real hardware device, but in the case of the OpenSSH server to a TCP/IP connection managed by INET.

Disk Driver The disk driver reads and writes disk blocks from and to the local disk(s). The disk driver contains no state information which makes recovering from a crash easier. The Virtual File System Server (VFS) and the MINIX File System Server (MFS) carry all the state information neccessary for handling concurrent requests from user processes. MINIX currently supports ATA and AHCI disks.

Memory Driver The memory device driver is used during the bootstrapping of MINIX to serve as an initial file system. It contains configuration files and programs needed to startup MINIX. Additionally, the memory device driver serves various special files in the */dev* directory, including:

/dev/ram In-memory file system, for temporary files.

/dev/mem Access to absolute memory addresses.

/dev/kmem Access to kernel virtual memory.

/dev/null Null device (data sink).

/dev/boot Boot device loaded from boot image.

/dev/zero Zero byte stream generator.

/dev/imgrd Boot image RAM disk.

Network Driver MINIX supports various types of network cards. Each type of network card is implemented in a network device driver. Like the disk driver it does not contain any state information, except for the status of the network hardware itself. It is the responsibility of the network device driver to send and receive network packets. The Networking Server maintains all state information about the network, such as TCP/IP connections and open TCP or UDP ports. Currently MINIX supports a wide range of different network card families, including gigabit (Intel Pro/1000 [27], Realtek 8169 and Broadcom [28]) and wireless network adapters (Prism II based).

2.2.9 SMP

The latest version of MINIX implements support for SMP. Multicore systems allow processes to run concurrently in MINIX. Due to the design of MINIX, context switching between processes can become a possible performance bottleneck on unicore systems. Multicore systems can potentially mitigate this effect. Since server processes can run on separate CPUs in a multicore environment, context switching overhead can be spread among the cores to reduce its effect. Moreover, if the number of CPUs is greater than the number of processes in the system, context switching can be avoided all together. Additionally, server processes in MINIX share little (if any) memory. Therefore locking overhead is minimal.

Currently the SMP implementation of MINIX is only different from standard MINIX in the kernel. In short, the most important changes introduced to the kernel are:

- **Booting**: When booting MINIX, only the first CPU runs. In the low level startup code of MINIX, the boot CPU performs Intel's SMP boot protocol [29] to bring all other CPUs online.
- **Multicore Scheduler**: The scheduling code is changed such that every CPU receives processes to run and that a process is scheduled only for one CPU at a time.
- **Locking**: Currently the implementation has a single big kernel lock which avoids race conditions when writing to shared data, such as the process table.
- **APIC support**:Intel SMP systems use a different interrupt controller called the *Advanced Programmable Interrupt Controller* [30]. The APIC supports all basic functionality of the older PIC and also offers *Inter Processor Interrupts* (IPI).
- **IOAPIC support**:Another extra hardware component called the *Input and Output Advanced Programmable Interrupt Controller* (IOAPIC) [31] deals with distributing external interrupts from hardware devices to a configurable subset of CPUs.

2.3 Single Chip Cloud Computer

The *Single Chip Cloud Computer* (SCC) is a research project by Intel Corporation. The SCC project is introduced mainly for two reasons [32]. To promote manycore processor [33] and parallel programming research [34]. As a result, several researchers are developing [35, 36] or modifying applications [37, 38] and operating systems [39, 40] to run efficiently on the SCC. In addition to increasing scalability [41, 42, 43], reducing power consumption [44, 45] is an important subgoal of the SCC project. Eventually Intel aims to bring terascale computing [46] to single die computer systems. "Tera" means 1 trillion, or 1,000,000,000,000. Intel envisions to create platforms capable of performing trillions of calculations per second (teraflops) on trillions of bytes of data (terabytes). The SCC hardware prototype features 48 cores on a single die, which according to Intel has never been done before [47]. It is designed to scale up to even hundreds or more of cores on a single die. With the SCC, Intel is encouraging researchers to explore the possibilities of single die manycore computer systems.

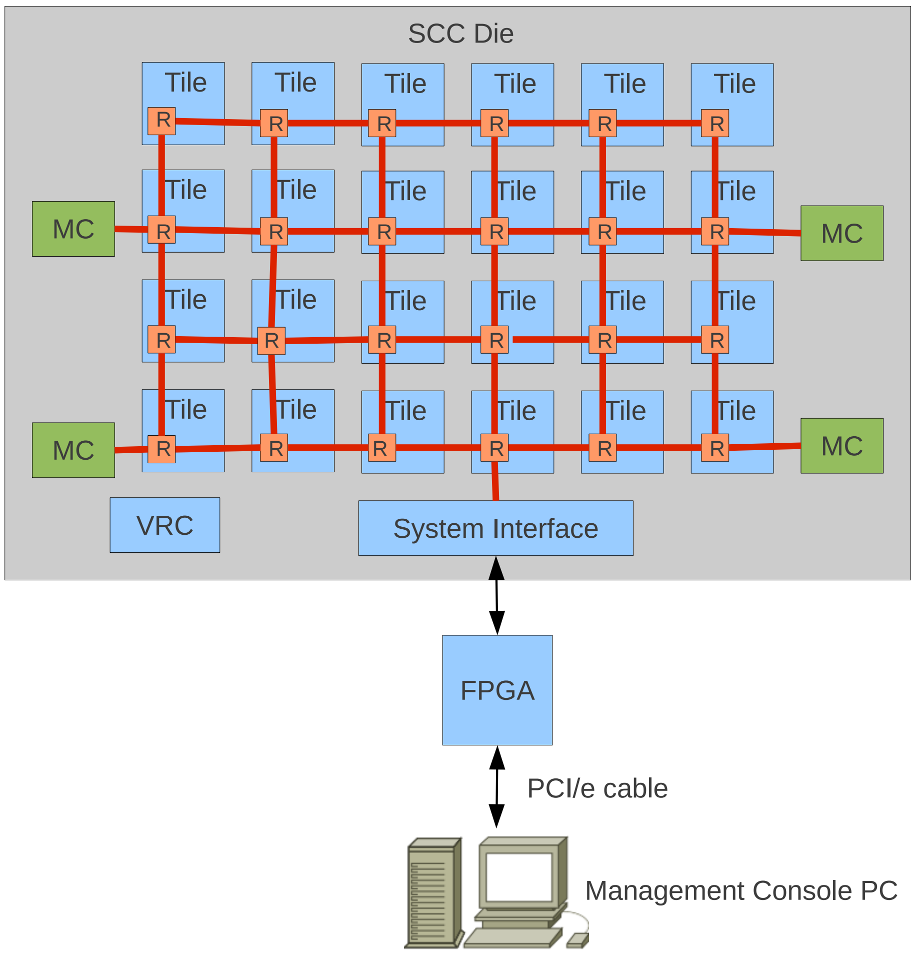


Figure 2.9: Architecture of the Single Chip Cloud Computer.

Figure 2.9 illustrates the architecture of the SCC. It has a total of 24 tiles containing two Intel Pentium (P54C) [48] processors. Intel has chosen the P54C because it has a relatively simple design and is much smaller compared to newer processors. This allowed the designers to put more cores on the die in the SCC prototype, also taking the available room and financial resources into account. Additionally, many existing applications can already or easily be modified to run on the P54C. The 24 tiles in the SCC are connected by an on-die high performance mesh network. As indicated by the "R" in Figure 2.9, every tile has a special routing hardware component called the *Mesh Interface Unit* (MIU) for transferring and receiving packets on the mesh. Tiles can send packets to other tiles for inter-processor communication or to one of the four memory controllers for reading and writing memory. Note that the current SCC does not have any extra hardware such as a keyboard, mouse, display screen or even a local disk. The only way to interact with the SCC is using a special PCI/e interface to the SCC's FPGA from a Management Control PC (MCPC). From the MCPC developers can load operating systems on the SCC, read and write all available memory, interact with the individual software emulated consoles running on each processor, control power management functions and more.

2.3.1 Tiles

The SCC consists of 24 identical tiles as presented in Figure 2.10, inter connected by an on-die 2D mesh network. In this section we describe each component on the tiles. First, every tile in the SCC contains two P54C processors. Intel released the P54C processor in october 1994 and could operate at frequencies between 75 to 120 MHz. Fortunately the P54Cs in the SCC are able to operate at much higher frequencies between 100 and 800 MHz.

Both P54Cs are equiped with two levels of caches: level 1 and level 2. The level 1 cache is internal on the P54C and is separated into instruction and data caches, each 16KB in size. Level 2 caches is are external components and have a capacity of 256KB. These caches have a cache line size of 32 bytes, giving a total of 512 individual cache lines. One fundemantal design choice of the SCC is that the caches are incoherent. In most standard multiprocessor PCs the caches of individual CPUs are coherent, meaning whenever a CPU changes a memory value which other CPUs have cached, they must all be notified. For only a small number of CPUs this is not yet a serious performance bottleneck, but when hundreds or thousands of CPUs are put on a single die it will be. Cache incoherent Therefore Intel decided to remove cache coherence in order to be able to scale up the number of CPUs on a single die even further. This means that whenever a CPU writes some data to memory, and thus also to its own cache, the other CPUs are not notified of the changed data. Therefore when other CPUs attempt to read the changed data, they may read stale values. This design choice is especially relevant for operating system programmers, as many general purpose operating systems including MINIX are designed for coherent caches in a multi processor environment. An important note about the caches for operating systems is that cache lines are only flushed back to memory when writing to the last bit of a cache line [49].

Additionally, both P54C processors share the *Message Passing Buffer* (MPB) memory component. The MPB aids applications and operating systems to implement message passing. Each CPU has a 8KB partition of MPB space, 16KB in total per tile. Access to the MPB of a CPU in the SCC is faster then accessing the DDR3 memory controllers. MPBs are not cached in the level 2 cache, but only in the level 1 cache. Of course when reading the MPBs, a CPU needs to clear the level 1 cache. Intel has introduced an extra instruction called *CL1INVMB*, which can be used to clear level 1 cache lines containing MPB data. To determine the kind of data in the level 1 cache a special MPBT bit has been added to the cache lines. Only cache lines from the MPB have the MPBT bit set and CL1INVMB clears only the cache lines with the MPBT bit set.

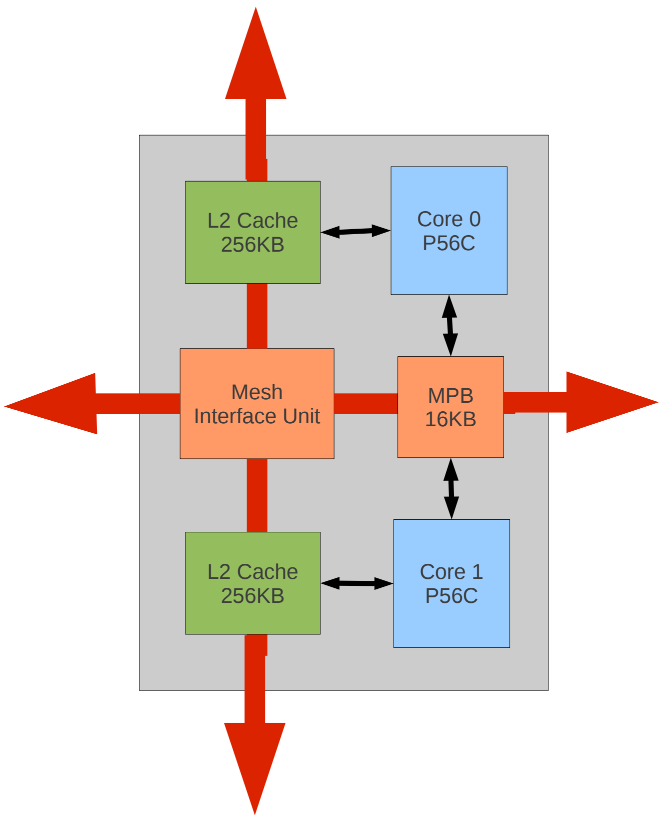


Figure 2.10: Tile architecture design of the SCC.

Each tile also has a *Mesh Interface Unit* (MIU) which is responsible for routing packets on the 2D mesh. According to Intel's documentation [50] the MIU contains hardware logic for: (de)packeting, command interpretation, address decoding, local configuration registers, link level flow control, credit management and an arbiter. Operating systems do not need to access the MIU directly.

Finally, every tile in the SCC contains configuration registers. Operating systems running on the SCC may want to change some configurable settings about the tile(s). It is possible for operating system designers to use the SCC configuration registers for this purpose. The configuration registers are accessible via memory mapped I/O. Each tile has its own set of configuration registers and is also available to all other tiles. There are a total of 12 different configuration registers as shown in Figure 2.11. The LUT0 and LUT1 registers contain the physical address of the LUTs (explained later in Section 2.3.2) of core 0 and 1, respectively. LOCK0 and LOCK1 is used for synchronizing CPUs when they need to access a shared resource, for example the MPB. MYTILEID is a read only configuration register and may be used by the operating system to obtain the local tile and core ID. GCBCFG is a special register for configuring parameters and settings about the file frequency and contains various reset flags. SENSOR and SENSORCTL are useful when using the on-chip thermal sensors for measuring the heat produced by the hardware. L2CFG0 and L2CFG1 are configuration registers of the level 2 caches for cores 0 and 1, respectively. Finally, GLCFG1 and GLCFG0 are mainly used for triggering *Inter Processor Interrupts* (IPI).

Register	Description	Access
LUT1	LUT register core 1	Read/Write
LUT0	LUT register core 0	Read/Write
LOCK1	Atomic Test and Set Lock Core 1	Read/Write
LOCK0	Atomic Test and Set Lock Core 0	Read/Write
MYTILEID	Tile ID	Read
GCBCFG	Global Clock Unit Configuration	Read/Write
SENSOR	Thermal Sensor Value	Read
SENSORCTL	Thermal Sensor Control	Read/Write
L2CFG1	L2 Cache Core 1 Configuration	Read/Write
L2CFG0	L2 Cache Core 0 Configuration	Read/Write
GLCFG1	Core 1 Configuration	Read/Write
GLCFG0	Core 0 Configuration	Read/Write

Figure 2.11: SCC configuration registers description, access and offset

2.3.2 Memory Architecture

The SCC is equiped with a total of four DDR3 memory controllers. Each memory controller can manage two DIMMs with each two ranks of capacities 1GB, 2GB and 4GB. This gives a maximum capacity of 16GB per memory controller, resulting in a total system wide capacity of 64GB. Memory controllers in the SCC can operate at frequencies of 800MHz up to 1066MHz.

Each CPU needs to access memory which is located at the memory controllers. Therefore there must be a way to tell which parts of memory at the controllers belongs to which CPU(s). Intel has come up with *Lookup Tables* (LUT) as a solution for this problem. LUTs perform a mapping of core addresses to system addresses. Every CPU has a private LUT which can map up to 4G of system memory, which can be either of the type DDR3, MPB or configuration registers. Each slot in the LUT can map 16MB of system memory data, which means there are a total of 256 LUT slots maximum.

Figure 2.12 illustrates the calculation of mapping of memory in the SCC using LUT's. When a program attempts to access a 32-bit core virtual address on a standard P54C, the address is first translated from a virtual address to a physical address using the core's *Memory Management Unit* (MMU). After this translation the result is a 32-bit core local physical address which needs another translation from the core address to the system address. The LUT translation takes the upper 8-bits from the core 32-bit address to find the correct LUT entry slot. The LUT entry contains the neccessary translation information to obtain the system address and routing information. *Destination ID* is the tile destination ID for routing on the 2D mesh, where *Sub Destination ID* specifies the type of memory for access: DDR3, MPB or configuration registers. *Bypass* is a 1-bit value which may be used for bypassing the MIU. Finally, the 46-bit system address value is obtained by adding the bypass bit, 3-bit subdestination ID, 8-bit tile ID, 10-bit address extention to the lower 24 bits of the 32-bit core physical address. Only the lower 34-bits of the 46-bit system address get send to the destination specified by the tile ID. See Appendix A for the default LUTs.

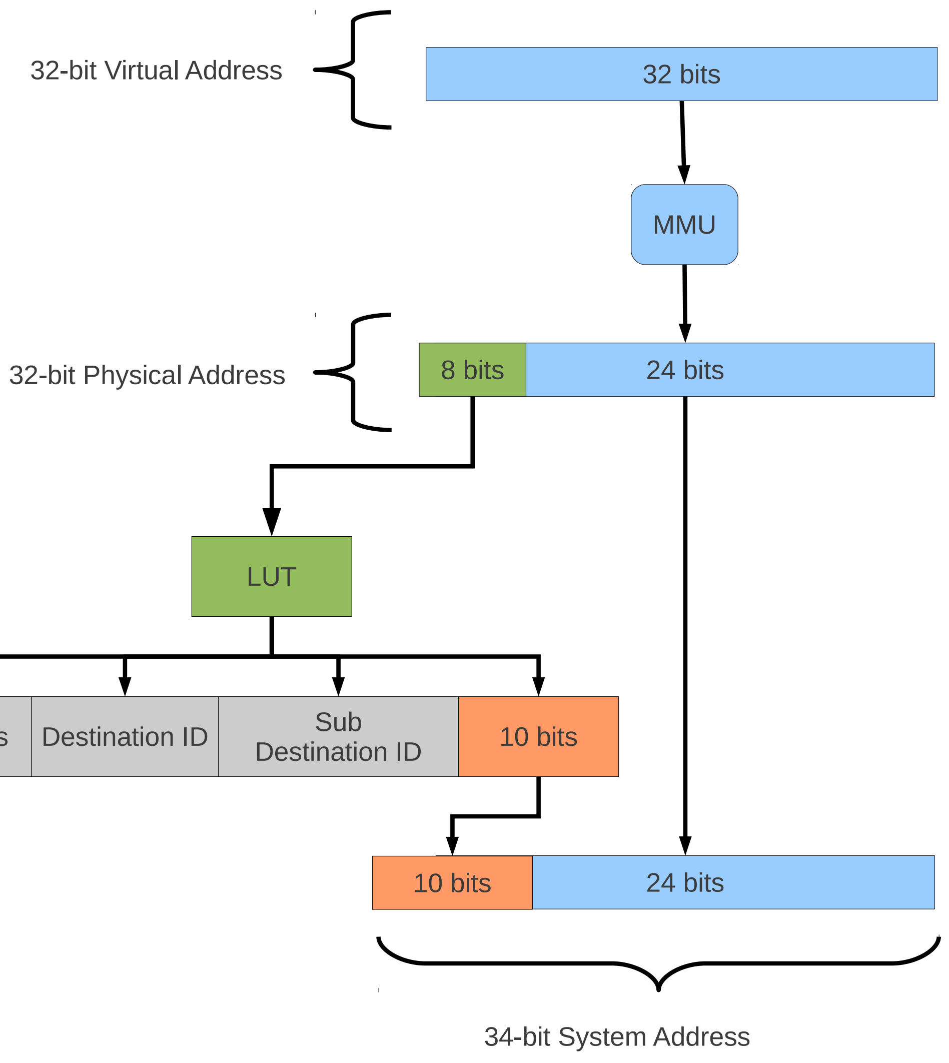


Figure 2.12: Lookup Tables map local core addresses to system memory addresses.

2.3.3 Management and Control PC

In order to be able to use the SCC, there must be a MCPC connected by a PCI/e cable to the SCC. The MCPC is used to power on the SCC, load programs on the CPUs and as the system console for operating systems running on the SCC. In addition to the PCI/e cable the MCPC should have the SCC Software Kit installed. This is a Linux software distribution containing programs for both command line and graphical management of the SCC hardware, including:

- **sccBmc**
  This application will either initialize the attached SCC Platform with one of default settings or execute one or more dedicated BMC commands (e.g. "status"). It is required to run this command with the -i option when power cycling the SCC in order to initialize the system. Without proper initialization the SCC will not function. BMC stands for Board Management Controller.
- **sccBoot**
  Starts an operating system on a selection of CPUs on the SCC. It is possible to boot the default Linux operating system or a custom image.
- **sccBootlogs** Outputs the boot messages of the operating systems started on the SCC. This command is specific to the Linux operating system implementation.
- **sccDisplay** Open the graphical virtual display of the selected CPUs on the SCC. The Linux implementation provides a graphical console which can be viewed on the MCPC using this command, including support for keyboard and mouse interaction.
- **sccDump** This application reads memory content (or memory mapped registers) from the SCC Platform. It is possible to address the memory controllers, the MPBs or the configuration registers. Addresses need to be 256-bit aligned as this software reads whole cachelines. The number of requested bytes may not exceed 1MB.
- **sccKonsole** Opens the text console to Linux operating system instances running on the selected CPUs. The console application connects to the Linux instances using an TCP/IP connection to the CPUs using an range of uncachable DDR3 memory as packet buffers.
- **sccMerge** Uses an input bootable image to generate boot image configurations for the individual DDR3 memory controllers. This command is mostly used internally by the SCC kit when booting operating systems on the SCC.
- **sccPerf** Starts the Linux specific performance measuring widget. This application reports the CPU load of the individual CPUs in the SCC.
- **sccPowercycle** Completely restarts the SCC by turning the power off and on. After doing a power cycle the sccBmc application must be used to re-initialize the SCC before it can be used again.
- **sccProductiontest** Performs a range of Linux specific tests on the SCC.
- **sccReset** This application can be used to reset the given subset of CPUs in the SCC. In most cases sccReset is used after booting a tryout bootable image to reset the CPUs to a known state. After a CPU is reset, it has stopped executing and will only begin executing again until either sccBoot or sccGui is used to start a new operating system.
- **sccTcpServer** This application provides remote socket based access to SCC using the sccKit interface.
- **sccWrite** This application can write to to system memory addresses of the SCC Platform. It is possible to address the memory controllers, the MPBs or the configuration registers. Addresses need to be 32-bit aligned and will be truncated otherwise. If it is required to write smaller chunks, the solution is to perform a read-modify-write.

