Design and Implementation of a Distributed Operating System for the XMOS XS1 Microprocessor Architecture

BACHELOR THESIS

in partial fulfillment of the requirements for the degree of Bachelor of Information and Communication Technology at Saxion University of Applied Sciences

by

Bianco Zandbergen

Bristol, July 2010

Abstract

XMOS Semiconductor is a designer of semiconductor products located in Bristol, the United Kingdom. XMOS has developed a new microprocessor architecture for embedded systems.

This processor architecture can execute multiple real-time tasks concurrently by using multiple hardware threads per processor core. Each hardware thread can use a guaranteed part of the available processing time. The maximum processing time available to a hardware thread is one fourth of the total amount of processing time available on a processor core. The processor architecture provides mechanisms for hardware threads to communicate and synchronize with each other. The processor architecture is scalable: a processor may incorporate one or more processor cores and multiple processors can be connected with each other using an external processor bus and programmed as one system.

The number of hardware threads that a processor can execute concurrently is limited to eight. This causes in some situations inefficient use of the available processing time because hardware threads can block or it might be more difficult to program the system because multiple tasks have to be combined in a single hardware thread. It is desirable that the processing time available to a hardware thread can be shared among multiple (often non real-time) tasks. To achieve this an Operating System is needed.

We have looked at if porting an existing embedded Operating System will meet the requirements. We have ported the FreeRTOS Operating System to the processor architecture created by XMOS. We came to the conclusion that existing Operating Systems are not able to exploit the advanced features provided by the processor architecture and thus cannot make optimal use of the processor. The reason for this is that the processor architecture of XMOS differs from most existing processor architectures.

Existing Operating Systems are often only able to let tasks running on the Operating System share the processing time of just a single hardware thread. Furthermore it is not possible to run multiple instances of an Operating System concurrently or to define which hardware thread may be controlled by the Operating System. We argue that writing an Operating System from scratch targeted at the XMOS processor architecture is the best solution because the architecture differs significantly from other architectures. We have designed and implemented a new Operating System targeted at the XMOS processor architecture which better exploits the features provided by the processor architecture than existing Operating Systems.

The Operating System that we[1] designed and implemented consists of two important components: the kernel and the Communication Server. The programmer has full control over which hardware threads in a system are configured to share the processing

[1] 'we' can be read as 'I' throughout this document because I am the only contributor to this document and the project.

time between multiple tasks. Each thread configured to share the processing time runs a fully independent kernel. The kernel controls when each task runs and provides other facilities to tasks running under control of the Operating System.

All kernels on a processor core are connected to a Communication Server. The Communication Server enables that kernels on a core can communicate with each other. Tasks running on different kernels can communicate with each other. To enable communication of kernels and tasks controlled by these kernels on different cores, the Communication Servers are connected to a bus. A task running under control of the Operating System can start new tasks that also share the processing time available to the hardware thread on which the creating task runs. A task can also run a program on a dedicated hardware thread which does not share the processing time between multiple tasks. The hardware thread on which the program will run can be located on the same or a different core. The Communication Server plays a key role in this and makes it possible that the task running under control of the Operating System and the dedicated hardware thread can communicate with each other.

The conclusion is that this Operating System compared to existing Operating Systems is better able to exploit the features provided by the XMOS processor architecture. The Operating System is scalable and can be used on systems consisting of just one processor core up to 64 or more cores.

Abstract (Dutch)

XMOS Semiconductor is een ontwikkelaar van halfgeleiderproducten en is gevestigd in Bristol in het Verenigd Koninkrijk. XMOS heeft heeft een nieuwe microprocessor architectuur voor embedded systemen ontwikkeld.

Deze processor architectuur kan meerdere real-time taken parallel uitvoeren door gebruik te maken van meerdere hardware threads per processor core. Elke hardware thread kan gebruik maken van een gegarandeerde hoeveelheid rekentijd. De maximale rekentijd die beschikbaar is voor een hardware thread is een kwart van de totale beschikbare rekentijd van een processor core. De processor architecuur biedt mechanismen voor hardware threads om met elkaar te communiceren en synchroniseren. De processor architectuur is uitermate schaalbaar: een processor kan bestaan uit één of meerdere cores en meerdere processoren kunnen met elkaar verbonden worden via een bus en geprogrammeerd worden als één systeem.