- **sccGui** Starts the graphical interface for managing the SCC. It is possible to do anything using this application in GUI mode which is also possible using the command line programs, such as booting operating system images and reading or writing memory.

Chapter 3

Design

Our ultimate goal is to run MINIX as a single system image on the SCC. Unfortunately, we cannot use the MINIX SMP implementation for this purpose mainly due to its cache incoherent nature and lack of SCC specific hardware support. To implement the MINIX operating system on the SCC as a single system image, several major changes are needed to multiple components, including the kernel. This requires a careful design and in this chapter we present the design we choose for the MINIX SCC implementation.

3.1 Requirements

The overall design of the MINIX SCC implementation should be extendable, which means that future work can be done without re-doing previous work. Therefore we have attempted to think of which requirements the design should meet, in order to make future work as easy as possible:

- **Keep Core Principles**: Any changes to MINIX for the SCC should not remove any of the core principles, such as isolation, least privilege, fault tolerance, dynamic updating or standard compliance.
- **Cache Incoherence**: The design of MINIX should be changed such that it can run in a cache incoherent environment like the SCC.
- **New Boot Code**: Since the SCC lacks several hardware and software components from the PC, it requires a custom bootstrapping implementation.
- **Cross Core IPC**: Processes must be able to send messages to processes at other CPUs in the SCC. Therefore, we must extend the current IPC mechanisms to support global IPC. Currently processes in MINIX communicate using endpoint numbers. It is not our intention to remove the endpoints, but to change the design such that processes can address remote processes in addition to local processes. The global addressing design should allow any pair of processes to communicate, including all processes in the MINIX boot image. In addition to global addressing, there should be some mechanism by which processes can find remote processes.
- **Process Migration**: Eventually it should be possible to migrate processes on the SCC. This means moving processes from one CPU to another, for example to balance the CPU load. The IPC design should not become a problem when implementing this.
- **Bulk Memory Copy**: We need a fast way to copy large portions of memory between processors, for example to implement process migration and UNIX sockets. Due to the cache incoherency of the SCC, this is not trivial.

- **Global View**: Applications running on a core in the SCC should have the same view of resources, such as files, the process list, shared memory and network connections. This requires either a single instance of each server in the SCC or multiple instances which synchronize state, depending on performance implications. Additionally, we need to decide which processes are visible to other cores.
- **Parallelizing Servers**: Currently the server processes in MINIX work together in a sequential fashion. This means the server processes are often block waiting on each other. In order to gain higher performance on manycore systems, the servers must be parallelized such that they can handle requests concurrently.

Since all requirements cannot fit within the scope of one master thesis, we focus in our work on providing a bootable MINIX implementation for the SCC with an efficient cross core IPC mechanism. Additionally, we assume all processes require global IPC.

3.2 Cache Incoherence

The first challange for our design is to deal with the cache incoherency of the SCC. A recent design principle which allows running an operating system in cache incoherent environments is the *multikernel* [51]. In this design each CPU has their own local instance of the operating system kernel, with data and variables kept in private local memory only. The kernel instances do not share any datastructures via shared memory but instead use message passing to communicate any changes in the operating system state. Having multiple kernel instances with only local memory removes the requirement of shared cache coherent memory.

The multikernel design principle matches closely with the microkernel design. Since microkernels contain minimal functionality, they also contain minimal state which must be communicated to other kernels. The majority of state information in a microkernel based operating system is stored in server processes, which already use message passing to communicate state or invoke services.

Figure 3.1 shows how we applied the multikernel principle to MINIX. First, every CPU on the SCC runs an instance of the MINIX microkernel. The microkernels do not share any data, but use the MPBs to communicate changes to the internal state, such as the process table. Second, every CPU has all relevant MINIX servers and drivers including PM, VFS, RS and VM. User applications running on a particular CPU only need to send messages to local instances of the servers. However servers and drivers are able to to send messages to remote instances on other CPUs with help from the kernel. The kernel IPC code should be extended to use the MPBs for sending messages to remote processes.

3.3 Global Addressing

As described in Section 2.2.5, MINIX uses endpoints for addressing local processes. A single endpoint can point to any of the currently 1024 maximum processes in the MINIX operating system. When implementing MINIX on the SCC, processes should be able to do IPC with any other process running on any other core. Therefore, to provide global addressing we need to have global endpoints, in addition to the local endpoints. This requires changing the endpoints mechanism such that it is possible to point at remote processes. Additionally, the global endpoints solution should also satisfy the requirement for process migration. Global endpoints should not become a problem when migrating processes between CPUs. In the following Section 3.3.1, we present our solution for global endpoints. Additionally, from Section 3.3.2 and on we discuss other alternative designs we considered. In every Section we evaluate the advantages and disadvantages of the particular design.

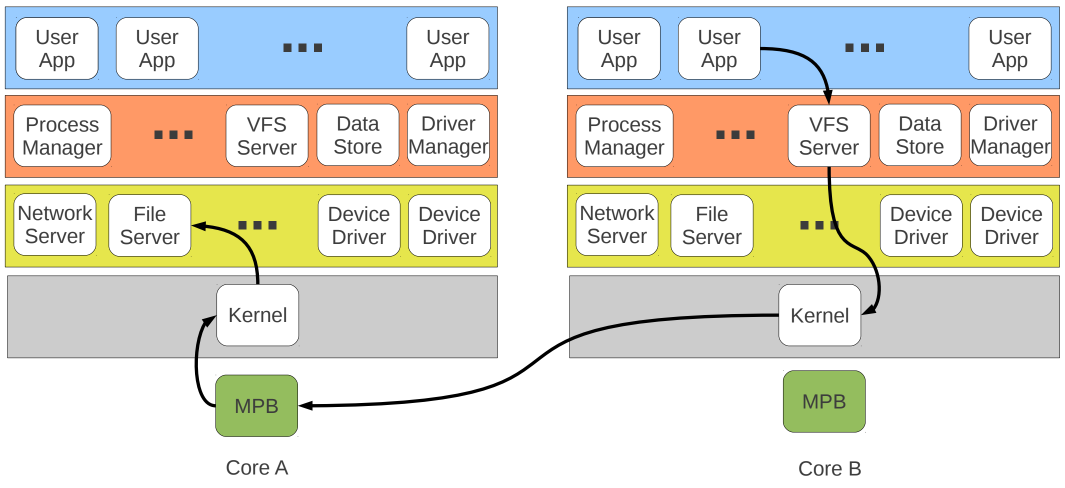


Figure 3.1: The multikernel design principle applied to MINIX.

3.3.1 Remote Process Stubs

Our solution for the global endpoints design is to store remote processes in the kernel process table with a special remote flag. We call such remote process entries *remote stubs*. Conveniently the kernel's *struct proc* already contains a field *p_cpu* from the SMP implementation for storing the CPU number. When implementing process migration, this field should be updated. Additionally, the *p_rts_flags* field may be used for the remote flag. This field contains flags which, if set, prevents a process from being scheduled. Of course process table entries for remote processes should never be schedulable, so the p_rts_flags field is perfect match.

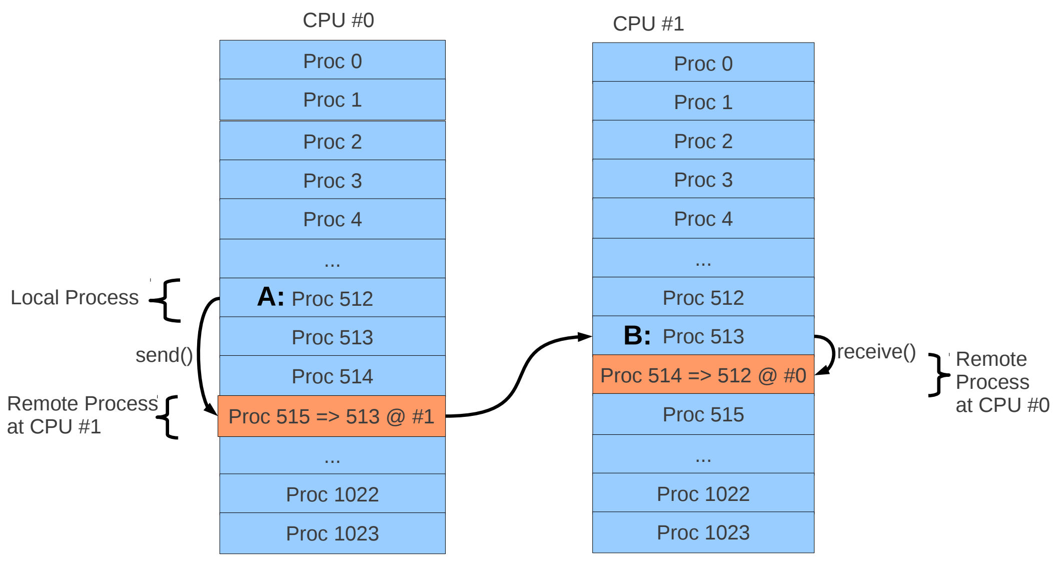


Figure 3.2: Set a special remote flag in the kernel process table to implement global endpoints.

Figure 3.2 illustrates how two processes A and B can communicate with this design. Process A is located on CPU 0 with local process slot 512, process B resides on CPU 1 with local process slot 513. When process A wishes to send a message to process B, it sends a message to the remote stub endpoint of process B at CPU 0, which has slot 515. When delivering the message to process B, CPU 1 can fill in the source endpoint with the remote stub endpoint of process A at CPU 1, which has slot 514. An advantage of this design is that processes can still use the endpoints for remote endpoints just like they would for local processes. Additionally, the design is relatively simple to implement with no large changes to the process table. Finally, a minor issue with this design is that the Process Manager currently chooses new process slot numbers. Since PM cannot look directly in the kernel process table, it could choose a slot which is already taken by a remote stub. The solution is to let the kernel choose slots.

3.3.2 Alternative: Encoding Endpoints

Another solution we came up with is to encode the CPU number inside the endpoints. As shown in Figure 3.3, the upper 8 bits of the endpoint number could be used to contain the destination CPU number. For endpoints pointing to remote processes running on another CPU, the upper 8 bits would be non zero and for local endpoints the upper 8-bits would be zero.

An advantage of this design is that it is very simple to implement. No drastic changes are needed in the kernel nor in any of the user processes. However, when a user process wants to do IPC with another process running on a different CPU, it must know in advance the destination CPU number and put it in the upper 8-bits of the endpoint. A much bigger problem with this design is that encoding CPU numbers in the endpoints makes process migration much harder, as the endpoints would need to be changed in all user processes on migration.

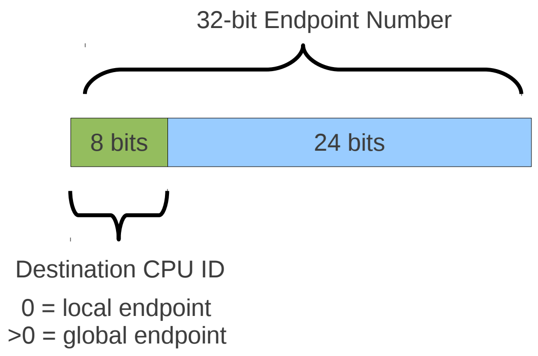


Figure 3.3: Encoding endpoints with the destination CPU number.

3.3.3 Alternative: Splitting Process Slot Address Space

Another way to design the global endpoints in MINIX on the SCC is to split the kernel process table in two halfs. One half contains local only processes from slot 0 until 1023 and the other half contains entries for (remote) processes ranging from slot 1024 until 2047. Any process which has a slot number in the lower half is not visible to other CPUs. Processes with a slot in the upper half are visible to all other CPUs and have the same slot number on all CPUs. The endpoints remain unchanged in this design. Any endpoints pointing to the lower half, the local only processes, still do local IPC. Endpoints pointing to processes in the upper half, thus processes which may run on other CPUs, should trigger sending a remote IPC message to another core if remote. Figure 3.4 illustrates the split process table design. Any remote processes in the upper half of the kernel process table should contain a flag indicating that its a remote process, a CPU number and must not be flagged schedulable.

Splitting the process table in a local-only and remote-only half solves some challenges when implementing MINIX on the SCC. For example, the endpoints are independent of the CPU numbers, which makes process migration much easier. Additionally, the endpoint design for remote processes is transparent to user processes. The endpoints pointing to remote processes look the same compared to endpoints of local processes from the perspective of user processes. However, one problem with this design is to decide when a process should receive such remotely visible slot. Should the parent process inform the kernel before, during of after creating the child process that it should receive such remote slot? In the case a process needs to do remote IPC in the future, it must know this in advance. Moreover any process in the MINIX boot image cannot do remote IPC with this design, as they have special static low numbered endpoints. Finally, this design requires synchronization when adding and removing process table entries, which may be a difficult and costly operation in a 48-core environment.

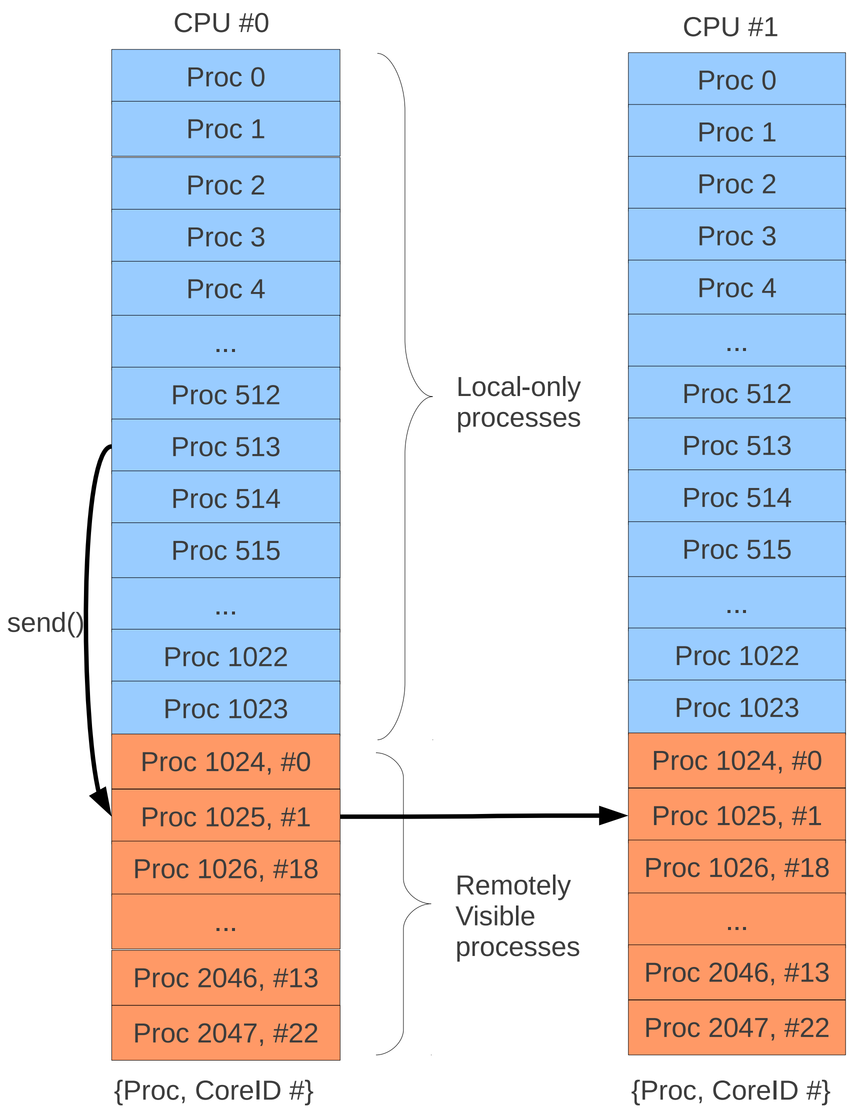


Figure 3.4: Splitting the kernel process tables to distinguish local/remote processes.

3.3.4 Alternative: Global Process Table

In an attempt to solve the problems with earlier designs, we came up with yet another global endpoints design. In order to let all processes in MINIX do remote IPC, including the boot image processes, we modified the design with another global process table separately from the local kernel process table. When a process wants to do IPC to another remote process, it would need to have its global endpoint which points to the entry of the destination process in the new global process table. Therefore any process which wishes to do remote IPC and be visible to all other processes would need to have a *Global Process Slot* in the global process table next to its local process slot. Of course any program from the boot image is also able to get such global slot. Delivering local IPC messages to local process endpoints still happens the same as in default MINIX.

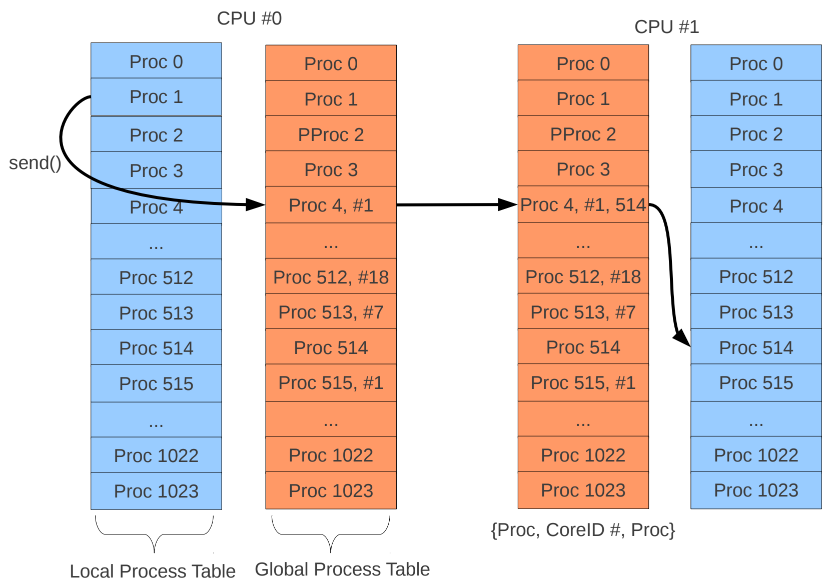


Figure 3.5: Assigning a Global Process Slot to solve the global endpoints problem.

Although this design solves some problems it also introduces several new ones. The design requires a clear distinction between local and remote endpoints. Therefore user processes must also use different IPC primitives when doing remote IPC or pass additional flags to the current IPC primitives indicating they mean global endpoints. Either way, the user processes in MINIX would need to be changed for using the global endpoints in this design. A more fundemental problem with this design is what would happen if a process wishes to receive from the ANY source. How would the kernel indicate to the process the message is from a local or remote endpoint? Finally, this design also requires synchronization when creating or removing entries from the new global process table.

3.3.5 Alternative: Userspace IPC Server

A more different design we looked at is having a dedicated userspace IPC server for remote messaging. Whenever a local process wishes to send a message to another remote process running on a different CPU, it would send a message to the IPC server with the payload message embedded. The IPC server would have a known static endpoint so that user processes can always find it. Then the IPC server should take care of delivering the message to another IPC server instance running on another CPU where it would finally be delivered to the destination.

Having a userspace IPC server deliver remote messages will cost additional context switching overhead. However, this is not the biggest problem of this design. The problem which this design does not solve is: how can user processes address other processes running on other cores? Since user processes only have the endpoint of the IPC server, there is no obvious way by which they can achieve this. On the other hand, it is possible to combine this alternative design with our solution discussed in Section 3.3.1, but it would not be helpful.