Het aantal hardware threads dat een processor core parallel kan uitvoeren is echter gelimiteerd tot acht. Dit zorgt in sommige situaties voor inefficiënt gebruik van de rekentijd van de processor omdat hardware threads kunnen blokkeren of het lastiger programmeren van het systeem omdat meerdere taken samengevoegd moeten worden tot n hardware thread. Het is wenselijk dat de rekentijd die beschikbaar is voor een hardware thread kan worden gedeeld door meerdere (vaak niet real-time) taken. Om dit te bewerkstelligen is een besturingssysteem nodig.

Allereerst heb ik gekeken of het porten van een bestaand besturingssysteem voldoende is om aan de wensen te voldoen. Ik heb het FreeRTOS besturingssysteem geport naar de processor architectuur van XMOS. Ik kwam tot de conclusie dat bestaande besturingssystemen verre van optimaal gebruik maken van de de functionaliteit die de processor architectuur van XMOS biedt. Dit komt doordat de processor architectuur van XMOS zeer afwijkend is van de meeste bestaande processor architecturen.

Bestaande besturingssystemen kunnen veelal alleen de rekenkracht van maar n hardware thread verdelen. Verder is het niet mogelijk om meerdere besturingssystemen parallel te draaien of te bepalen welke hardware thread in het systeem het besturingssysteem mag beheren. Om deze tekortkomingen aan te pakken heb ik een nieuw besturingssysteem ontwikkeld welke beter gebruik maakt van de functionaliteit die de processor architectuur van XMOS biedt.

Het besturingssysteem bestaat uit twee belangrijke componenten: de kernel en de Communication Server. De programmeur heeft volledige controle over welke hardware threads in het systeem geconfigureerd moeten worden om de rekentijd te verdelen over meerdere taken. De hardware threads die configureerd zijn om de rekentijd te verdelen draaien elk een volledig onafhankelijke kernel. De kernel zorgt voor de verdeling van de beschikbare rekentijd onder de taken.

Alle kernels op een processor core zijn verbonden met een zogenaamde Communication Server. De Communication Server zorgt ervoor dat de kernels op een core infor-

matie kunnen uitwisselen. In samenwerking met de kernels kunnen taken die draaien op verschillende kernels met elkaar communiceren. Om informatie uit te wisselen met andere cores zijn alle Communication Servers met elkaar verbonden via een bus. Een taak die draait op het besturingssysteem kan nieuwe taken starten die mee delen in de rekenkracht die beschikbaar is voor de hardware thread waar de crerende taak op draait. Een taak kan er echter ook voor kiezen op een nieuwe taak te starten die uitgevoerd wordt op een eigen hardware thread zonder de rekentijd van deze hardware thread te delen met andere taken. Dit kan een hardware thread op dezelfde of een andere core zijn. De Communication Server speelt hierin een cruciale rol en zorgt ervoor dat de taak die beheerd wordt door een kernel en de taak die uitgevoerd wordt op een eigen hardware thread met elkaar kunnen communiceren.

De conclusie is dat dit besturingssysteem in vergelijking met bestaande besturingssystemen beter kan omgaan met de functionaliteit die de processor architectuur van XMOS biedt. Het besturingssysteem is uitermate schaalbaar en kan gebruikt worden om systemen bestaande uit een enkele core tot systemen met 64 of meer cores te beheren.

Acknowledgement

I would like to thank XMOS Semiconductor for giving me the opportunity to execute my project at their office. I would also like to thank dr. Henk Muller, my supervisor at XMOS for guiding me and answering numerous questions, whether project related or not. I would like to thank Robin Tobias, my supervisor at Saxion University of Applied Sciences for supervising me. Last but not least i would like to thank family and friends for supporting me during my time in Bristol.

Contents

1 Introduction

Electronic devices often have to perform multiple tasks at the same time. Take for example a mobile phone. When someone is calling, the phone has to receive data through the antenna. It has to process this data to get the receiving voice. At the same time it has to record sound from the microphone and transmit it. Furthermore it has to update the display of the phone.