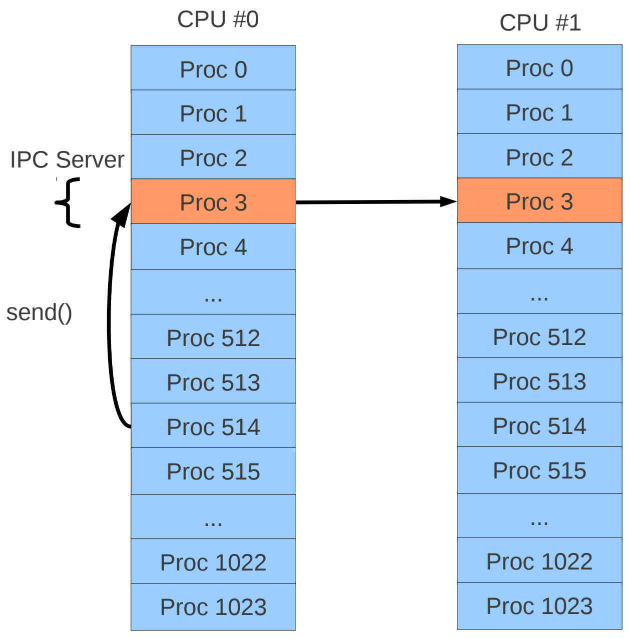


Figure 3.6: Use a dedicated userspace IPC server with a known endpoint for remote IPC.

3.4 Endpoint Discovery

Another challange for the design is to determine how processes can find global endpoints of a particular other process. Our solution is to publish global endpoints. As illustrated by Figure 3.7, the idea is that whenever a process wishes to do remote IPC, it should publish itself to the local DS server using a unique key. Then the local DS server would broadcast to all other DS server instances running on the other CPUs that it has a new process which is publishing itself. The remote DS server instances would then ask the kernel to install a *remote stub* for the publishing process in its process table. Then whenever a process wants to do IPC to another possibly remote process, it asks the DS server for the endpoint by giving it the unique keyname. The process then receives an endpoint number and can use any of the standard IPC primitives available in MINIX on it.

An advantage of this design is that the local DS server already functions as a local naming server for finding endpoints. The DS server code only needs to be extended to implement the broadcasting and installation of remote stubs. Note that this design does not require synchronization of kernel process tables. The remote stubs may have different local process ID entries without problem, as long as the local DS servers return the correct endpoint for the given unique keynames. Note that any process which wishes to do remote IPC must publish itself, including applications, since remote CPUs must be able to fill in the correct source endpoint of remote messages using a stub endpoint. On the other hand, it is possible in this design to restrict to which other CPUs to publish endpoints. Finally, since the endpoints remain unchanged for user processes, it is easier to implement process migration since only the kernel process tables need to be changed.

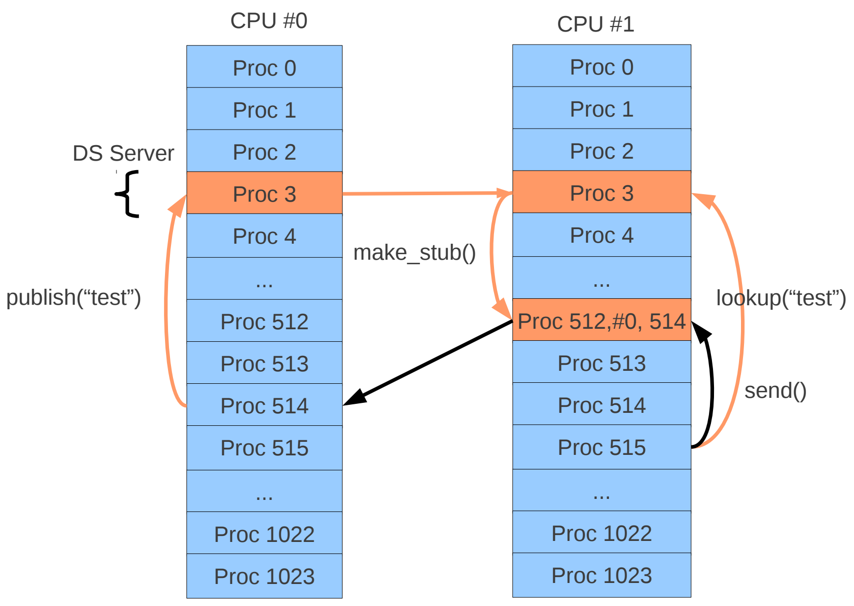


Figure 3.7: Publishing an endpoint to the DS server for remote IPC.

3.5 IPC Message passing

Another design decision for the implementation of MINIX on the SCC is about how the IPC message passing between the CPUs should work. The first choice is to decide where the intercore messages should be stored. As discussed in Section 2.3, the SCC has a special dedicated memory space called the MPB which has been designed with message passing in mind. Relevant to whether the MPB is useful for the MINIX implementation is to see how many messages can fit in the MPB. Since cache lines in the SCC are only written back to memory when touching the last bit of the 32-byte cache line, the maximum number of available space for messages in the MPB is limited by the amount of cache lines available in the MPB. Unfortunately, as discussed in Section 2.2.3, MINIX messages are 36 bytes, which is slightly larger than a single cache line. Therefore we are required to reserve two cache lines (64 bytes). On the other hand, the extra space gives us room for storing relevant meta data about the message. When dividing the 8KB MPB size by the size of two cache lines, we get:

$8192\ /\ 64 = 128\ messages$

Now that we know we can fit in 128 message maximum per MPB, we can look into different designs for storing messages in the MPB. In Sections 3.5.1 and 3.5.2 we discuss two different designs which we both implemented and evaluated.

3.5.1 Split Shared Queue

One way to implement IPC message passing between the CPUs is to split the MPBs into equaly sized slots, allocating one slot for every CPU in the system. As illustrated by Figure 3.8, when a CPU wishes to send a message to another CPU it can write the message in its slot at the MPB of the destination CPU. The advantage of this design is that CPUs do not need to obtain a lock when writing to the destination CPUs MPB. On the other hand the maximum amount of messages that can be in transit is quite low:

8192 / 64 / 48 = 2 messages

An important problem in this design which may occur without careful programming is an IPC deadlock. It may happen that at CPU 0 there is a process which wishes to receive only from another specific source process residing on another CPU. The source process has a message ready to be send, only the MPB slots are already filled with messages from another process destinated for the same process at CPU 0. In that case all processes would be waiting forever for more space to become available in the MPB. Therefore, it is important that the implementation takes care of removing messages from the MPB into a different local buffer as soon as it finds the message to avoid closed receive() deadlocks.

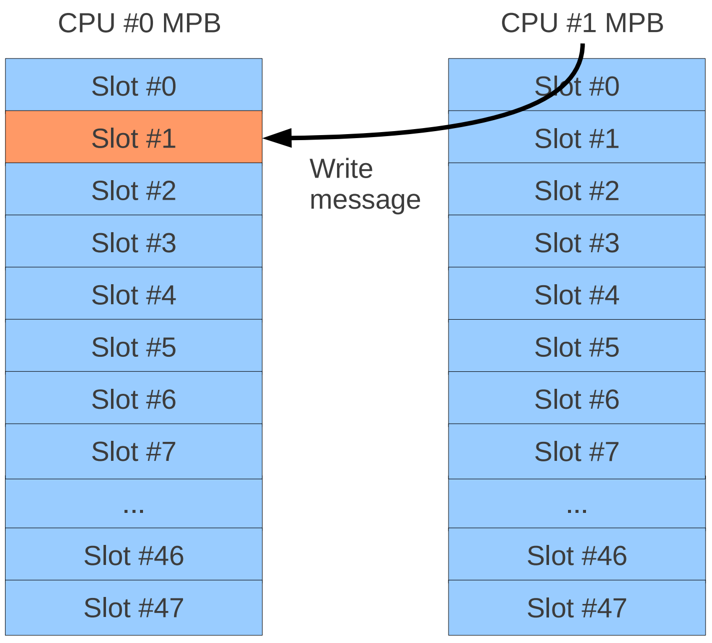


Figure 3.8: Splitting the MPB is separate slots per source CPU to avoid locking.

3.5.2 Lock Shared Queue

A different approach for IPC message passing between CPUs is to view the MPB as a big lock shared queue. When a CPU wishes to send a message to another CPU, it needs to aquire either the appropriate LOCK0 or LOCK1 register before it can write new messages to the MPB. As shown in Figure 3.9, the first cacheline(s) contains the current head and tail of the queue, such that both the receiving and sending CPUs known when the queue is full. The advantage of this design is that it allows the maximum amount of messages possible in transit. On the other hand the locking of the queue may become a performance problem. Also note that the receive() deadlock scenario still applies in this design.

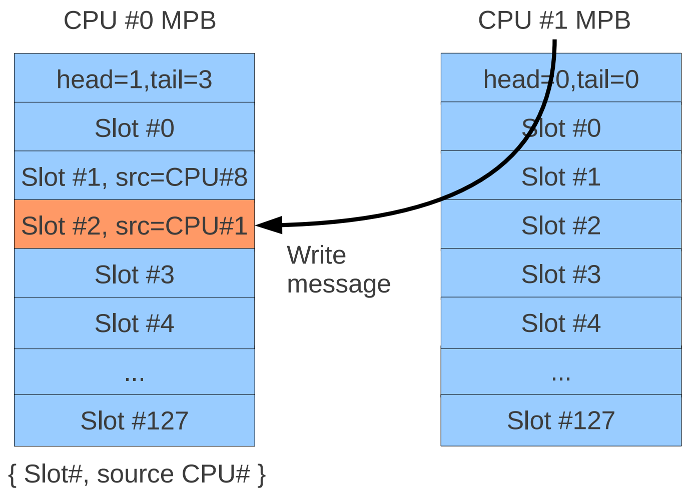


Figure 3.9: Lock-sharing the MPB for sending messages between CPUs.

3.5.3 Event Notifications

Another design choice regarding the IPC messaging is to decide when the MINIX kernel should look in the MPBs. For example, CPU 0 writes a message in the MPB of CPU 1. How does CPU 1 know that there is a new message from CPU 0? One way to deal with notifying other CPUs of new messages in the MPB is to use an *Inter Processor Interrupt* (IPI). It is possible to trigger an interrupt of other CPUs in the SCC by using the GLCFG0 or GLCFG1 configuration registers. Whenever a CPU receives such interrupt, it can look in its MPB to see if there are new message(s) from other CPUs. Performance of message delivery will be bounded by the performance of IPI delivery in this design.

Polling for new message in the MPBs is also possible. For example, the kernel can read its MPB at regular intervals or particular code sections. Polling can be done every time a process invokes a kernel trap, which is quite often. In that scenario the IPI are not necessary anymore. Performance of message delivery in this design is probably closely related to the number of kernel traps received and therefore the system load as a whole. A possible problem which may arrise in this design is that there are (temporarily) no kernel traps being delivered when the system is idle. In that case the kernel should either poll for new messages in a busy loop, or poll at regular time intervals.

Chapter 4

Implementation

This chapter describes our implementation of MINIX on the SCC in detail. Several components in MINIX have been added, modified or disabled. In Section 4.1 we give a high level overview of what changes were needed to which components. Later in Sections 4.2 and on, we look in detail to each modified component.

4.1 Overview

The MINIX SCC implementation required changes to several components, including the kernel, servers, libraries, drivers and build scripts. In short, the following changes are introduced:

Custom Bootloader Booting an operating system on the SCC requires a radically different procedure than on a PC. Since the SCC has no BIOS, Disk, CD-ROM, floppy, keyboard nor screen support, the bootmonitor does not work on the SCC. We implemented a custom bootloader for MINIX on the SCC, which is described in detail in Section 4.2.

Kernel The component in MINIX which required the most changes for the SCC implemention is the kernel. Most changes involved removing incompatible code, modifying existing functionality to support the SCC and extending the kernel for SCC specific procedures. The following is a summary of all kernel changes:

Unsupported Hardware Many hardware components on a PC do not exist on the SCC, which include ACPI, UART, PIC, PIT and IOAPIC. It is critical to disable any access to non existing hardware, as otherwise execution results are unpredictable or in some cases the kernel crashes. Unfortunately, there is no easy configurable way in MINIX to disable these components. We had to manually update the kernel code and retry booting on the SCC in an iterative process, every time solving one or more offending code occurences. Additionally, in the case of PIT, PIC and IOAPIC we had to modify the code to support other hardware, which is APIC.

APIC Every core in the SCC incorporates an APIC controller. Fortunately, MINIX already supports APIC from the SMP implementation. However, we had to modify the APIC implementation for the SCC in two places. First, the APIC timer calibration normally happens in MINIX using the PIT, which does not exist on the SCC. We changed to APIC timer calibration to use the configurable value of 800MHz, which is the highest possible tile frequency. Second, the APIC initialization is modified to support capturing the SCC specific IPIs.

SCC Extensions The kernel has been modified to support all SCC specific hardware. First, we modified to kernel such that it maps MPB memory and configuration register I/O memory in its own address space. Second, the implementation supports utilities for reading and writing SCC specific registers, detecting online CPUs, sending and receiving IPIs and using the per tile locking mechanism.

IPC As discussed in Section 3.3.1, our design uses remote stubs for global endpoints. We modified to kernel IPC code to detect remote stubs and invoke SCC specific subroutines for any of the IPC primitives. The kernel also creates remote stubs for all remote DS instances when it boots. Additionally, we implemented both MPB designs, split shared and lock shared. Finally, our implementation also supports both interrupt and polling mode. In Chapter 5 we evaluate which design shows the best performance.

Idle Process Idle is a special process in MINIX which runs only when no other process wants to be scheduled. In default MINIX, the CPU is switched to low powered mode inside the idle process. For the SCC implementation, we modified the idle process to busy loop on the MPB, if the implementation is configured for polling.

Constants Some constants in the default MINIX kernel are modified in our implementation. First, the SAMPLE_BUFFER_SIZE determines the size of the profiling buffer inside the kernel. This allows the kernel to save performance profiling information in the sample buffer. Per default, this buffer is configured for 80MB. In the SCC implementation we removed this buffer to reduce memory overhead. Additionally, _NR_SYS_PROCS defines how many server processes MINIX supports at maximum. We increased this number to 256 to make more room for the remote stubs.

Debugging When developing the custom bootloader, we faced many difficulties to debug boot problems. Often when booting the kernel failed, the CPU would reset and we got very little (if any) debug output. In attempt to tackle the issues, we modified the startup code of the kernel to load interrupt support as early as possible to capture exceptions. Additionally, we modified the kernel exception handler output to output a full register dump of the CPU and installed debug sccprintf() statements throughout the kernel.

Kernel Calls To allow servers in MINIX to read and write SCC specific information, we added the new sys_scc_ctl() kernel call. For example, a server process can use sys_scc_ctl() to read the core and tile ID it currently runs on. Additionally, the sys_getkinfo() is extended with GET_NEXTPROC and GET_SCCSTATS arguments, for retrieving the next free process slot and SCC specific performance statistics, respectively.

DS As discussed in Section 3.4, our design allows servers to publish themselves globally on the SCC. We modified DS in our implementation to support publishing global endpoints. DS is responsible broadcasting any new publishings to all other DS instances and for asking the kernel to create remote stubs.

VM The VM server is unchanged in our implementation with one exception. VM has a constant MAX_KERNEL_MAPPINGS, which defines the maximum kernel mappings. In default MINIX the value is 10, but for the SCC we need more to map all MPB and configuration registers in memory. We changed this value to 128.

PM In default MINIX, PM chooses process slot numbers for new processes. However, since the kernel needs to install remote stubs, PM could choose a slot which is already taken by a remote stub. Therefore, we modified PM to ask the kernel what the next slot should be with sys_getkinfo(GET_NEXTPROC).

TTY Default MINIX uses a VGA screen and PS2 keyboard as the system console. Unfortunately, neither exists on the SCC. Additionally, the TTY driver uses the BIOS in some places which the SCC also does not have. The only allowed output mechanism on the SCC is by writing to the serial port, such that the MCPC can capture the serial data and display it. Unfortunately, the full UART is not available on the SCC, so reading or configuring the serial port is not supported. Therefore, in our implementation we modified the TTY driver to only write to the serial port and disabled all references to the BIOS, keyboard, VGA screen and UART.

Ipctest In order to test and benchmark our implementation, we developed the Ipctest server. This server can generate remote and local IPC traffic and supports different test scenarios, including one-to-one, one-to-many and many-to-one.

RAM Disk When starting the MINIX operating system, the RAM disk plays an important role. It contains an in-memory file system with configuration files, startup scripts and programs which are needed to start the rest of the system. In default MINIX it starts several components which the SCC does not have, including ACPI driver, PCI server, Disk driver and INET server. All are disabled in our implementation. Additionally, we added a small script to start Ipctest server instances.

Headers The MINIX SCC implementation has additional header files which allows developers to configure the implementation. For example, these headers files can enable or disable debugging, interrupt mode and polling mode.

System Library In our implementation we added asynput(), asynflush() and asynwait() to the libsys library. These functions allow more efficient management of the asynchronous messages table used in the Ipctest server. Additionally, we increased the size of the asynchronous messages table to 8K to allow submitting more asynchronous messages using one senda() call.

Applications We added one new application, which is the seq command. It allows creating loops in shell scripts more conveniently, which was needed in the modified RAM Disk startup scripts.

Build Scripts The MINIX operating system is compiled using Makefiles, which describe what source files and compiler flags are needed for a program. Since the SCC has two P54C processors, MINIX must be compiled for specifically the P54C. Therefore we added the appropriate flags to the Makefiles. Additionally, more Makefiles were added for every new component described earlier.

4.2 Bootcode

Since the SCC has no BIOS, disk, floppy nor CD-ROM support, the default MINIX boot code does not work on the SCC. In our implementation we completely rewrote the bootstrap code. This Section explains all relevant details on the new startup code.

To start an operating system on the SCC, the user must use the sccGui program on the MCPC to select and upload a *operating system image*. The image should be in the Intel specific OBJ format. This format describes the operating system code and data in hexadecimal, text encoded format. Fortunately, the Linux implementation for the SCC provided by Intel already contains a program called *bin2obj* to convert a binary OS image to the OBJ format. Our implementation uses the bin2obj program to convert our own binary boot image to OBJ. As described in Section 2.2.1, the MINIX kernel requires a boot image containing several core servers for bootstrapping. Note that the MINIX boot image is different from the OS image needed to start an operating system on the SCC. The OS image should contain at least a bootloader with a kernel and optionally more.

Figure 4.1 shows the contents of the MINIX OS image. When CPUs in the SCC receive a reset signal from sccGui, each core starts execution at the *reset vector*. The reset vector is a predefined memory location, namely address 0xFFFFFFF0. The MINIX OS image contains a small piece of code at the reset vector, which jumps to *Bootreset.S* at the beginning of the MINIX bootstrapping code on address 0x90000. Bootreset is an assembly program responsible for the following:

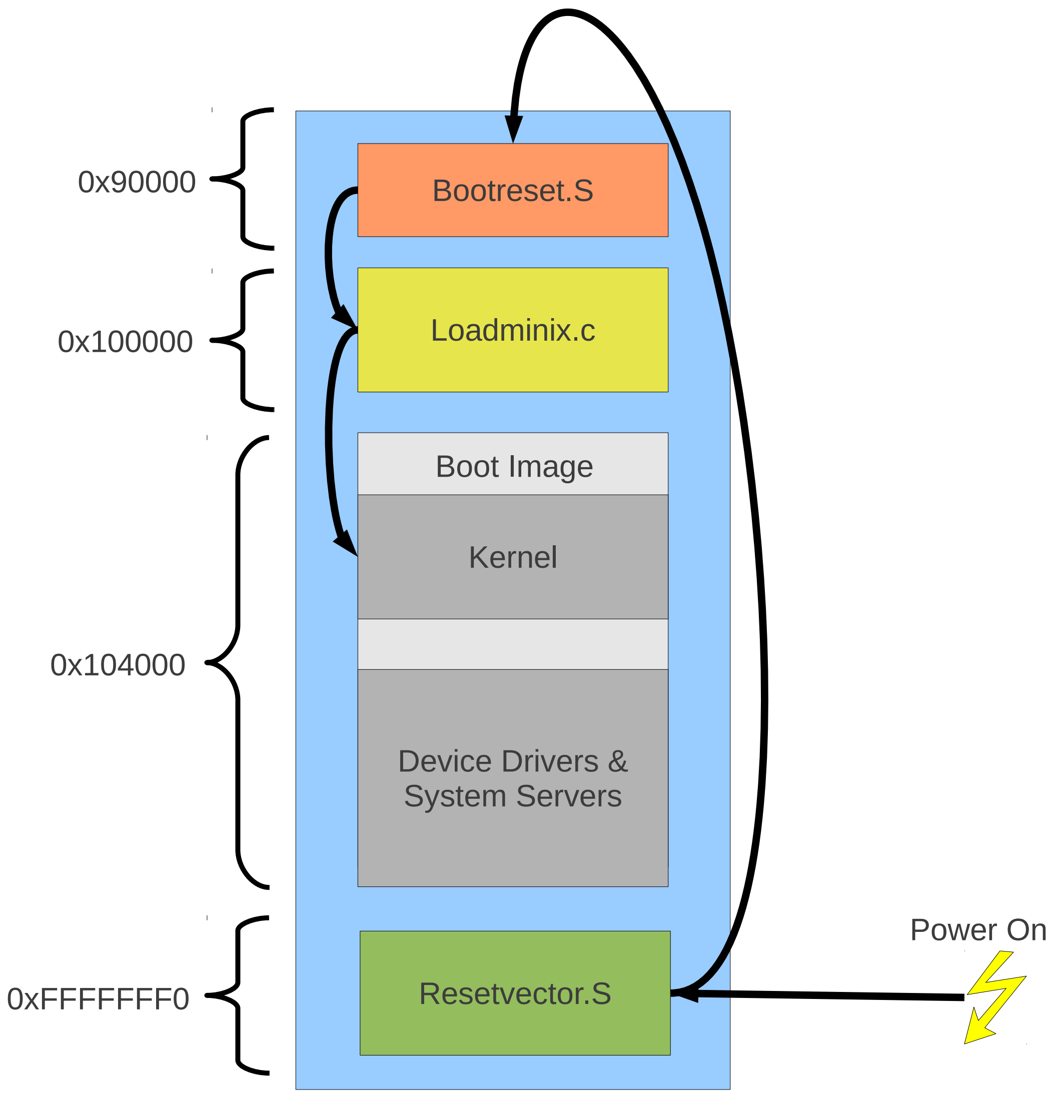


Figure 4.1: Contents of the MINIX OS image for booting on the SCC.

Initializing Segmentation Most Intel based CPUs offer a mechanism called *segmentation* [52] to provide memory protection. With segmentation, memory is divided in different parts called segments. Every segment has configurable access permissions, such as read(only), write(only) and whether it can be used by unprivileged programs. The bootreset program is responsible for initializing segmentation

Switching to Protected Mode Additionally the Bootreset program must switch to *protected mode* [53]. When the CPU is turned on, it operates in 16-bit *realmode.* MINIX requires 32-bit protected mode, which allows more system memory and enables memory protection.

Allocate Temporary Stack In order to continue to the next program, the stack must be initialized. Bootreset allocates a 8KB stack and pushes the address of the MINIX boot image on it.

After the Bootreset.S program finished, it jumps to the *Loadminix.c* program at address 0x100000. Loadminix is a more complex C program which implements similar functionality as the bootmonitor. In short, the Loadminix program includes:

Image Validation The first task of the Loadminix program is to validate the boot image. This is needed to prevent accidentally loading an invalid image. Inside the boot image there are several predefined *magic numbers*, which the Loadminix program reads and checks. If an incorrect magic number is found, the boot image is rejected and MINIX does not boot.

Image Relocation Since the MINIX boot image only contains code and data, there is no extra room reserved for the heap in the boot image itself. However, the kernel expects that the bootloader loads all programs from the boot image in memory, include code and data but also reserve room for the heap. Therefore, the Loadminix program scans the boot image, and relocates the programs to another memory location to allocate heap space. Additionally, the heaps are cleared to zero, including the kernel heap.

Kernel Arguments Another task of the Loadminix program is to collect kernel arguments. This includes a list of A.OUT headers from the boot image and optional configuration flags, such as debug mode. One kernel parameter is critical for booting and running MINIX, which is the memory map parameter. It defines which memory locations are used and which ones are free. An incorrect setting could cause VM to allocate memory pages twice to different processes.

Once the Loadminix program completed the above tasks, it must jump to the kernel. At that point, the bootstrapping code is done and the kernel takes over execution. The kernel then schedules all programs from the boot image to finalize the boot process.

4.3 DS

As described in Section 3.4, our endpoint discovery solution uses DS to publish global endpoints to all cores in the SCC. In the implementation, we modified the following in the standard DS server. First, in default MINIX, RS is the only server which is allowed to publish endpoints to DS. We removed this restriction such that servers can ask DS directly to publish themselves. Second, we extended the publishing code in DS with the new DFS_REMOTE_PUB flag. If it is set, DS will treat the publish request as a global endpoint request. Otherwise, it assumes the publish request is for a local endpoint only. Finally, when DS starts it asks the kernel which CPUs are running using the new sys_scc_ctl() kernel call. This is necessary, as DS needs to know to which other remote DS instances it should forward new publish requests.

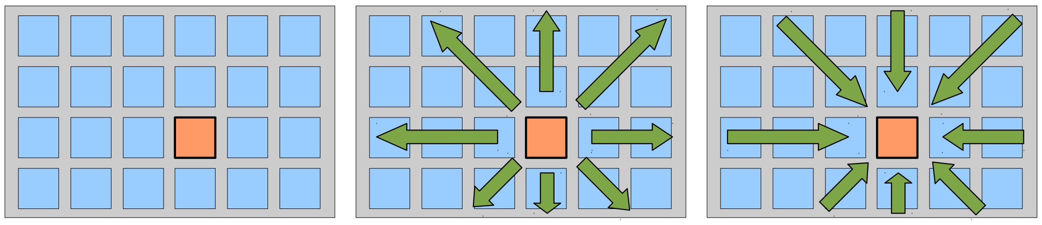

(a) Server wants to publish (b) DS forwards to all others (c) DS receives acknowledgements

Figure 4.2: Publishing a global endpoint to all cores.

As shown in Figure 4.2, DS publishes a global endpoint in three steps. First, the server sends a publish request message to DS using the DFS_REMOTE_PUB flag. The publish request message must contain a unique keyname. This keyname is later used by other servers, possibly on other CPUs, to find the endpoint. When DS receives the message in Figure 4.2a, it detects the sender requests a global publish. DS proceedes to the next step in Figure 4.2b, which is to forward the global publish request to all other DS instances at other CPUs in the SCC. DS only forwards the publish requests to DS instances on CPUs which are actually online. When the remote DS instances receive the publish request, they create a remote stub for the server and send back an acknowledgement message. Once all acknowledgements arrived at the local DS server in Figure 4.2c, it replies back to the server to unblock it. At that point, the server has an endpoint on all CPUs and other servers can find the corresponding endpoint using the unique key.

Note that DS does not immediately reply back to the sender when it first received the global publish request. It must wait with sending the reply until it knows all other CPUs have installed the remote stub. If DS would not wait, the server could start sending messages to other remote servers. But since some CPUs may not have installed a remote stub for the sender, the kernel does not know what source endpoint it should put in the messages.

4.4 Kernel

In order to implement the cross core IPC mechanism, we had to make several changes to the kernel. This section describes in detail the most important changes for both synchronous and asynchronous messaging.

4.4.1 Process Structure

As described in Section 3.3.1, our solution for global addressing is to install remote stub processes in the kernel process table. In the implementation, we reused many existing fields from the process structure, which is fully listed in Appendix B. First, the *p_cpu* field from the SMP implementation can be reused for our work to also store the CPU number of a remote stub. Second, we added the RTS_REMOTE flag for the *p_rts_flags* field, indicating the process entry is a remote stub. This also prevents the remote stub from being schedulable, because the p_rts_flags field determines if a process is schedulable in MINIX. If it is zero, the process can be run, otherwise it must never be scheduled. Finally, we also needed to add a few new fields, including:

Remote Send Queue The SCC messaging code keeps track of a list of processes which are sending one or more message(s) to remote processes. This is required, as sometimes when a process tries to send a message to a remote process at a particular CPU, the destination MPB may be full. In that case, the SCC messaging code should retry transmission later for each process on the remote send queue.

Remote Deliver Queue This field is used for fast remote message delivery. Whenever the SCC messaging code receives a message from a remote process at another CPU, it first tries to deliver it immediately. If the message cannot be delivered immediately, for example because the destination process is not currently receiving, the message is stored in a temporary buffer. Additionally, the temporary buffer is added to the Remote Deliver Queue of the local process. Eventually, when the local process invokes receive(), the remote message is consumed and the buffer is removed from the remote deliver queue.

Stub Remote Endpoint When a local process sends a message to a remote process using its published global endpoint, the kernel must know the local endpoint of the actual process at the remote CPU. The kernel can then put this endpoint number as the destination on the message in the MPB, such that the remote kernel can deliver the message to the correct process.

Published Remote Endpoints When a published process sends a message to a remote process, the kernel must also fill in a source endpoint. Since the global endpoint of a published process may differ on the CPUs, the kernel must know what the endpoint of a published process is, at the remote side.

4.4.2 Synchronous IPC

When a process on a particular core in the SCC wishes to send a message to another process on a different core, the kernel needs the following. First, the kernel needs the message to be send, including the source endpoint of the sender. When sending the message to the other core, the remote kernel must know from which process it came. As discussed in Section 4.4.1, we added the *Published Remote Endpoints* field to the process structure for this. On sending, the local kernel writes the endpoint number of its published stub at the remote side in the MPB. When the remote kernel reads in its MPB, it will find a new message with the source endpoint of the remote stub of the sender.

```
typedef struct
{
    message data;                     /* Payload including source */
    endpoint_t dest;                  /* Destination endpoint */
    int status;                       /* Contains result flags */
    char padding[SCC_MSG_PADDING];    /* Pad message to 2 cache lines */
}
scc_message;
```

Figure 4.3: Message format for intercore MINIX messages on the SCC

Second, the correct destination endpoint must be known. Since we save the original local endpoint of remote stubs in the process table, we can fill in that value as the destination endpoint. However, as described in Section 2.2.3, the default MINIX message format does not include a destination endpoint as it is already given as a parameter for the IPC primitives. As shown in Figure 4.3, we created a new intercore message format which includes a destination field. Additionally, the intercore message format contains a status field. The status field defines the type of message, which can be any of the following:

SCC_MSGSTAT_SEND Message was created by a blocking send() call.

SCC_MSGSTAT_ACK Acknowledgement for messages created by send() (explained later).

SCC_MSGSTAT_SENDA Asynchronous message from senda().

SCC_MSGSTAT_NOTIFY Notification message from notify().

Figure 4.4 presents the high level overview of the SCC messaging implementation in MINIX. There are three execution paths which invoke the SCC messaging code. First, the IPC primitives discussed in Section 2.2.3 and 2.2.4 detect if the endpoint argument points to a remote stub. If that is the case, the SCC messaging code takes over execution and otherwise the default local messaging code continues. Second, when a core receives an IPI, the SCC messaging code is also invoked to check for new messages and to (re)try sending messages to other cores. Finally, if polling mode is enabled the SCC messaging code is called from either the idle process or when a kernel call is made.

The SCC messaging code is structured in routines with a well defined task. On the highest level, the *scc_process_msg()* function is responsible for (re)trying sending messages to other cores and to scan the local MPB for incoming messages from other cores. The *scc_try_transfer()* and *scc_mpb_scan()* functions implement these purposes, respectively. Every process on the remote send queue, as described in Section 4.4.1, is invoked for the scc_try_transfer() function such that all pending messages get a chance to eventually be delivered. When the scc_mpb_scan() routine finds a new message in the MPB, it invokes *scc_incoming_msg()* which validates and processes the message. It can then decide to deliver the message to a local process with *scc_try_deliver()*.

Sending a message to a remote process requires several steps. Figure 4.5 illustrates the flow of execution when invoking send() on a global endpoint. This begins as soon as the process calls sends(). The kernel messaging code will detect that the destination endpoint is a remote process and enters the SCC specific *scc_send()* routine. There the kernel constructs a scc_message and attempts to write it to the destination CPU's MPB with *scc_write_mpb()*. This operation could fail if there is no room in the remote MPB. If the write failed because there was no room, the kernel flags in the remote MPB that it wishes to receive an IPI back when room comes free and adds the sender to the Remote Send Queue. When that happens the kernel will retry to send to complete delivery. If the write succeeded, the kernel sends an IPI to the remote CPU such that it knows there is a new message waiting. The sending process is then blocked until an acknowledgement is received from the remote CPU. Note that this is required as the send() primitive must only return when the receiver has either received the message or an error occured, as described in Section 2.2.3. When the remote process consumes the message with receive(), the remote kernel writes back the acknowledgement. On reception in scc_incoming_msg(), the sending process is unblocked.

Receiving a message from a global endpoint consists of the steps shown in Figure 4.6. There are two main execution paths when receiving a message. Either the sending process first tries to deliver the message before the receiving process calls receive() or the receiving process is first. In the first case, execution starts with *scc_recv()*. The kernel looks if there is a buffered message which was received earlier before receive() was called. Since there is a buffered message, the kernel next looks if the message has the type SCC_MSGSTAT_SEND, meaning it was generated by send(). If that is true the kernel must send back an acknowledgement message now or try again later when room comes free by setting the "IPI back" flag in the remote MPB. If the message is not generated by send(), it is delivered immediately. Consider the second case, where the receive() is called first. The kernel then scans the MPB to see if there are more messages now, which can happen if the IPI has not yet arrived while the sender did already write, or in polling mode. If there are messages, the kernel tries to deliver them as described before and otherwise buffers it for later. When there are no messages in the MPB, the kernel sends an IPI to any core which requested it with an "IPI back" flag.

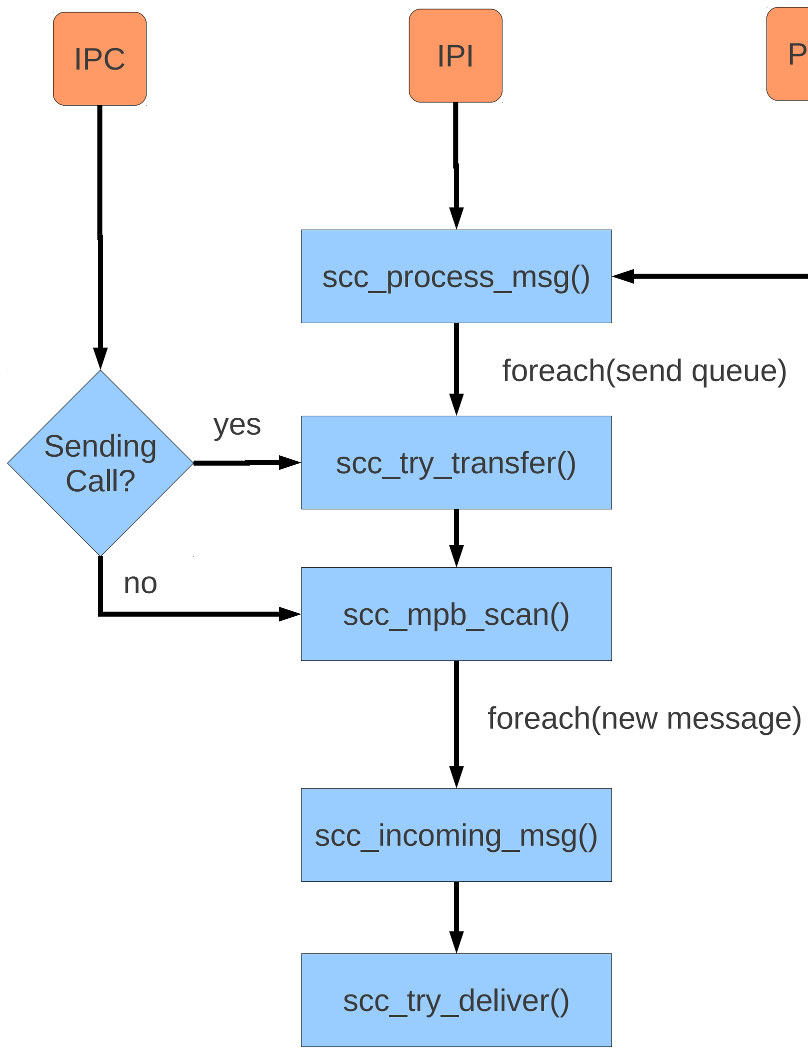


Figure 4.4: High level overview of the SCC messaging implementation in MINIX.

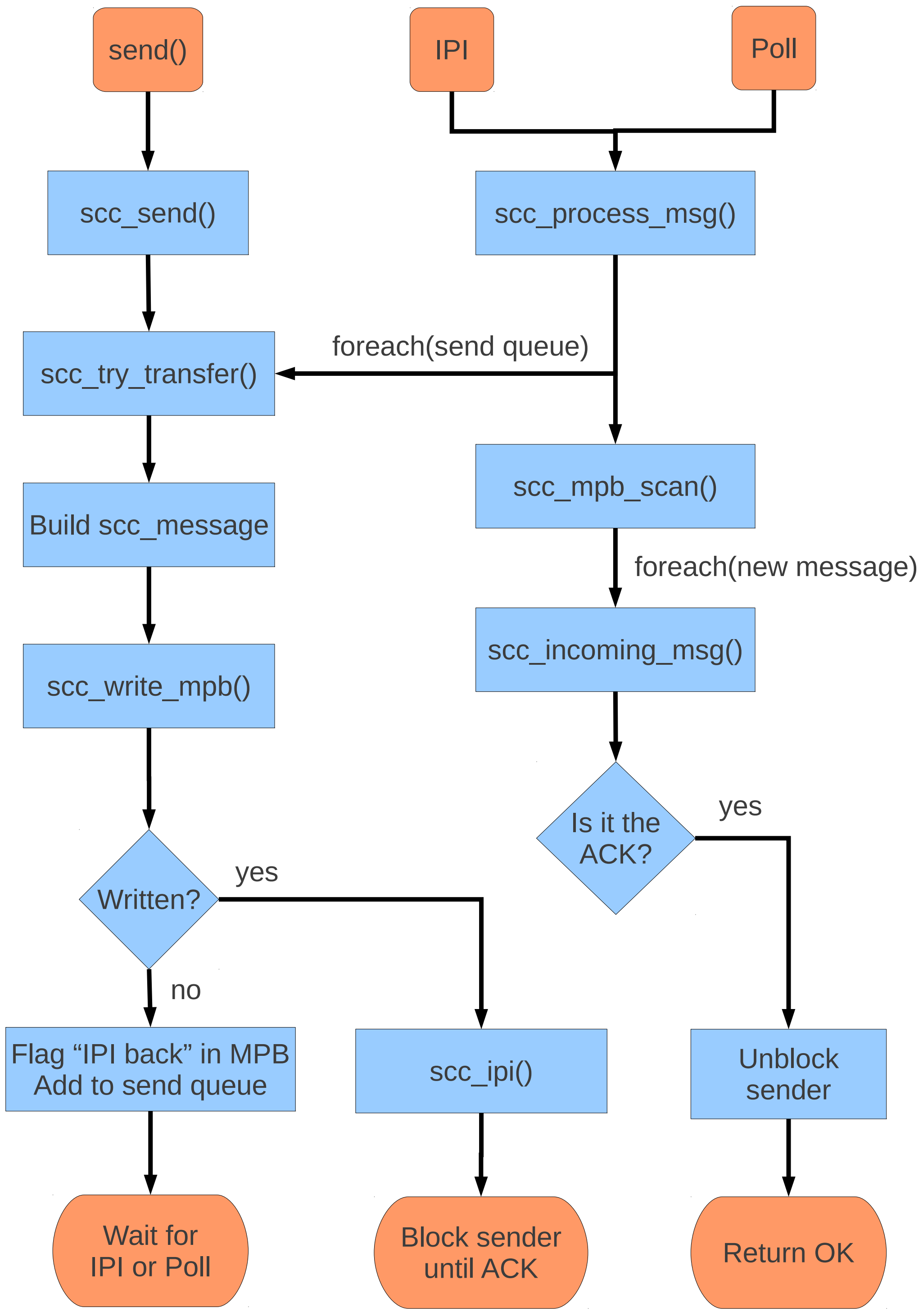


Figure 4.5: Flow of execution when invoking send() on a global endpoint.