Such devices are implemented using an embedded microprocessor. Most embedded microprocessors are single-threaded. An Operating System is often used to execute multiple tasks concurrently by sharing the processing time among the tasks. Other microprocessors such as microprocessors based on the XMOS XS1 microprocessor architecture support the execution of multiple tasks concurrently, however the number of tasks that can be executed concurrently is limited. Also in this case an Operating System can be used to execute more tasks concurrently than the microprocessor supports.

1.1 Contributions and Project Objectives

1. Reseach how a simple multitasking kernel can be implemented on the XMOS XS1 processor architecture.

   We have studied the FreeRTOS Operating System to learn how simple embedded Operating Systems work. This resulted in a working product: a port of FreeRTOS to the XMOS XS1 architecture. We discovered that the XMOS XS1 processor architecture differs from common other architectures and that existing Operating Systems are not suitable if one wants to fully exploit the processor features. We have looked at the Barrelfish distributed Operating System, a research project by Microsoft and ETH Zurich. Section 6 discusses the results of our research.

2. Design of a multitasking kernel for the XMOS XS1 microprocessor architecture.

   We have designed a new Operating System targeted at the XMOS XS1 architecture. It is a distributed Operating System designed to run on small systems to large systems with many processors. The design of our Operating System is discussed in section 7

3. Implementation of a multitasking kernel for the XMOS XS1 microprocessor architecture.

   We have succesfully implemented a prototype of the Operating System with the requirements discussed in section 5.
   The implementation of the operating system is discussed in section C.

4. Research how XC language features can be used by tasks running on the Operating System.

   We have looked at if tasks can be programmed in XC or how they can use features of the XC language. This is discussed in section section **??**

5. Implementation of the XC language features in the Operating System.

   We have implemented the ability for tasks to communicate with hardware threads programmed in XC using the XC Application Binary Interface [4]. The implementation is discussed in section C.

6. Implementation of an application to demonstrate the multitasking kernel.

   We have made example programs to test the features and performance of our Operating System. Appendix B evaluates the performance and scalability of the Operating System by executing applications and collecting data concerning the performance.

2 Background

In this chapter we will provide the background information needed to understand the succeeding chapters. We will first discuss XMOS Semiconductor and the microprocessor architecture created by XMOS. Finally we will discuss common Operating System concepts.

2.1 XMOS Semiconductor

XMOS Semiconductor is a designer and manufacturer of embedded microprocessors located in Bristol, United Kingdom. XMOS Semiconductor has created the XS1 microprocessor architecture.

2.2 XMOS XS1 architecture

The XMOS XS1 microprocessor architecture is a RISC processor architecture targeted at embedded systems. The architecture offers some functionality often implemented by Operating Systems such as multi-threading, inter-thread communication and resource management. The architecture is also scalable so that larger problems can be addressed by using multiple processor cores in a chip or multiple chips (nodes) connected to each other. The architecture supports event-driven and interrupt-driven execution of programs [2].

2.2.1 Threads

The processor core is called Xcore and is an independent processor core with its own memory and I/O system. The Xcore can execute up to eight concurrent threads [2]. All threads on a Xcore share the same address space. This makes it possible for threads to exchange data with each other by using shared memory. This memory is used to store both the instructions to be executed by the threads and the data used by the threads. There are no protection mechanisms to prevent a thread from writing to a memory area it is not supposed to write.

Each thread has its own set of registers and program counter, so the state of a thread does not have to be saved to memory when a context switch happens. A hardware thread scheduler schedules the threads using a round-robin algorithm. At each instruction cycle a different thread executes: there is a context switch at every instruction cycle. The processor time is divided equally between threads and each thread gets a guaranteed part of the processor time. For up to 4 threads each thread can use a maximum of 25% of the processor time due to the design of the architecture. This means that if for example only one thread is running, it can only utilize 25% of the processor time and not the full processor time.

A few instructions can block a thread, for example some of the instructions dealing with events. When such an instruction is executed but the thread cannot continue the thread will be descheduled, releasing the processor to a different thread (if there is one ready). The blocked thread will be rescheduled if the condition to resume execution is met.

Threads on the same processor core can synchronize with each other. This can be done in different ways. The processor has hardware locks which can be used for synchronisation and mutual exclusion. A different way is to set up one thread as a master thread and other threads as slave threads. The master thread can wait until all slave threads have synchronised using a synchronizer. In this way a barrier synchronisation can be implemented.