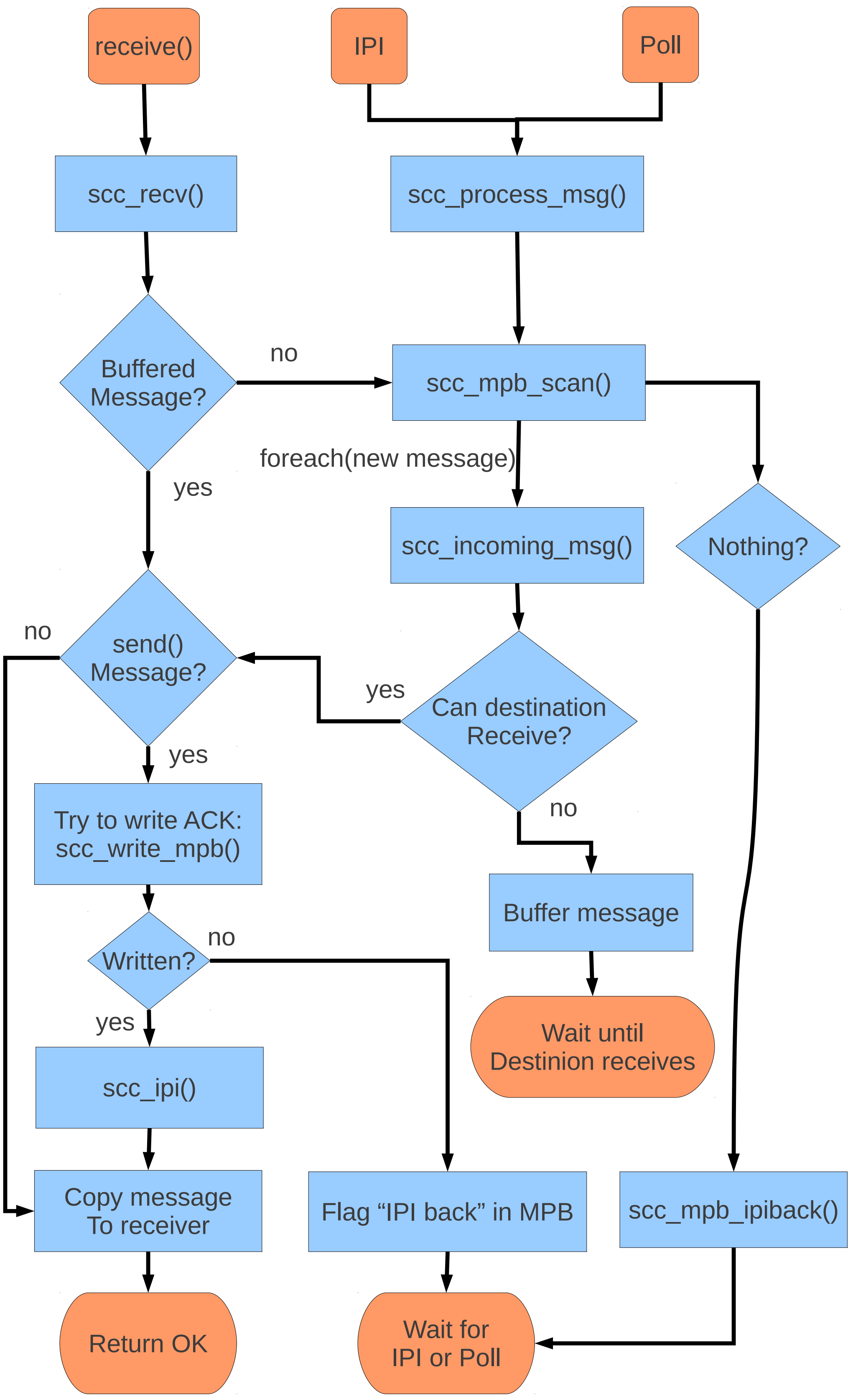


Figure 4.6: Flow of execution when receiving from a global endpoint.

4.4.3 Asynchronous IPC

In contrast to the send() primitive, sending asynchronous messages does not require acknowledgements as the sender is never blocked. Figure 4.7 illustrates the flow of execution when senda() is called with global endpoints. For every global endpoint in the messaging table, the kernel calls *scc_senda()*. There it constructs a scc_message with the type SCC_MSGSTAT_SENDA and tries to write it to the destination CPU's MPB. If the write is successful, the kernel sends an IPI to the destination CPU and updates the messaging table accordingly. On failure, the kernel sets the "IPI back" flag in the destination MPB and retries later at the next IPI or polling.

Notifications to global endpoints follow a similar path as senda() with some small differences. When creating the scc_message, the SCC_MSGSTAT_NOTIFY type is set instead. Additionally, at the receiving side the scc_try_deliver() function updates the notify bitmap if the message cannot be delivered immediately instead of keeping it buffered, to save memory.

Unfortunately, our implementation does not fully implement the sendnb() primitive for global endpoints. The reason is that it is not possible for the sender to look if the receiving remote process is ready to accept the message without blocking. As discussed in Section 3.2, kernels cannot read each other's private memory, including process table entries. Therefore, the sendnb() would require sending an inter-processor message to ask the remote side if it can immediatly consume the message. However, this would require blocking the sending process until the reply is received, which violates the non blocking constraint of sendnb(). For this reason our implementation returns an error when sendnb() is called on a global endpoint. A better alternative for programs is to use senda() instead.

4.4.4 Event Notifications

IPIs function as a way to notify other CPUs when a special event occurs. In our implementation, IPIs are send for two types of events:

New Message(s) As soon as a message is written to the MPB of another core, the sending core triggers an IPI at the receiving core. This invokes the scc_process_msg() function at the receiving core, which will in turn call scc_scan_mpb() to read new incoming messages in the MPB.

MPB Space Available Sometimes it can happen the MPB of a core becomes full. In that case, new messages cannot be written to the MPB and the implementation must retry sending again later. To know when the kernel can retry sending more messages, it sets an "IPI back" flag in the MPB of the remote core. If the MPB at the remote core becomes empty, it will send an IPI to all cores which have set such flag so that they can retry sending more messages.

Additionally, our implementation supports polling mode. The kernel will invoke scc_process_msg() every time a kernel trap is done and when idle process runs. It is necessary to let the kernel busy loop on the scc_process_msg() function, as it may happen messages are written to the MPB while the system is completely idle.

4.4.5 MPB schemes

Our implemention supports the Split Shared and Lock Shared MPB schemes, as discussed in Section 3.5.1 and 3.5.2, respectively. The implementation does not significantly differ from the design, with one exception. Since the MPBs also need to store the "IPI back" flag, the first cache line is reserved in the Lock Shared scheme for storing the "IPI back" bitmap. Every core in the SCC has its own bit in every MPB. If the bit is 1, the implementation assumes it must send an IPI when the MPB becomes empty. The bit is cleared when sending IPIs back. For the Split

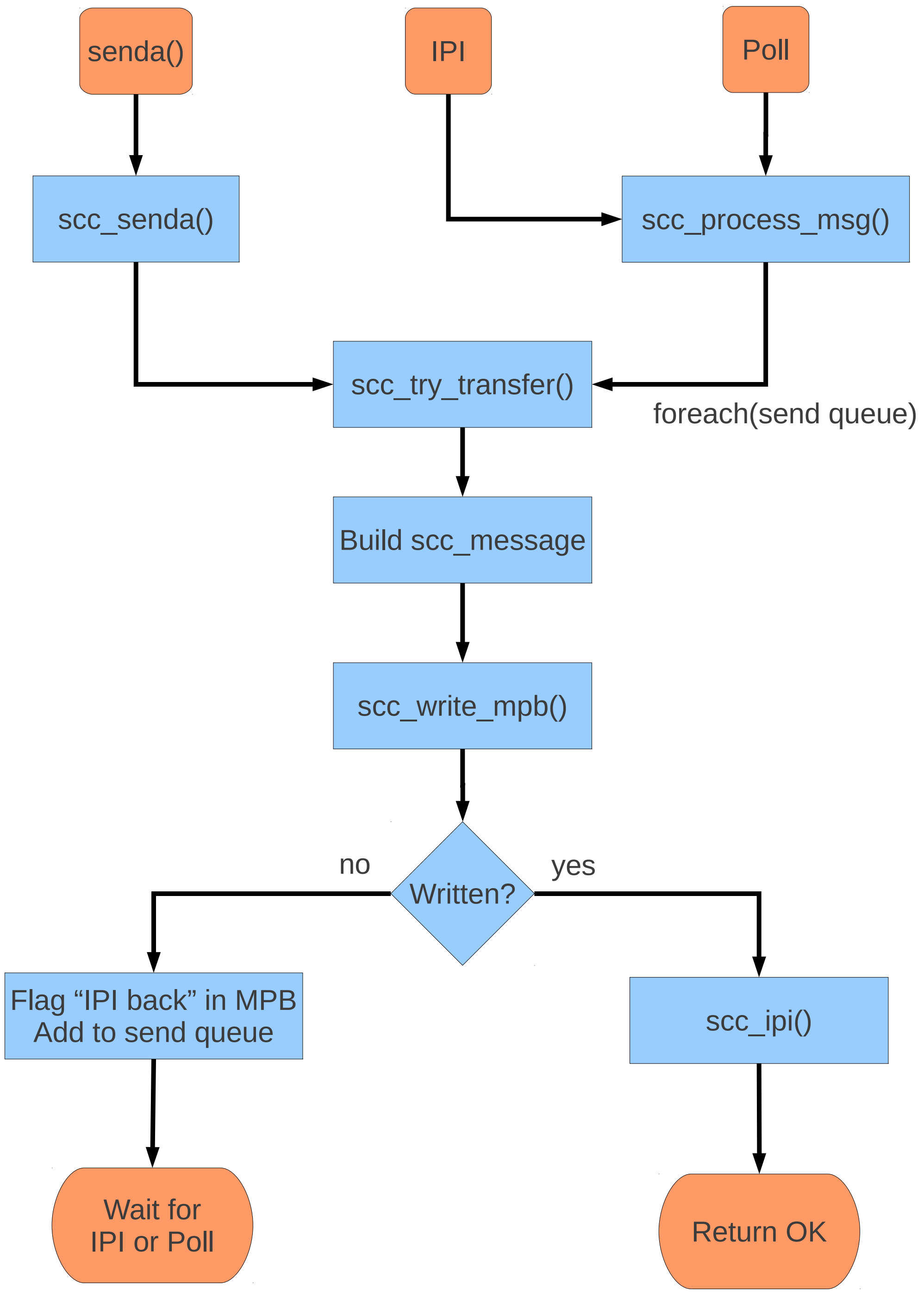


Figure 4.7: Flow of execution when sending asynchroneous messages to a global endpoint.

Shared MPB scheme, we did not need to reserve a cache line, as from the design we can determine which slots have become full. For example, if CPU 1 is flooding messages to CPU 0, the MPB slot of CPU 1 at CPU 0 quickly becomes full. Once all messages from the slot are consumed, the implementation sends back an IPI only to CPU 1.

4.4.6 New Kernel Calls

To give processes access to SCC specific properties, we introduced the *sys_scc_ctl()* kernel call. It should only be called by server processes, and has the following interface:

int sys_scc_ctl(int command, int param, int param2, int param3)

This function enables processes to use SCC specific functions. The first parameter is the command, followed by a list of parameters. Depending on the command, the parameters should be set accordingly. The following commands are supported:

SCC_CTL_MYTILEID
Returns local tile ID. No arguments.

SCC_CTL_MYCOREID
Returns local core ID. No arguments.

SCC_CTL_ISONLINE
Checks if the given core ID is online. Returns 1 if online and 0 otherwise.

SCC_CTL_NEWSTUB
Used by DS to install a remote stub for the given core ID and endpoint pair. Returns the endpoint of the stub on success and an error code otherwise.

SCC_CTL_MYREMOTENDPT
Used by DS to set the Published Remote Endpoints field for the local process during publishing. DS calls this command for every acknowledgement message from other DS instances. Return OK on success and an error code otherwise.

SCC_CTL_RESETPERF
Used internally by our implementation to reset performance counters.

In addition we extended the *sys_getkinfo()* kernel call with the following two commands. GET_NEXTPROC, returns the next free PID in the process table. Used by PM to find an empty slot in the kernel process table which does not collide with remote stubs. GET_SCCSTATS, used internally by our implementation to retrieve the values of performance counters.

Chapter 5

Evaluation

This chapter evaluates the performance of our implemention. First in Section 5.1, we describe the setup we used to perform the benchmarks, including the Ipctest load generator program. From Section 5.2 and one, we look at different load scenarios using various IPC primitives.

5.1 Setup

To benchmark our implementation we created the Ipctest server program. It is a configurable server process and sends and receives IPC messages as fast as possible. As shown in Figure 5.1, we implemented three different load scenarios which we believe come the closest to real workload. First, the *one-to-one* scenario in Figure 5.1a runs with two Ipctest instances. The first instance sends a certain number of messages to the other instance. The scenario finishes when all messages have been received by the other instance. Second, the *one-to-many* scenario in Figure 5.1b runs with three or more Ipctest instances. One instance sends a predefined number of messages to all other instances. Each other instance always receives the same amount and the scenario terminates when all messages have arrived at their destination. Finally, the *many-to-one* scenario in Figure 5.1c also runs with three or more instances, but here the other instances send a predefined number of messages to one particular instance. Many-to-one terminates when this instance received all messages from all other instances.

To gain insight in the performance of the SCC messaging code, we need to compare it against existing mechanisms. The performance benchmark results should give answers to the following questions:

IPI versus Polling Mode An influential variable in the performance of the SCC messaging code is IPI delivery. Interesting is to compare the performance of IPI event notification against polling mode. Our expectation is that polling mode will be faster for at least point to point delivery, since the code checks more often for new events.

Split versus Lock Shared MPB Another interesting question which MPB scheme performs the best for sending and receiving messages in the SCC. Both schemes have potential for high performance. The performance results should also answer the question whether the locking in the Lock Shared scheme becomes a bottleneck with an increasing number of cores.

Synchronous versus Asynchronous As discussed earlier in Section 4.4.2, our implemention needs to send acknowledgement messages back when doing a send(). For asynchronous messages this is not required. Therefore, we expect that asynchronous messaging is rougly two times as fast as synchronous messaging.

Local versus Remote IPC Relevant for the implementation is also the cost of remote messages compared to local messages. We expect that there will be some overhead on the SCC, but it should not be substantial. For local we started all Ipctest instances on the same CPU.

SCC versus SMP To learn more about the quality of our implementation, we compared the ratio of local versus remote IPC on both the SCC and a SMP system. This should answer the question whether the overhead of remote messaging in our implementation is reasonable. We used an 8-core 2.26Ghz Intel Xeon to compare with the SCC.

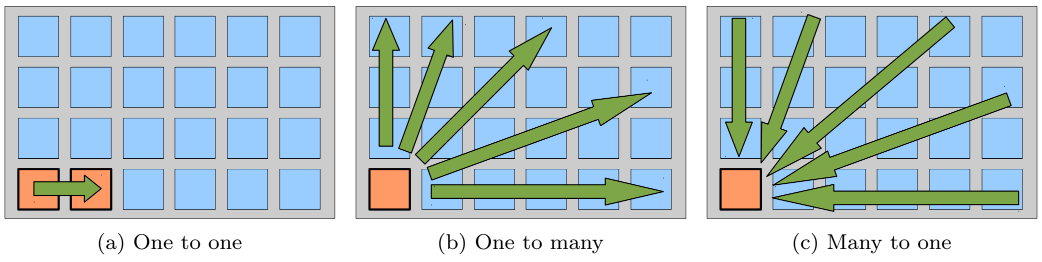

(a) One to one (b) One to many (c) Many to one

Figure 5.1: IPC messaging scenarios implemented in ipctest.

5.2 One-to-One

The one-to-one scenario runs with two instances. The first instance sends a certain number of messages to the second instances. For the SCC we ran the one-to-one test with 10000 messages. Figure 5.2 shows the results of one-to-one send() on the SCC. The results show that polling mode is almost twice as fast as IPI mode. The explanation is that IPIs take certain time to be delivered at the destination. Additionally, the results show that the cost of remote send() is between 2 to 3 times slower than local send(). Another interesting observation is that when IPIs are combined with polling, the result is slower than both enabled alone. The reason is that when enabling both, the code also suffers from the overhead of both. Finally, there is a difference between the performance of Split and Lock Shared MPB schemes. Unfortunately, we do not have a clear explanation for this behaviour.

We ran the same scenario on the 8-core Intel Xeon, as shown in Figure 5.3. Since the Intel Xeon is a much more advanced processes as the older P54C, we could send much more messages in approximately the same amount of time. The first Ipctest instance sends 10 million messages to the second Ipctest instance. The remote messaging is about 2 times as slow compared to local message delivery on the SMP, which matches closely with our local versus remote ratio on the SCC.

Finally, we ran a similar test on the SCC with asynchronous message delivery. Unfortunately, this test does not show data on local message delivery, as the senda() implementation for local delivery is unreasonably slow and makes no sense to compare against. The reason for this is that the senda() code for local delivery is not optimized for the extreme high load we test against. Figure 5.4 shows the performance results of remote asynchronous message delivery with senda() on the SCC. First, polling with Lock Shared MPB is the fastest, but the others are not far behind. Second, we can conclude from these results that senda() is indeed twice as fast as send(), due to the extra IPI needed for send().

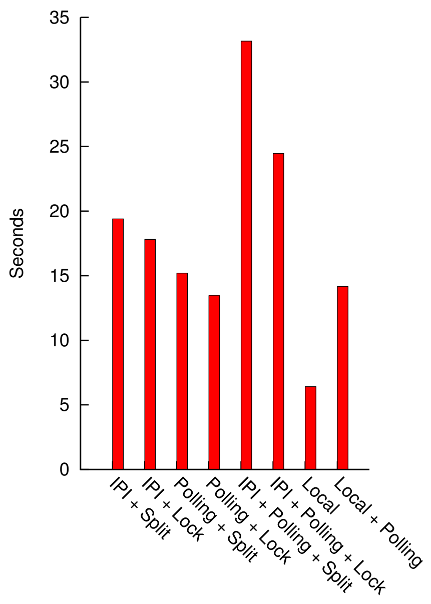


Figure 5.2: One-to-One send() 10000 messages on the SCC.

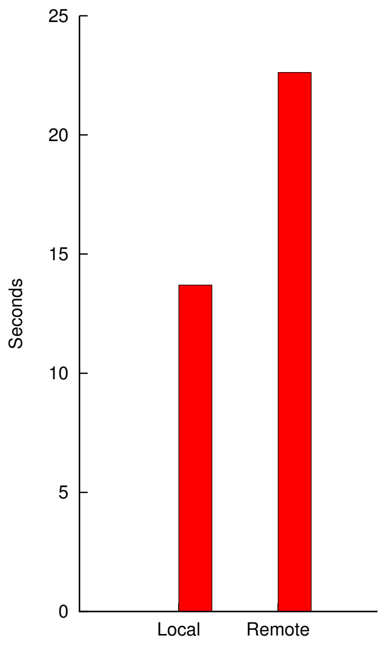


Figure 5.3: One-to-One send() 10 million messages on the Intel Xeon.

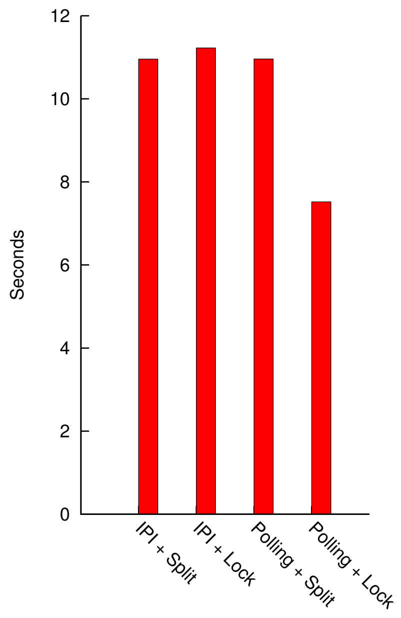


Figure 5.4: One-to-One senda() 10000 messages on the SCC.

5.3 One-to-Many

The next scenario we tested is one-to-many, which is useful to simulate broadcasting state updates. One instance sends a predefined number of messages to all other instances, which is set to 1000 on the SCC. Figure 5.5 shows the performance results of one-to-many send() on the SCC. The first observation is that the fastest remote mechanism, IPIs enabled with Lock Shared MPBs, is about 2x slower than local message delivery. This matches with earlier results in the one-to-one send() test. Second, polling mode is slightly slower than IPI mode. The reason for this behaviour is the higher load on the mesh network. Finally, we see that when polling mode is enabled, local message delivery also suffers from its overhead. This is because our implementation polls the MPB on every kernel call.

Figure 5.6 displays the same test scenario for the Intel Xeon. One instance sends 1 million messages to each other instance. We can see that the cost of local message delivery increases slighty with the number of extra instances, but remote messaging comes with far more overhead. With only two destination instances, the remote messaging overhead is 2 times the cost of a local message. Later with 7 destination instances the cost is 3 times. The most obvious explanation for this behaviour is due to the design of SMP. The increased load on the shared bus between the CPUs slowly becomes a bottleneck.

Finally, we evaluated the performance of senda() in the one-to-many scenario, as presented in Figure 5.7. The results in this case are very close. Both IPI and polling mode show the same performance with 32 or more cores. The reason is that the sending core simply cannot write messages faster at that point. It does not need to wait for acknowledgements in this test, thus the performance is limited by the speed at which the receivers can consume the messages from the sender. Of course, the receivers together are faster than the sender alone, explaining why IPI and polling mode both receive the same performance in this case. An unexplained oddity in the results is there is a sudden drop in the graph for IPI mode with Split Shared MPB. We do not have a clear explanation for this behaviour.

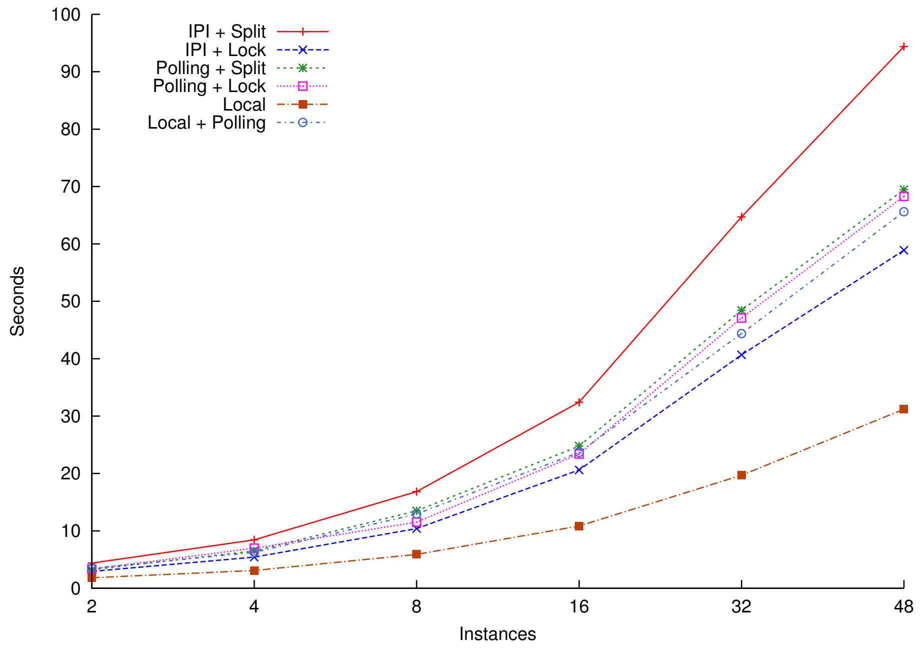


Figure 5.5: One-to-Many send() with 1000 messages on the SCC.

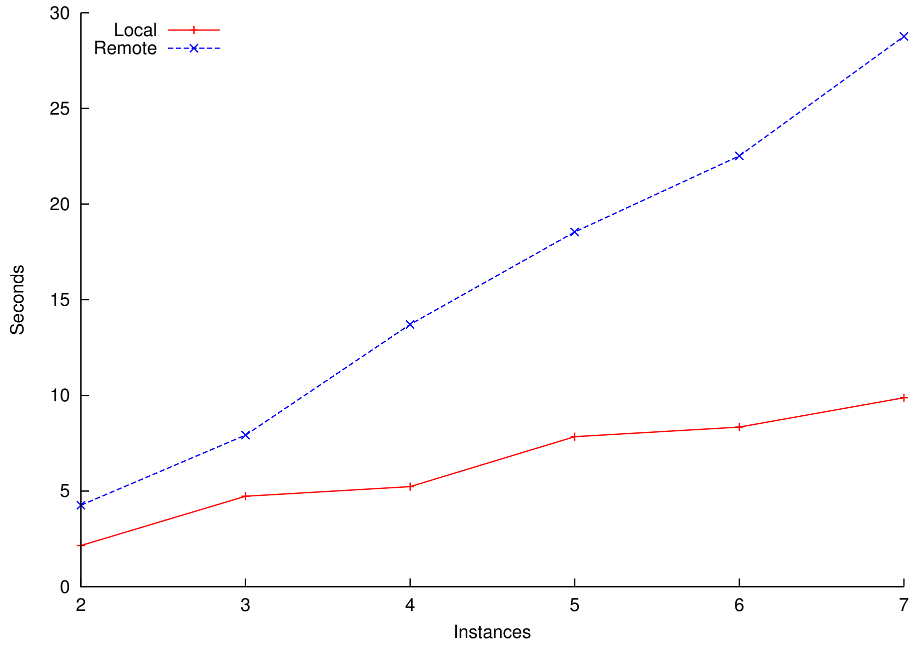


Figure 5.6: One-to-Many send() with 1 million messages on the Intel Xeon.

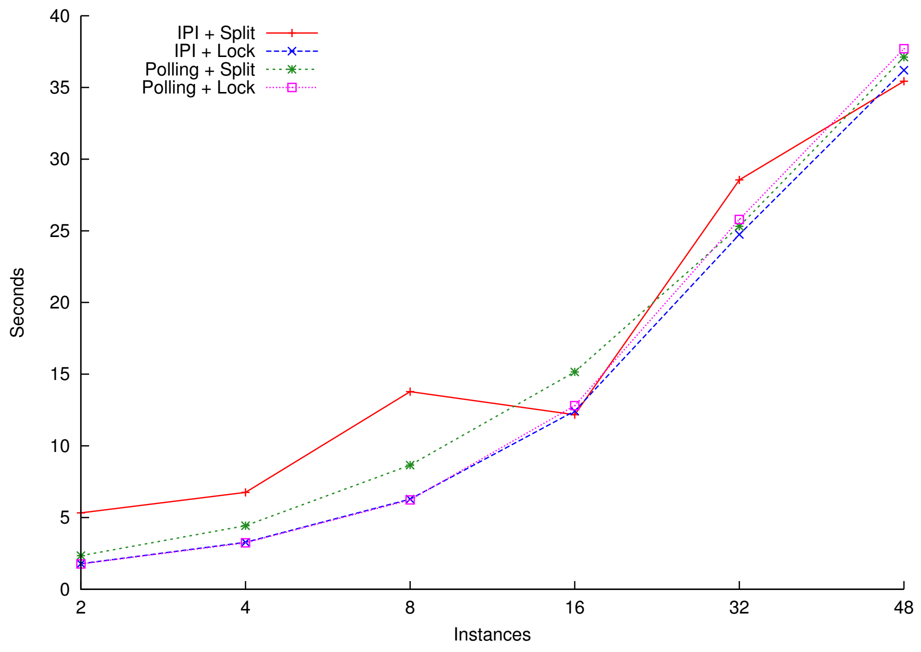


Figure 5.7: One-to-Many senda() with 1000 messages on the SCC.

5.4 Many-to-One

The last test scenario we evaluated is many-to-one. Here two or more instances a predefined number send messages to a particular other instance. Figure 5.8 displays the performance results when sending 1000 synchronous messages on the SCC. Interesting in these results is that remote messaging is only 1.5 times the cost of local messaging. A possible explanation is that in contrast to the one-to-many scenario, the receiver wastes less time waiting for acknowledgements and spends more time processing messages. Additionally, when polling mode is enabled, local messages is the slowest of all. Finally, we can see from the results that the choice between IPI and polling mode does not make a significant difference in this case, nor is Split Shared or Lock Shared MPB. This can also be explained by the fact the receiver spends more time doing useful work and less time waiting for IPIs or polls.

Figure 5.10 shows the results of many-to-one send() on the Intel Xeon. Here we see again that remote messaging is 2 times the cost of local messaging. However, in contrast to the one-to-many scenario, performance does not slowly drop to 3 times the cost. These results are similar to the results of the SCC implementation. Finally, Figure 5.9 displays the results of many-to-one senda(). Unlike one-to-many, polling mode performance is very bad in this case. With 8 cores or more, polling mode becomes too slow to be practical. The most obvious reason is that polling puts a too high load on the mesh, since all cores are flooding mesh with polling requests on all other MPBs. However, it is unclear why in this case performance is much worse than the one-to-many test. Finally, the results show that Lock Shared is slightly faster than Split Shared.

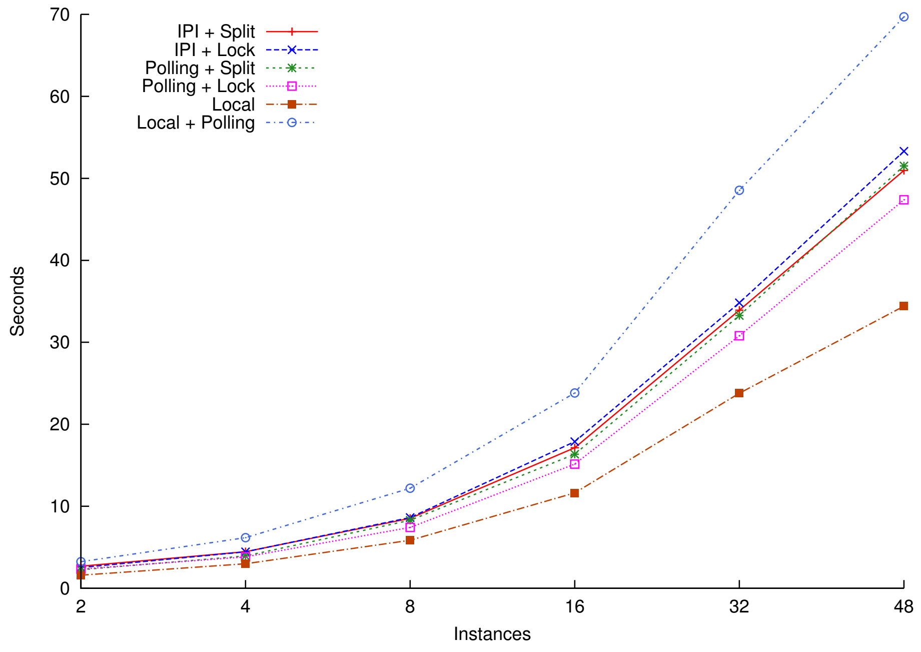


Figure 5.8: Many-to-One send() with 1000 messages on the SCC.

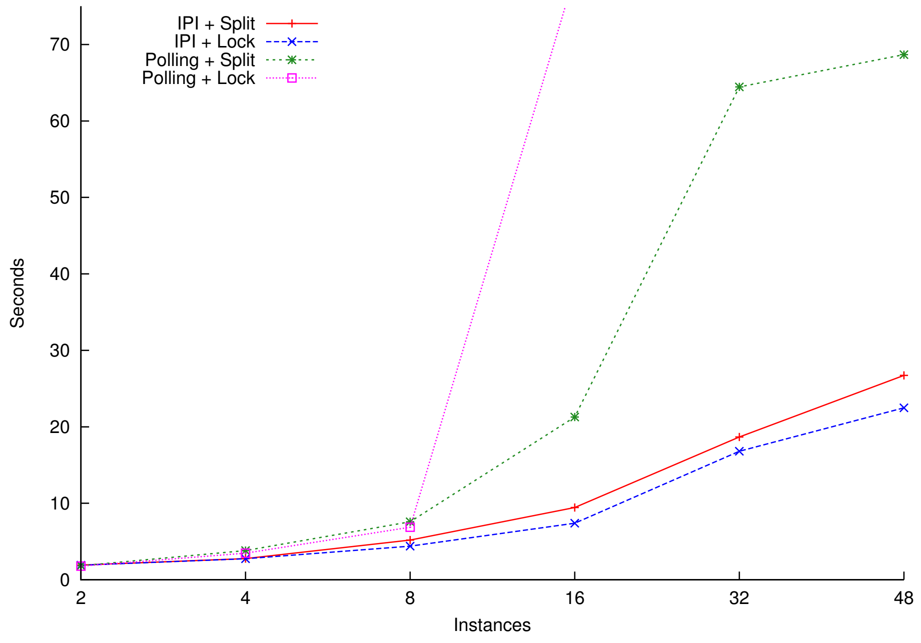


Figure 5.9: Many-to-One senda() with 1000 messages on the SCC.

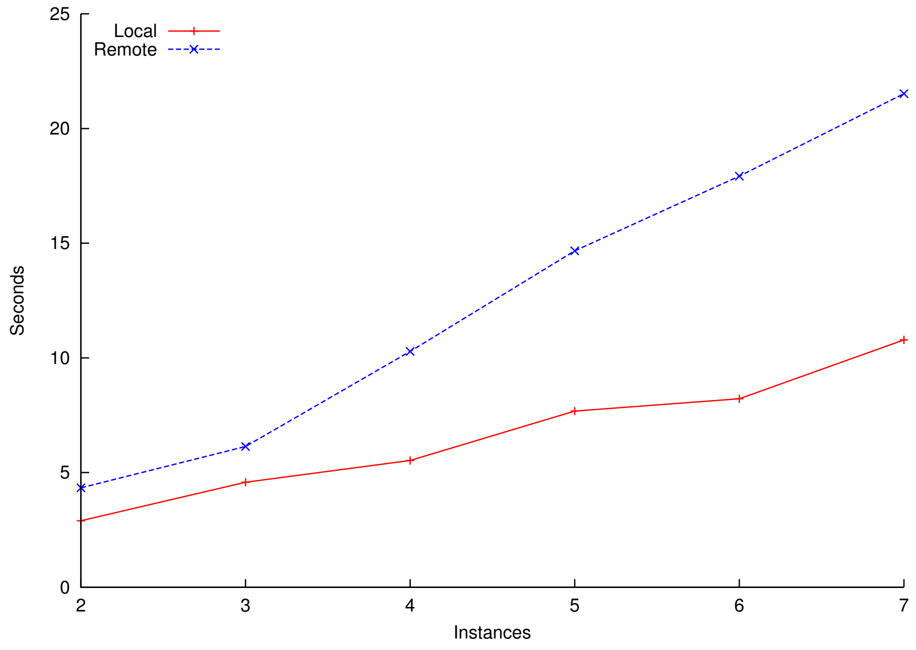


Figure 5.10: Many-to-One send() with 1 million messages on the Intel Xeon.

Chapter 6

Future Work

Our implementation provides a solid basis for future development and research on the SCC platform with MINIX. There are many things which could be improved or added. First, the implementation currently only supports publishing endpoints to all CPUs. With small changes to DS and libsys, it could be extended to support publishing to subsets of CPUs. This is useful when a process does not need to be visible on all cores. Additionally, unpublishing is needed to cleanup terminating processes.

While the MINIX instances on the SCC are currently able to run and communicate, they are still more or less separate instances. We envision future work includes transforming MINIX into a true manycore operating system where the MINIX instances together form a single entity. This requires choosing which server processes should run on which cores in the SCC, and how many instances. For example, there could be an PM instance on every core, but PM would then need to synchronize when performing updates to PM's internal process table. This is especially relevant when creating new processes, as it should not be possible for PM instances to choose the same PID when forking at the same time. Another way is to have a single PM instance in the SCC, which manages all processes system wide. A potential problem with this design is that the single PM instance could become a performance bottleneck under high system load. The same choice applies to other servers, such as VM, RS and VFS. In the case of VM it would be difficult to have a single instance, as VM currently assumes it can manage and modify all memory directly, including page tables, which is a problem for the cache incoherent SCC. RS should be able to run as a single instance, as it can send keep alive messages to global endpoints of servers. However, care must be taken that RS also receives signal messages when a remote process crashes.

Another interesting addition would be support for migrating processes between cores. Migration is useful to balance the load in the SCC. When migrating a process, all cores which have a stub installed should receive a message to update the appropriate fields, such as the coreid. Additionally, migration requires copying large chunks of memory, which may not be trivial on the SCC [54]. One way to implement bulk memory copy is to use uncached shared memory, but this will probably be slow. Additionally, in case every core has an instance of VM, the VM server must update memory mapping information when a process is migrated.

Another challenge for future work is to parallelize the server processes in MINIX. Currently the server processes work together in a sequential fashion. When handling service requests from applications, the server processes are often blocked waiting on each other. This should be avoided in manycore systems to achieve the best performance. Possible ways to parallelize the servers is to use asynchronous messaging instead or implement multiple threads in each server which can use blocking IPC. Finally, it would be good to have a better system console in MINIX on the SCC, for example a graphical frame buffer console similar to the Linux implementation.

Chapter 7

Related Work

Other research groups are currently doing similar work on the SCC. First, the Barrelfish [55] operating system implements support for the SCC. In our work we successfully applied their multikernel design principles to MINIX. In general the Barrelfish project is mostly about exploring the possibilities of multikernel designs where MINIX is more concerned with the benefits of microkernels, especially fault tolerance. Second, Intel provides Linux as the default operating system for the SCC. Since Linux is a monolithic operating system, it is only partly relevant for our work. When implementing the new boot code for MINIX, the Linux SCC source code was useful to extend our knowledge of the SCC where the documentation proved unsatisfying. Finally, researchers from the Aachen University in Germany are working on MetalSVM [56], a fast inter-kernel communication and synchronization layer. MetalSVM implements a hypervisor on the SCC mainly for experiments with distributed shared memory. On a later stage, MetalSVM could become more relevant for our work with MINIX.

Similar work on many core operating systems include Corey [57], which argues that applications should control sharing data in the OS. In Corey applications can control sharing of three abstractions: address ranges, kernel cores and shares. The LavA [58] project is concerned with many core embedded operating systems and argue that when sufficient chip space is available, a CPU can be dedicated to a single task. This avoids the need of scheduling, context switching and memory protection completely. Additionally, it is easier to meet realtime constraints if only a single task is executed per CPU. Moreover, a waiting CPU can be stopped completely until the arrival of an external interrupt. The factored operating system (FOS) [59] attempts to achieve high scalability with message passing and groups of server processes, inspired by the distributed design of internet services. Each group of server processes implements a system service, for example the file system or process manager. The server groups work closely together to balance the load. Furtermore, FOS provides a single system image to applications and supports the same elasticity of cloud computing platforms. Finally, the Tessellation OS [60] argues for space-time partitioning (STP). STP divides resources such as cores, cache, and network bandwidth amongst interacting software components. The components can use the resources in an application specific fashion and interact with each other by message passing.

Chapter 8

Conclusion

In this work we implemented MINIX on the SCC. While we needed to introduce several major changes to MINIX, the overall design of MINIX persisted. Our work shows that microkernels can function in cache incoherent environments and potentially scale to a large number of cores. The SCC has been a helpful platform for experiments. Although it does not provide solid debugging functionality for operating system developers, which made booting MINIX a time costly process, the SCC shows much potential for providing a scalable hardware platform. The small size of the MPB is not a problem for MINIX, nor is its cache incoherent nature. We were able to evaluate different message passing mechanisms, each showing different behaviour under various workloads. Additionall, we learned that our messaging implementation on the SCC performs excellent compared to local only messaging and MINIX SMP. With the performance results we conclude that IPIs are fast enough compared to polling mode and in some cases even faster. Overall, Lock Sharing the MPB showed the best performance compared to a Split Shared MPB design. Although more research is needed to transform MINIX in a mature manycore operating system, we provided a solid basis for further experimentation on the SCC. As Intel and other manufactures continue to improve parallelism in microprocessors, we software designers continue to improve scaling the software, which should eventually evolve manycore computer systems in reliable, high quality products.

Bibliography

[1] James E. Thornton. Considerations in Computer Design Leading up to the Control Data 6600. Technical report, Control Data Corporation, 1964.

[2] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 19 1965.

[3] Shekhar Y. Borkar, Hans Mulder, Pradeep Dubey, Stephen S. Pawlowski, Kevin C. Kahn, Justin R. Rattner, David J. Kuck, R. M. Ramanathan, and Vince Thomas. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. Technical report, Intel Corporation, `http://www.intel.com/go/platform2015`, 2005.

[4] Philip Machanick. Approaches to Addressing the Memory Wall. Technical report, School of IT and Electrical Engineering, Brisbane, QLD 4072, Australia, 2002.

[5] Sally A. McKee. CF'04. In *Reflections on the Memory Wall*, Ischia, Italy, April 14-16 2004. ACM.

[6] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded SPARC Processor. *Micro, IEEE*, 25(2), March-April 2005.

[7] Intel Labs. SCC External Architecture Specification (EAS) Revision 0.94. Technical report, Intel Corporation, May 2010.

[8] Andi Kleen. Linux Multi-core Scalability. In *Linux Kongress*. Intel Corporation, Germany, 2009.

[9] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, Amsterdam, Netherlands, 3rd edition, 2006.

[10] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: ten years later. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 305–318, New York, NY, USA, 2011. ACM.

[11] Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. We Crashed, Now What? In *Proceedings of the 6th International Workshop on Hot Topics in System Dependability*, 2010.

[12] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the EuroSys Conference*, April 2009.

[13] Jochen Liedtke. Towards Real u-Kernels, 1996.

[14] MIPS Technologies Incorporated. MIPS32 1074K Processor Core Family. Technical report, 955 East Arques Avenue, Sunnyvale, CA 94085, United States, 2011.

[15] Andrew Tanenbaum, Raja Appuswamy, Herbert Bos, Lorenzo Cavallaro, Cristiano Giuffrida, Tomáš Hrubý, Jorrit Herder, Erik van der Kouwe, and David van Moolenbroek. MINIX 3: Status Report and Current Research. *;login: The USENIX Magazine*, 35(3), June 2010.

[16] Jorrit N. Herder. *Building A Dependable Operating System: Fault Tolerance In MINIX 3.* De Nederlandse Organisatie voor Wetenschappelijk Onderzoek, Amsterdam, Netherlands, 2010.

[17] The Open Group. IEEE Std 1003.1: POSIX.1-2008. `http://www.opengroup.org/onlinepubs/9699919799/`, December 2008.