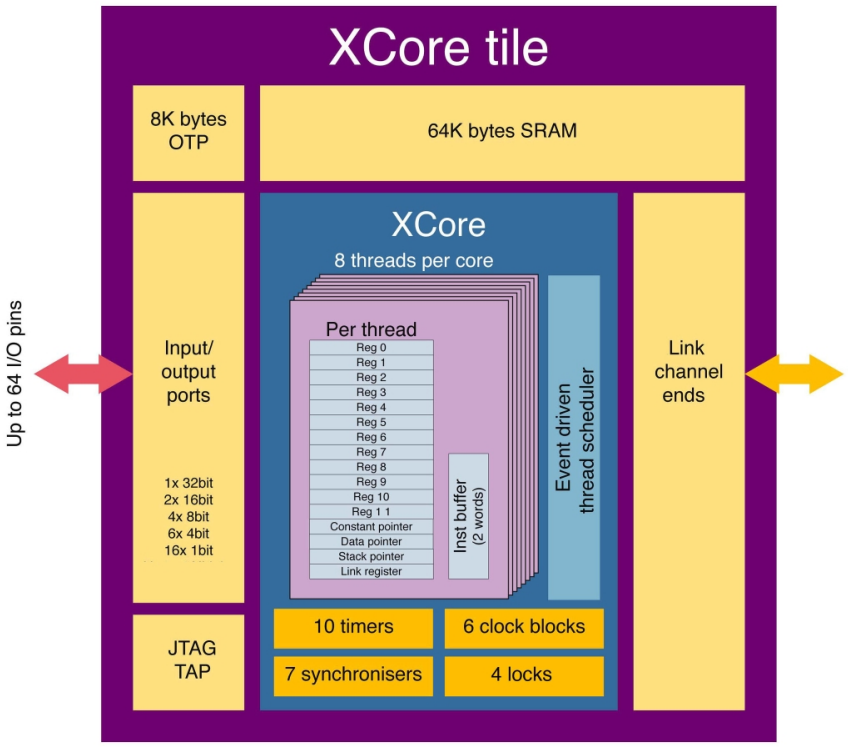

Figure 1: Xcore processor core. Source: XMOS

2.2.2 Events and Interrupts

The XMOS XS1 architecture supports both events and interrupts [2]. Resources such as timers and channel ends can be programmed to either generate an event or interrupt.

A program that uses events can block at certain points in execution until an event

happens. When the event occurs it executes an event handler. At the end of the event handler the program can execute an instruction to wait for the next event [6]. Events always happen when a program expects it. Because the event is expected there is no need to save the thread registers.

A program that uses interrupts will be interrupted in execution when the interrupt triggers. Execution continues at the interrupt handler. Because the program can be interrupted at any moment the interrupt handler must save and restore the used registers.

The advantages of using events instead of interrupts is that events are more deterministic (events are synchronous, interrupts are asynchronous) and can be handled faster (no need for context saving). A disadvantage is that while waiting for an event, no other processing can be done unlike interrupts.

Listing 1: Typical use of events.

```
setup_resources:
    getr    r0,         XS1_RES_TYPE_TIMER      // (1)
    getr    r1,         XS1_RES_TYPE_CHANEND

    ldap    r11,        tmr_vec                 // (2)
    setv    res[r0],    r11
    ldap    r11,        chan_vec
    setv    res[r1],    r11

    ldc     r2,         1024                    // (3)
    setd    res[r0],    r2
    setc    res[r0],    XS1_SETC_COND_AFTER

    setc    res[r0],    XS1_SETC_IE_MODE_EVENT  // (4)
    setc    res[r1],    XS1_SETC_IE_MODE_EVENT

    eeu     res[r0]                             // (5)
    eeu     res[r1]

    waiteu                                      // (6)

chan_vec:
    ...     // process event                       (7)

    waiteu                                      // (8)

tmr_vec:
    ...     // process event                       (7)

    waiteu                                      // (8)
```