[18] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. Technical report, 2200 Mission College Blvd. Santa Clara, CA 94054-1549, USA, February 2008. Section 3.6: Paging (Virtual Memory) Overview.

[19] Intel Corporation. 82C54 CHMOS Programmable Interval Timer. Technical report, 2200 Mission College Blvd. Santa Clara, CA 94054-1549, USA, October 1994.

[20] Bjorn P. Swift. Individual Programming Assignment User Mode Scheduling in MINIX 3. Technical report, Vrije Universiteit Amsterdam, 2010.

[21] Intel Corporation. 8259A Programmable Interrupt Controller (8259A 8259A-2). Technical report, 2200 Mission College Blvd. Santa Clara, CA 94054-1549, USA, December 1988.

[22] Niek Linnenbank. Journaling Support in MINIX 3. Technical report, Vrije Universiteit Amsterdam, February 2011.

[23] PCI-SIG. PCI Local Bus Specification Revision 3.0. Technical report, 5440 SW Westgate Drive, Suite 217, Portland, Oregon 97221, February 2004.

[24] Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291. Transmission Control Protocol DARPA Internet Program Protocol Specification. Technical report, Defense Advanced Research Projects Agency, Information Processing Techniques Office, 1400 Wilson Boulevard, Arlington, Virginia 22209, September 1981.

[25] The ethernet: a local area network: data link layer and physical layer specifications. *SIGCOMM Comput. Commun. Rev.*, 11:20–66, July 1981.

[26] OpenSSH Community. OpenSSH Project Home Page. `http://www.openssh.org`, August 2011.

[27] Niek Linnenbank. Implementing the Intel Pro/1000 on MINIX 3. Technical report, Vrije Universiteit Amsterdam, December 2009.

[28] Colin Fowler. The Minix3 Tigon III (TG3) Driver. Technical report, Vrije Universiteit Amsterdam, May 2010.

[29] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. Technical report, 2200 Mission College Blvd. Santa Clara, CA 94054-1549, USA, February 2008. Chapter 8: Processor Management and Initialization.

[30] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. Technical report, 2200 Mission College Blvd. Santa Clara, CA 94054-1549, USA, February 2008. Chapter 9: Advanced Programmable Interrupt Controller (APIC).

[31] Intel Corporation. 82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC). Technical report, 2200 Mission College Blvd. Santa Clara, CA 94054-1549, USA, May 1996.

[32] Jim Held. Single-chip Cloud Computer: An experimental many-core processor from Intel Labs. In *Single-chip Cloud Computer Symposium*, page 3. Intel Labs, February 10 2010.

[33] Alexander Arlt, Jan H. Schonherr, and Jan Richling. Meta-programming Many-Core Systems. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Communication and Operating Systems Group, Technische Universiteit Berlin, Germany, July 5-6 2011.

[34] Andreas Prell and Thomas Rauber. Task Parallelism on the SCC. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Department of Computer Science, University of Bayreuth, Germany, July 5-6 2011.

[35] Nils Petersen, Julian Pastarmov, and Didier Stricker. ARGOS - a software framework to facilitate user transparent multi-threading. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* DFKI GmbH and Google Germany, July 5-6 2011.

[36] Wasuwee Sodsong and Bernd Burgstaller. A Fast Fourier Transformation Algorithm for Single-Chip Cloud Computers Using RCCE. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Department of Computer Science, Yonsei University, Seoul, Korea, July 5-6 2011.

[37] Merijn Verstraaten, Clemens Grelck, Michiel W. van Tol, Roy Bakker, and Chris R. Jesshope. On Mapping Distributed S-NET to the 48-core Intel SCC Processor. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Informatics Institute, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands, July 5-6 2011.

[38] Steffen Christgau, Bettina Schnor, and Simon Kiertscher. The Benefit of Topology-Awareness of MPI Applications on the SCC. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Institute of Computer Science, University of Potsdam, August-Bebel-Strasse 89, 14482 Potsdam, Germany. Potsdam Institute for Climate Impact Research, P.O. Box 60 12 03, 14412 Potsdam, Germany, July 5-6 2011.

[39] Simon Peter, Adrian Schpbach, Dominik Menzi, and Timothy Roscoe. Early experience with the Barrelfish OS and the Single-Chip Cloud Computer. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Systems Group, Department of Computer Science, ETH Zurich, July 5-6 2011.

[40] Jan-Arne Sobania, Peter Troger, and Andreas Polze. Linux Operating System Support for the SCC Platform - An Analysis. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Hasso Plattner Institute, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany, July 5-6 2011.

[41] Panayiotis Petrides, Andreas Diavastos, and Pedro Trancoso. Exploring Database Workloads on Future Clustered Many-Core Architectures. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Department of Computer Science, University of Cyprus, July 5-6 2011.

[42] Stephan-Alexander Posselt Bjorn Saballus and Thomas Fuhrmann. A Scalable and Robust Runtime Environment for SCC Clusters. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Technische Universitat Munchen, Boltzmannstrasse 3, 85748 Garching/-Munich, Germany, July 5-6 2011.

[43] Anastasios Papagiannis and Dimitrios S. Nikolopoulos. Scalable Runtime Support for Data-Intensive Applications on the Single-Chip Cloud Computer. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Institute of Computer Science (ICS), Foundation for Research and Technology Hellas (FORTH), GR70013, Heraklion, Crete, GREECE, July 5-6 2011.

[44] Florian Thoma, Michael Hubner, Diana Gohringer, Hasan Umitcan Yilmaz, and Jurgen Becker. Power and performance optimization through MPI supported dynamic voltage and frequency scaling. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Karlsruhe Institute of Technology (KIT), Germany. Fraunhofer IOSB, Ettlingen, Germany, July 5-6 2011.

[45] Pedro Alonso, Manuel F. Dolz, Francisco D. Igual, Bryan Marker, Rafael Mayo, Enrique S. Quintana-Ort, and Robert A. van de Geijn. Power-aware Dense Linear Algebra Implementations on Multi-core and Many-core Processors. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Univ. Politecnica de Valencia, 46.022 - Valencia, Spain. Univ. Jaume I de Castellon, 12.071 - Castellon, Spain. The University of Texas at Austin, TX 78712, July 5-6 2011.

[46] Intel Corporation. Terascale Computing Research Program. `http://techresearch.intel.com/ResearchAreaDetails.aspx?Id=27`, August 2011.

[47] Intel Corporation. Manycore Applications Research Community. `http://techresearch.intel.com/ProjectDetails.aspx?Id=1`, August 2011.

[48] Intel Corporation. Pentium Processor Family Developers Manual Volume 3: Architecture and Programming Manual. Technical report, 2200 Mission College Blvd. Santa Clara, CA 94054-1549, USA, 1995.

[49] Intel Labs. The L2 Cache on the SCC. Technical report, 2200 Mission College Blvd. Santa Clara, CA 94054-1549, USA, November 2010.

[50] Intel Labs. SCC External Architecture Specification (EAS) Revision 0.94. Technical report, Intel Corporation, May 2010. Section 4.2.6 MIU (Mesh Interface Unit).

[51] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schupbach, 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.* Systems Group, ETH Zurich. Microsoft Research, Cambridge. ENS Cachan Bretagne, ACM, October 11-14 2009.

[52] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. Technical report, 2200 Mission College Blvd. Santa Clara, CA 94054-1549, USA, February 2008. 3.2 Using Segments.

[53] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. Technical report, 2200 Mission College Blvd. Santa Clara, CA 94054-1549, USA, February 2008. 2.2 Modes of Operation.

[54] Michiel W. van Tol, Roy Bakker, Merijn Verstraaten, Clemens Grelck, and Chris R. Jesshope. Efficient Memory Copy Operations on the 48-core Intel SCC Processor. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Informatics Institute, University of Amsterdam, Sciencepark 904, 1098 XH Amsterdam, The Netherlands, July 5-6 2011.

[55] Systems Group, ETH Zurich. Barrelfish Operating System Homepage. `http://www.barrelfish.org`, February 2009.

[56] Pablo Reble, Stefan Lankes, Carsten Clauss, and Thomas Bemmerl. A Fast Inter-Kernel Communication and Synchronization Layer for MetalSVM. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Chair for Operating Systems, RWTH Aachen University, Kopernikusstr. 16, 52056 Aachen, Germany, July 5-6 2011.

[57] Silas B. Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[58] Michael Engel, Matthias Meier, and Olaf Spinczyk. LavA: An Embedded Operating System for the Manycore Age. Technical report, Embedded System Software, Technische Universitat Dortmund, 2009.

[59] A Unified Operating System for Clouds and Manycore: fos. Technical report, David Wentzlaff and Charles Gruenwald III and Nathan Beckmann and Kevin Modzelewski and Adam Belay and Lamia Youseff and Jason Miller and Anant Agarwal, 25-27 January 2010.

[60] Tessellation: Space-Time Partitioning in a Manycore Client OS. Technical report, 2010.

Appendix A

Default LUTs

LUT #	**Physical Address**	**Description**
255	FFFFFFFF - FF000000	Private
254	FEFFFFFF - FE000000	N/A
253	FDFFFFFF - FD000000	N/A
252	FCFFFFFF - FC000000	N/A
251	FBFFFFFF - FB000000	VRC
250	FAFFFFFF - FA000000	Management Console TCP/IP Interface
249	F9FFFFFF - F9000000	N/A
248	F8FFFFFF - F8000000	N/A
247	F7FFFFFF - F7000000	System Configuration Registers Tile 23
246	F6FFFFFF - F6000000	System Configuration Registers Tile 22
245	F5FFFFFF - F5000000	System Configuration Registers Tile 21
244	F4FFFFFF - F4000000	System Configuration Registers Tile 20
243	F3FFFFFF - F3000000	System Configuration Registers Tile 19
242	F2FFFFFF - F2000000	System Configuration Registers Tile 18
241	F1FFFFFF - F1000000	System Configuration Registers Tile 17
240	F0FFFFFF - F0000000	System Configuration Registers Tile 16
239	EFFFFFFF - EF000000	System Configuration Registers Tile 15
238	EEFFFFFF - EE000000	System Configuration Registers Tile 14
237	EDFFFFFF - ED000000	System Configuration Registers Tile 13
236	ECFFFFFF - EC000000	System Configuration Registers Tile 12
235	EBFFFFFF - EB000000	System Configuration Registers Tile 11
234	EAFFFFFF - EA000000	System Configuration Registers Tile 10
233	E9FFFFFF - E9000000	System Configuration Registers Tile 9
232	E8FFFFFF - E8000000	System Configuration Registers Tile 8
231	E7FFFFFF - E7000000	System Configuration Registers Tile 7
230	E6FFFFFF - E6000000	System Configuration Registers Tile 6
229	E5FFFFFF - E5000000	System Configuration Registers Tile 5
228	E4FFFFFF - E4000000	System Configuration Registers Tile 4
227	E3FFFFFF - E3000000	System Configuration Registers Tile 3
226	E2FFFFFF - E2000000	System Configuration Registers Tile 2
225	E1FFFFFF - E1000000	System Configuration Registers Tile 1
224	E0FFFFFF - E0000000	System Configuration Registers Tile 0
223	DFFFFFFF - DF000000	N/A
...	...	...

LUT #	**Physical Address**	**Description**
216	D8FFFFFF - D8000000	N/A
215	D7FFFFFF - D7000000	MPB in Tile 23
214	D6FFFFFF - D6000000	MPB in Tile 22
213	D5FFFFFF - D5000000	MPB in Tile 21
212	D4FFFFFF - D4000000	MPB in Tile 20
211	D3FFFFFF - D3000000	MPB in Tile 19
210	D2FFFFFF - D2000000	MPB in Tile 18
209	D1FFFFFF - D1000000	MPB in Tile 17
208	D0FFFFFF - D0000000	MPB in Tile 16
207	CFFFFFFF - CF000000	MPB in Tile 15
206	CEFFFFFF - CE000000	MPB in Tile 14
205	CDFFFFFF - CD000000	MPB in Tile 13
204	CCFFFFFF - CC000000	MPB in Tile 12
203	CBFFFFFF - CB000000	MPB in Tile 11
202	CAFFFFFF - CA000000	MPB in Tile 10
201	C9FFFFFF - C9000000	MPB in Tile 9
200	C8FFFFFF - C8000000	MPB in Tile 8
199	C7FFFFFF - C7000000	MPB in Tile 7
198	C6FFFFFF - C6000000	MPB in Tile 6
197	C5FFFFFF - C5000000	MPB in Tile 5
196	C4FFFFFF - C4000000	MPB in Tile 4
195	C3FFFFFF - C3000000	MPB in Tile 3
194	C2FFFFFF - C2000000	MPB in Tile 2
193	C1FFFFFF - C1000000	MPB in Tile 1
192	C0FFFFFF - C0000000	MPB in Tile 0
191	BFFFFFFF - BF000000	N/A
...	...	...
132	84FFFFFF - 84000000	N/A
131	83FFFFFF - 83000000	Shared MCH3
130	82FFFFFF - 82000000	Shared MCH2
129	81FFFFFF - 81000000	Shared MCH1
128	80FFFFFF - 80000000	Shared MCH0
127	7FFFFFFF - 7F000000	N/A
...	...	...
42	2AFFFFFF - 2A000000	N/A
41	29FFFFFF - 29000000	N/A
40	28FFFFFF - 28000000	Private
39	27FFFFFF - 27000000	Private
...	...	...
1	01FFFFFF - 01000000	Private
0	00FFFFFF - 00000000	Private

Appendix B

Kernel Process Table Structure

```
struct proc {
  struct stackframe_s p_reg;      /* process' registers saved in stack frame */
  struct fpu_state_s p_fpu_state;        /* process' fpu_regs saved lazily */
  struct segframe p_seg;          /* segment descriptors */
  proc_nr_t p_nr;                 /* number of this process (for fast access) */
  struct priv *p_priv;            /* system privileges structure */
  u32_t p_rts_flags;              /* process is runnable only if zero */
  u32_t p_misc_flags;             /* flags that do not suspend the process */

  char p_priority;                /* current process priority */
  u64_t p_cpu_time_left;          /* time left to use the cpu */
  unsigned p_quantum_size_ms;     /* assigned time quantum in ms
                                     FIXME remove this */
  struct proc *p_scheduler;       /* who should get out of quantum msg */
  unsigned p_cpu;                 /* what CPU is the process running on */
#ifdef CONFIG_SMP
  bitchunk_t p_cpu_mask[BITMAP_CHUNKS(CONFIG_MAX_CPUS)]; /* what CPUs is hte
                                                            process allowed to
                                                            run on */
#endif

  /* Accounting statistics that get passed to the process' scheduler */
  struct {
        u64_t enter_queue;        /* time when enqueued (cycles) */
        u64_t time_in_queue;      /* time spent in queue */
        unsigned long dequeues;
        unsigned long ipc_sync;
        unsigned long ipc_async;
        unsigned long preempted;
  } p_accounting;

  struct mem_map p_memmap[NR_LOCAL_SEGS];    /* memory map (T, D, S) */

  clock_t p_user_time;            /* user time in ticks */
  clock_t p_sys_time;             /* sys time in ticks */

  clock_t p_virt_left;            /* number of ticks left on virtual timer */
  clock_t p_prof_left;            /* number of ticks left on profile timer */

  u64_t p_cycles;                 /* how many cycles did the process use */

  struct proc *p_nextready;       /* pointer to next ready process */
  struct proc *p_caller_q;        /* head of list of procs wishing to send */
  struct proc *p_q_link;          /* link to next proc wishing to send */
  endpoint_t p_getfrom_e;         /* from whom does process want to receive? */
  endpoint_t p_sendto_e;          /* to whom does process want to send? */
  sigset_t p_pending;             /* bit map for pending kernel signals */

  char p_name[P_NAME_LEN];        /* name of the process, including \0 */

  endpoint_t p_endpoint;          /* endpoint number, generation-aware */

  message p_sendmsg;              /* Message from this process if SENDING */
  message p_delivermsg;           /* Message for this process if MF_DELIVERMSG */
  vir_bytes p_delivermsg_vir;     /* Virtual addr this proc wants message at */

  /* If handler functions detect a process wants to do something with
   * memory that isn't present, VM has to fix it. Until it has asked
   * what needs to be done and fixed it, save necessary state here.
   *
   * The requestor gets a copy of its request message in reqmsg and gets
   * VMREQUEST set.
   */
  struct {
        struct proc     *nextrestart;  /* next in vmrestart chain */
        struct proc     *nextrequestor; /* next in vmrequest chain */
#define VMSTYPE_SYS_NONE        0
#define VMSTYPE_KERNELCALL      1
#define VMSTYPE_DELIVERMSG      2
#define VMSTYPE_MAP             3

        int             type;           /* suspended operation */
        union {
                /* VMSTYPE_SYS_MESSAGE */
                message         reqmsg; /* suspended request message */
        } saved;

        /* Parameters of request to VM */
        int             req_type;
        endpoint_t      target;
        union {
                struct {
                        vir_bytes       start, length;  /* memory range */
                        u8_t            writeflag;      /* nonzero for
                } check;                                 * write access */
                struct {
                        char            writeflag;
                        endpoint_t      ep_s;
                        vir_bytes       vir_s, vir_d;
                        vir_bytes       length;
                } map;
        } params;

        /* VM result when available */
        int             vmresult;

        /* If the suspended operation is a sys_call, its details are
         * stored here.
         */
  } p_vmrequest;

  int p_found;  /* consistency checking variables */
  int p_magic;          /* check validity of proc pointers */

#if DEBUG_TRACE
  int p_schedules;
#endif
};
```

Appendix C

Developers Guide for MINIX on the SCC

This guide explains how to use the MINIX implementation for the SCC. It is assumed that the reader is familiar with the command line and that the SCC and MCPC available and installed. Section C.1 explains how to build the MINIX OS image which can be uploaded and ran on the SCC. Section C.2 and C.3 describe step by step how to initialize the SCC and how to start MINIX using sccKit commands. Finally, Section C.4 explains how to do optional configuration of the MINIX implementation.

C.1 Building

The first step is to build the MINIX OS image. Assuming the source code is available in the **minix-scc.git** directory, the following commands compile MINIX and generate the **boot_scc/image.obj** image:

```
$ cd /path/to/minix-scc.git
$ make clean
$ make all
```

After **make all** completes, look in the **boot_scc** directory and check if the **image.obj** exists. It should be about 3MB large. On success, you may continue to the next step in Section C.3 to boot MINIX on the SCC.

C.2 Initialization

Running MINIX on the SCC requires two steps. First, the SCC must be powered on and initialized. The initialization must be done only before any program is able to run on the SCC and only needs to be done once. Second, the sccKit commands should be used to upload the MINIX OS image to the SCC and trigger a global CPU reset such that the cores start booting MINIX. If applicable, connect to the MCPC via SSH. Be sure to use the **-XC** arguments if you intent to use the **sccGui** command to boot MINIX (recommended):

```
niek@ubuntu$ ssh -p2222 -XC niekl@mcpc
```

To power on the SCC, you must connect to the on-board BMC via telnet. This is a small ARM board to manage the SCC. Give the **telnet_bmc** command to connect. You should see the following text on success:

```
niekl@arrakis:~/minix-scc.git$ telnet_bmc
Trying 192.168.2.127...
```

```
Connected to bmc.
Escape character is '^]'.
********************************************************************************

  ---- ---- ---- -  - -   -    -    ---- -  - ----    ---  ---- ---- ---- ---
  |__/ |  | |    |_/   \_/     |    |__| |_/  |___    |__] |  | |__| |__/ |  \
  |  \ |__| |___ | \_   |      |___ |  | | \_ |___    |__] |__| |  | |  \ |__/

     ---- ---- ----          ---- - -  - ---- -    ----    ---- -  - - ---
     [__  |    |       __    [__  | |\ | | __ |    |___ __ |    |__| | |__]
     ___] |___ |___          ___] | | \| |__] |___ |___    |___ |  | | |
     ---- -    ---- -  - ---     ---- ---- -  - ---  -  - --- - -  - ----
     |    |    |  | |  | |  \    |    |  | |\/| |__] |  |  |  | |\ | | __
     |___ |___ |__| |__| |__/    |___ |__| |  | |    |__|  |  | | \| |__]


********************************************************************************

 Copyright 2010 by Intel Corporation
  Intel Labs - Germany Microprocessor Lab

  Board Serial# 01095100089
  Usable GB ETH 0110
  Software:     1.10  Build: 1228  Oct 12 2010  18:18:01
  CPLD:         1.07
  HW-ID:        0x00
  POWR1220:     0xC0000001 (master), 0x40000001 (slave)
  DDR3 modules: Present: 0 1 2 3 4 5 6 7

Welcome to the BMC server of rocky_lake_board!
You are participant #1
]>
```

To turn on the SCC, you need to enter the **power on** command. Respectively, **power off** can be used to turn off the SCC. On success, you should see the following text:

```
]>power on
----------------------------------------------------------------------------
Switching on the board.
Initializing OPVRs
Downloading FPGA bitstream
SelectMap: Processing bitstream file "/mnt/flash4/rl_20110308_ab.bit".
           Filesize is 3889967 bytes.
Done.
Switching OPVRs on.
JTAG INIT
Resetting SCC JTAG interface.
JTAG READ LEFT PLL
Select JCU: JTAG_JCU_LEFT (PLL)!
Read Chain: 040F8002200A
END
JTAG WRITE LEFT PLL 008000A280A
JTAG WRITE RIGHT SYSIF 0A100E000F8E
PLL lock LED is ON.
```

```
JTAG WRITE MC0 AFE 001203201D8811800000F0012200000200000B000C70800E0
JTAG WRITE MC1 AFE 001203201D8811800000F0012200000200000B000C70800E0
JTAG WRITE MC2 AFE 001203201D8811800000F0012200000200000B000C70800E0
JTAG WRITE MC3 AFE 001203201D8811800000F0012200000200000B000C70800E0
JTAG WRITE MC0 RST FF
JTAG WRITE MC1 RST FF
JTAG WRITE MC2 RST FF
JTAG WRITE MC3 RST FF
JTAG WRITE MC0 RST 00
JTAG WRITE MC1 RST 00
JTAG WRITE MC2 RST 00
JTAG WRITE MC3 RST 00
PHY INIT
Done.
----------------------------------------------------------------------------
]>^]

telnet> close
Connection closed.
niekl@arrakis:~/minix-scc.git$
```

At this point the SCC is in a powered on state, but it is not yet initialized. You must use the **sccBmc** command with the **-i** option to do this, or the SCC will not work. When you execute **sccBmc -i**, you should see the following text:

```
niekl@arrakis:~/minix-scc.git$ sccBmc -i
INFO: openBMCConnection(192.168.2.127:5010): You are participant #1
INFO: Packet tracing is disabled...
INFO: Initializing System Interface (SCEMI setup)....
INFO: Successfully connected to PCIe driver...
INFO: Welcome to sccBmc 1.4.0 (build date Mar 21 2011 - 18:42:49)...
INFO: This tool allows you to (re-)initialize the SCC platform.
This means that the reset of the actual SCC device is triggered and
that all clock settings are programmed from scratch. The (re-)
programming of the clock settings also implies a training of the
physical System Interface!

Short said: The whole procedure takes a while and you should only do
it when necessary! This step is NOT always required after starting
the GUI. You would normally invoke it when the system reset executes
with errors or when the board has just been powered up...

Please select from the following possibilities:
INFO: (0) Tile533_Mesh800_DDR800
INFO: (1) Tile800_Mesh1600_DDR1066
INFO: (2) Tile800_Mesh1600_DDR800
INFO: (3) Tile800_Mesh800_DDR1066
INFO: (4) Tile800_Mesh800_DDR800
INFO: (others) Abort!
Make your selection:
```

Here **sccBmc** asks for the preferred configuration. Choose option **(1)** here. The MINIX implementation is configuration by default for a tile speed of 800MHz. You must choose the same tile frequency setting as configured in MINIX, as otherwise the clock timing code would be inaccurate. See Section C.4 for more details on changing configuration parameters in MINIX. After choosing the configuration, the following output should be shown on success:

```
INFO: Starting system initialization (with setting Tile800_Mesh1600_DDR1066)...
processRCCFile(): Configuring SCC with content of file "/opt/sccKit/1.4.0/"
INFO: Trying to train in preset mode:
INFO: Resetting Rocky Lake SCC device: Done.
INFO: Resetting Rocky Lake FPGA: Done.
INFO: Re-Initializing PCIe driver...
INFO: FPGA version register value: 0x20110308
INFO: Loading SIF system settings: RX Data delay is 27 and RX Clock delay is 46...
INFO: Programming done. Testing:
INFO: Enabling RC pattern generation + FPGA monitoring for current delay...
INFO: Enabling FPGA pattern generation + FPGA monitoring for current delay...
INFO: Done... Disabling pattern generation -> Good range of this run is 7...
INFO: Resetting Rocky Lake SCC device: Done.
INFO: Resetting Rocky Lake FPGA: Done.
INFO: Configuring memory controllers:
INFO: processRCCFile(): Configuring SCC with content of file "/opt/sccKit/1.4.0/"
INFO: [line 0014]  #####################################################
INFO: [line 0015]  ###     Rock Creek setup:                        ####
INFO: [line 0017]  ###         - Global Clock GC = 1600 MHz         ####
INFO: [line 0021]  ###         - Tile clock      =  800 MHz         ####
INFO: [line 0022]  ###         - Router clock    = 1600 MHz         ####
INFO: [line 0049]  ###         - SIF clock       =  266 MHz         ####
INFO: [line 0108]  ###         - DDR3-1066 7-7-7-20                 ####
INFO: [line 0164]  #####################################################
INFO: [line 0165]  ###     Rock Creek MC0 setup DDR3-1066 7-7-7-20    ####
INFO: [line 0166]  #####################################################
INFO: [line 0176]  ---- AFE Padscan init -------------------------------
INFO: [line 0239]  ---- AFE power on sequence --------------------------
INFO: [line 0272]  ---- Initial Compensation ---------------------------
INFO: [line 0298]  ---- Setup controller parameters ---------------------
INFO: [line 0900]  ---- Periodic Compensation --------------------------
INFO: [line 0926]  ---- Controller setup and SDRAM initialization -------
INFO: [line 0993]  #####################################################
INFO: [line 0994]  ###     Rock Creek MC1 setup DDR3-1066 7-7-7-20    ####
INFO: [line 0995]  #####################################################
INFO: [line 1005]  ---- AFE Padscan init -------------------------------
INFO: [line 1068]  ---- AFE power on sequence --------------------------
INFO: [line 1101]  ---- Initial Compensation ---------------------------
INFO: [line 1127]  ---- Setup controller parameters ---------------------
INFO: [line 1729]  ---- Periodic Compensation --------------------------
INFO: [line 1755]  ---- Controller setup and SDRAM initialization -------
INFO: [line 1822]  #####################################################
INFO: [line 1823]  ###     Rock Creek MC2 setup DDR3-1066 7-7-7-20    ####
INFO: [line 1824]  #####################################################
INFO: [line 1834]  ---- AFE Padscan init -------------------------------
INFO: [line 1897]  ---- AFE power on sequence --------------------------
INFO: [line 1930]  ---- Initial Compensation ---------------------------
INFO: [line 1956]  ---- Setup controller parameters ---------------------
```

```
INFO: [line 2558]  ---- Periodic Compensation ---------------------------
INFO: [line 2584]  ---- Controller setup and SDRAM initialization -------
INFO: [line 2651]  #######################################################
INFO: [line 2652]  ###     Rock Creek MC3 setup DDR3-1066 7-7-7-20    ####
INFO: [line 2653]  #######################################################
INFO: [line 2663]  ---- AFE Padscan init --------------------------------
INFO: [line 2726]  ---- AFE power on sequence ---------------------------
INFO: [line 2759]  ---- Initial Compensation ----------------------------
INFO: [line 2785]  ---- Setup controller parameters ---------------------
INFO: [line 3387]  ---- Periodic Compensation ---------------------------
INFO: [line 3413]  ---- Controller setup and SDRAM initialization -------
INFO: [line 3479]  ---- Start Normal Operation --------------------------
INFO: [line 3498]  ---- Write data --------------------------------------
INFO: [line 3541]  ---- Read data from MC0 rank 0 -----------------------------
INFO: [line 3542]  MEMRD INFO  MC0: Read data 34D865EF432C2AC6 729D177E3CD30EE6
INFO: [line 3543]  MEMRD INFO  MC0: Read data F4472F66280AD037 4957394751261518
INFO: [line 3544]  ---- Read data from MC0 rank 1 -----------------------------
INFO: [line 3545]  MEMRD INFO  MC0: Read data 657226DC28F56C3A 285F5179DE684FA0
INFO: [line 3546]  MEMRD INFO  MC0: Read data D54ACCA7421CD305 BE0E2B494EDA4C87
INFO: [line 3547]  ---- Read data from MC0 rank 2 -----------------------------
INFO: [line 3548]  MEMRD INFO  MC0: Read data 8309B135A9FD8430 4175B2E1DA0462D7
INFO: [line 3549]  MEMRD INFO  MC0: Read data 649BC11B02B4C008 D9358B5B461D5821
INFO: [line 3550]  ---- Read data from MC0 rank 3 -----------------------------
INFO: [line 3551]  MEMRD INFO  MC0: Read data 401839250FEAE7C0 56489C9ECFC5B485
INFO: [line 3552]  MEMRD INFO  MC0: Read data C60EB2EF13DDFF84 61288BF34DCA9F33
INFO: [line 3554]  ---- Read data from MC1 rank 0 -----------------------------
INFO: [line 3555]  MEMRD INFO  MC1: Read data 34D865EF432C2AC6 729D177E3CD30EE6
INFO: [line 3556]  MEMRD INFO  MC1: Read data F4472F66280AD037 4957394751261518
INFO: [line 3557]  ---- Read data from MC1 rank 1 -----------------------------
INFO: [line 3558]  MEMRD INFO  MC1: Read data 657226DC28F56C3A 285F5179DE684FA0
INFO: [line 3559]  MEMRD INFO  MC1: Read data D54ACCA7421CD305 BE0E2B494EDA4C87
INFO: [line 3560]  ---- Read data from MC1 rank 2 -----------------------------
INFO: [line 3561]  MEMRD INFO  MC1: Read data 8309B135A9FD8430 4175B2E1DA0462D7
INFO: [line 3562]  MEMRD INFO  MC1: Read data 649BC11B02B4C008 D9358B5B461D5821
INFO: [line 3563]  ---- Read data from MC1 rank 3 -----------------------------
INFO: [line 3564]  MEMRD INFO  MC1: Read data 401839250FEAE7C0 56489C9ECFC5B485
INFO: [line 3565]  MEMRD INFO  MC1: Read data C60EB2EF13DDFF84 61288BF34DCA9F33
INFO: [line 3567]  ---- Read data from MC2 rank 0 -----------------------------
INFO: [line 3568]  MEMRD INFO  MC2: Read data 34D865EF432C2AC6 729D177E3CD30EE6
INFO: [line 3569]  MEMRD INFO  MC2: Read data F4472F66280AD037 4957394751261518
INFO: [line 3570]  ---- Read data from MC2 rank 1 -----------------------------
INFO: [line 3571]  MEMRD INFO  MC2: Read data 657226DC28F56C3A 285F5179DE684FA0
INFO: [line 3572]  MEMRD INFO  MC2: Read data D54ACCA7421CD305 BE0E2B494EDA4C87
INFO: [line 3573]  ---- Read data from MC2 rank 2 -----------------------------
INFO: [line 3574]  MEMRD INFO  MC2: Read data 8309B135A9FD8430 4175B2E1DA0462D7
INFO: [line 3575]  MEMRD INFO  MC2: Read data 649BC11B02B4C008 D9358B5B461D5821
INFO: [line 3576]  ---- Read data from MC2 rank 3 -----------------------------
INFO: [line 3577]  MEMRD INFO  MC2: Read data 401839250FEAE7C0 56489C9ECFC5B485
INFO: [line 3578]  MEMRD INFO  MC2: Read data C60EB2EF13DDFF84 61288BF34DCA9F33
INFO: [line 3580]  ---- Read data from MC3 rank 0 -----------------------------
INFO: [line 3581]  MEMRD INFO  MC3: Read data 34D865EF432C2AC6 729D177E3CD30EE6
INFO: [line 3582]  MEMRD INFO  MC3: Read data F4472F66280AD037 4957394751261518
INFO: [line 3583]  ---- Read data from MC3 rank 1 -----------------------------
INFO: [line 3584]  MEMRD INFO  MC3: Read data 657226DC28F56C3A 285F5179DE684FA0
```

```
INFO: [line 3585]  MEMRD INFO  MC3: Read data D54ACCA7421CD305 BE0E2B494EDA4C87
INFO: [line 3586]  ---- Read data from MC3 rank 2 ----------------------------
INFO: [line 3587]  MEMRD INFO  MC3: Read data 8309B135A9FD8430 4175B2E1DA0462D7
INFO: [line 3588]  MEMRD INFO  MC3: Read data 649BC11B02B4C008 D9358B5B461D5821
INFO: [line 3589]  ---- Read data from MC3 rank 3 ----------------------------
INFO: [line 3590]  MEMRD INFO  MC3: Read data 401839250FEAE7C0 56489C9ECFC5B485
INFO: [line 3591]  MEMRD INFO  MC3: Read data C60EB2EF13DDFF84 61288BF34DCA9F33
INFO: [line 3595]  INFO: CCF enabled in normal op mode
INFO: [line 3596]  INFO: CCF enabled in normal op mode
INFO: [line 3597]  INFO: CCF enabled in normal op mode
INFO: [line 3598]  INFO: CCF enabled in normal op mode
INFO: [line 3600]  ---- Rock Creek setup DONE --------------------------
INFO: (Re-)configuring GRB registers...
niekl@arrakis:~/minix-scc.git$
```

Once **sccBmc** terminates, the SCC initialization is completed. You only need to do these steps once and not every time you boot a new tryout image.

C.3 Booting

After the SCC initialization has run, it is possible to boot a MINIX OS image on the SCC. In the following example, we start MINIX on cores 0 and 1, each with their own **Ipctest** server instance, where core 0 sends 10000 synchronous messages to core 1.

First, reset all cores in the SCC to a known state:

```
niekl@arrakis:~/minix-scc.git$ sccReset -g
INFO: Welcome to sccReset 1.4.0 (build date Mar 21 2011 - 18:40:25)...
INFO: Applying global software reset to SCC (cores & CRB registers)...
INFO: (Re-)configuring GRB registers...
```

Now we start the **sccGui** program. This is a graphical interface which can be used to upload OS images to the SCC and capture output from the cores:

```
niekl@arrakis:~/minix-scc.git$ sccGui
```

When **sccGui** starts, it should open an new window. If this is the first time you boot a MINIX image, you need to tell **sccGui** where to find the MINIX OS image. Click on **Settings** – **Linux Boot Settings** – **Choose Custom Linux Image (File Dialog)**. A new window should open. Select the **boot_scc/image.obj** from your **minix-scc.git** directory and click **OK**.

Additionally, if this is your first time with **sccGui**, you need to configure **sccGui** to capture writes to the serial port. Click on **Settings** – **Debug Settings** – **Enable Software RAM and UART**.

You are now ready to boot MINIX. Before you boot MINIX, all MPBs must be cleared to zero. You can do this manually by clicking on **Tools** – **Clear MPB(s)** – **OK**. Alternatively, you can configure **sccGui** to always clear MPBs when you boot a new image with: **Settings** – **Linux Boot Settings** – **Clear MPB(s) before Booting**.

Boot the MINIX OS image by clicking on **Tools** – **Boot Linux**. Select the appropriate cores to start, in our example core 0 and 1, and click on **OK**. A new window should popup quickly showing the MINIX boot output:

```
CPU 0 freq 800 MHz
APIC timer calibrated
Boot cpu apic id 0
APIC debugging is enabled
Initiating APIC timer handler

MINIX 3.1.8bs @ Aug 10 2011 23:47:51 [ SCC APIC ]
Copyright 2010, Vrije Universiteit, Amsterdam, The Netherlands
MINIX is open source software, see http://www.minix3.org
Build arrakis-10-08-2011-23:47
Using LAPIC timer as tick source
DS: CPUs online are [ 0  1 ]
MINIX SCC started
ipctest0: running in remote-mode on endpt 0x11daa, pid 14, coreid 0, tileid 0
ipctest0: instance 0 [ 0 sends, 0 receives, 17.150120 secs]
ipctest0: instance 1 [ 10000 sends, 0 receives, 17.150119 secs]
ipctest0: total start/stop execution time: 17.150145 secs
&scc_stats.scan_mpb took 316200 (AVG: 317887 in 10074 runs)
&scc_stats.loop_newq took 0 (AVG: 0 in 0 runs)
&scc_stats.try_receive took 0 (AVG: 0 in 0 runs)
&scc_stats.process_msg took 349110 (AVG: 351165 in 10002 runs)
&scc_stats.incoming_msg took 41040 (AVG: 41405 in 10002 runs)
&scc_stats.enqueue_msg took 13472 (AVG: 13481 in 2 runs)
&scc_stats.write_mpb took 32142 (AVG: 31978 in 10002 runs)
&scc_stats.ipis_send took 10289 (AVG: 10488 in 10002 runs)
&scc_stats.ipis_recv took 358934 (AVG: 361333 in 10002 runs)
&scc_stats.scc_send took 141861 (AVG: 141720 in 10000 runs)
&scc_stats.scc_recv took 415377 (AVG: 384881 in 74 runs)
&scc_stats.scc_senda took 86262 (AVG: 87145 in 2 runs)
&scc_stats.scc_notify took 0 (AVG: 0 in 0 runs)
```

If you see the above output, MINIX has started successfully. When you are done with testing, do not forget to run **sccReset** before you logout, as otherwise the SCC would still be running code. We had several MCPC crashes earlier when we left images running.

C.4 Configuration

Configuration the MINIX implementation is easy done with a few C header files and Makefiles. The **include/scc/debug.h** configures *Debug Mode*. You may set the **SCC_DEBUG** macro to any of the **SCC_DEBUG_*** to selectively enable debug output in components throught the SCC code. Set it to zero to disable debugging:

```
/* Debug type flags. */
#define SCC_DEBUG_BOOT     (1 << 0)
#define SCC_DEBUG_IPC      (1 << 1)
#define SCC_DEBUG_IPCDUMP  (1 << 2)
#define SCC_DEBUG_IPI      (1 << 3)
#define SCC_DEBUG_USER     (1 << 4)
#define SCC_DEBUG_USERIPC  (1 << 5)
#define SCC_DEBUG_SCHED    (1 << 6)
#define SCC_DEBUG_DS       (1 << 7)
#define SCC_DEBUG_ALL      (0xffffffff)

/* Enabled debugging flags. */
#define SCC_DEBUG 0
```

For changing the tile speed in MINIX, look in **include/scc/clock.h**. The **SCC_TILE_FREQ** macro is set by default to 800MHz:

```
#define SCC_TILE_FREQ (800 * SCC_MHZ)
```

The implementation has performance counters in the IPC code which may be used to debug performance problems. **include/scc/perf.h** contains the **SCC_PERF** macro. Uncomment it to enable performance counters and comment it to disable.

```
/* Enable/disable performance counting. */
#define SCC_PERF
```

To change MPB messaging schemes and choose between IPI and Polling mode, use the **include/scc/kern/conf.h** file. It is not allowed to choose both Split Shared and Lock Shared, only one must be enabled at a time. On the other hand, IPI mode and Polling mode can be enabled both if desired, but not recommended. You should enable only one, but never disable both:

```
/*
 * MPB Message Delivery.
 */

/* Split MPB in per-CPU slots. */
//#define SCC_USE_MPBSPLIT

/* Use the MPB as a big queue with locking. */
#define SCC_USE_MPBLOCK

/*
 * Notifications.
 */

/* Invoke scc_process_msg() regulary during execution. */
//#define SCC_USE_POLLING

/* Invoke scc_process_msg() from inside an IPI handler. */
#define SCC_USE_IPI

/* Only send IPI's once processing is complete. Avoids sending
 * a series of IPI's to the same destination. */
//#define SCC_DELAY_IPIS

/* Only send back an IPI if both the internal receive buffers
 * and the MPB have no new message for us. If disabled an IPI is
 * send every time a message is removed from the MPB.
 */
#define SCC_IPI_IFQEMPTY

/* Do not try again if the destination MPB is full. Instead
 * wait until the destination send back an IPI. Only valid for IPI mode.
 */
#define SCC_NO_TRYAGAIN
```

The kernel buffers incoming messages from other cores which cannot be delivered immediately. Per default, there is room for 8K messages but it can be changes in the **include/scc/kern/msg.h** file:

```
/* Maximum number of messages to buffer. */
#define SCC_RECV_SZ 8192
```

To change the test scenario in the **Ipctest** program, modify the **servers/ipctest/Makefile** to include a different test object:

```
# Makefile for ipctest
TARGETS := ipctest

OBJS-ipctest := main.o generic.o test/send/one_to_one.o

include ../Makefile.inc
```

If you want to start **Ipctest** in local mode, you can configure it in **servers/ipctest/conf.h**. Note that **Ipctest** can only run in either local or remote mode, not both.

```
/* Number of local instances to start. Undefine for remote. */
//#define SCC_IPCTEST_LOCAL 2
```

Finally, all tests have a constant which defines how many messages to send. For example, the one-to-one send() scenario has the **NUM_SEND** macro in **servers/ipctest/test/send/one_to_one.c**:

```
#define NUM_SEND 10000
```