Listing 1 shows an example of the use of events. At (1) it allocates a timer and a channel end resource. The resource IDs are saved in the general purpose registers r0 and r1. At (2) it saves the address of the timer event vector (tmr_vec) to register r11. After that it sets the event vector for the resource of which the resource id is saved in register r0 (timer) to the address saved in register r11. In the same way the event vector for the channel end is set. At (3) the condition for the timer to generate an event is set up. It loads the value of 1024 in the data register of the timer. After that it sets that the timer should generate an event if the timer value is larger than 1024. At (4) it sets that

the two resources should generate events (and not interrupts). This step can be omitted because events are the default setting. At (5) it enables the generation of events on the two resources. Even though events can be generated at this stage, event vectors will not be executed. At (6) it checks if an event has occurred prior to executing this instruction. If there is no event at the time of the execution of this instruction, the thread will be blocked until an event occurs. When an event occurs it will execute the event vector of the resource that generated the event. The event can be directly processed (7) without saving the context. After processing the event, it can check for pending events or wait for a new event (8).

Listing 2: Typical use of interrupts.

```
setup_resources:
    getr    r0,         XS1_RES_TYPE_TIMER      // (1)
    getr    r1,         XS1_RES_TYPE_CHANEND

    ldap    r11,        tmr_vec                 // (2)
    setv    res[r0],    r11
    ldap    r11,        chan_vec
    setv    res[r1],    r11

    ldc     r2,         1024                    // (3)
    setd    res[r0],    r2
    setc    res[r0],    XS1_SETC_COND_AFTER

    setc    res[r0],    XS1_SETC_IE_MODE_INTERRUPT // (4)
    setc    res[r1],    XS1_SETC_IE_MODE_INTERRUPT

    eeu     res[r0]                             // (5)
    eeu     res[r1]

    setsr   XS1_SR_IEBLE_MASK                   // (6)

wait_for_int:
    bu      wait_for_int                        // (7)

chan_vec:
    ...     // save context on stack               (8)
    ...     // process interrupt                   (9)
    ...     // restore context from stack          (10)

    kret                                        // (11)

tmr_vec:
    ...     // save context on stack               (8)
    ...     // process interrupt                   (9)
    ...     // restore context from stack          (10)

    kret                                        // (11)
```

Listing 2 shows an example of the use of interrupts. Steps (1), (2) and (3) are the same as the example of the use of events. At (4) it sets that the two resources should generate interrupts. At (5) it enables the generation of interrupts by the two resources. We also have to enable the global interrupt bit in the status register of the hardware thread (6). At (7) it waits for interrupts in a infinite loop. When an interrupt happens,

the interrupt vector of the resource that generated the interrupt will be executed. First it will save the context of the interrupted thread on the stack (8). All modified registers have to be saved. After that it can process the interrupt (9). When the interrupt is processed the context of the interrupted thread has to be restored (10). At (11) it returns from the interrupt vector and continue executing the interrupted thread (the infinite loop in this case). The infinite loop represents a task processing.

2.2.3 Inter-thread communication and scalability

Threads can communicate with each other using channels [2]. A channel is a link between two threads over which the threads can send tokens to each other. These tokens can be data and control tokens. To set up a channel between two threads each thread allocates a channel end. After that each thread sets the destination of the channel end to the channel end of the thread to which it wants to communicate. While the channel ends are set up there is no link between the channel ends. The link is set up when a thread sends a token to the other thread. There is only a limited number of links that can be established at any time between different cores. The exact number depends on the hardware configuration. When a thread has finished sending data to the other thread it can close the link using a special control token.

The links between the channel ends are circuit switched. By closing a link after sending a data packet it behaves as a packet switched network. The links between threads on the same core are handled by the Xcore itself. For links to threads on a different core on the same chip or other chip a switch is used. Each Xcore is connected to this on-chip switch. Two processors can be connected to each other by setting up an external link (XMOS Link) between the switches on each processor.

From the programmer's point of view it does not matter where a thread is located to communicate with it, apart from differences in latency and bandwidth. This makes it possible to program large multi-core, multi-chip systems as if it was a single microprocessor.

Figure 2 shows an example of threads communicating with each other using channels. Thread 0 on XCore 0 has established a link with thread 2 on the same XCore (light blue line). Thread 5 on XCore 0 has established a link with thread 4 on Xcore 1 which is routed by the on-chip switch. Thread 1 on Xcore 1 has set up a link with a thread on a different chip through an external link (red line, the destination is not drawn).

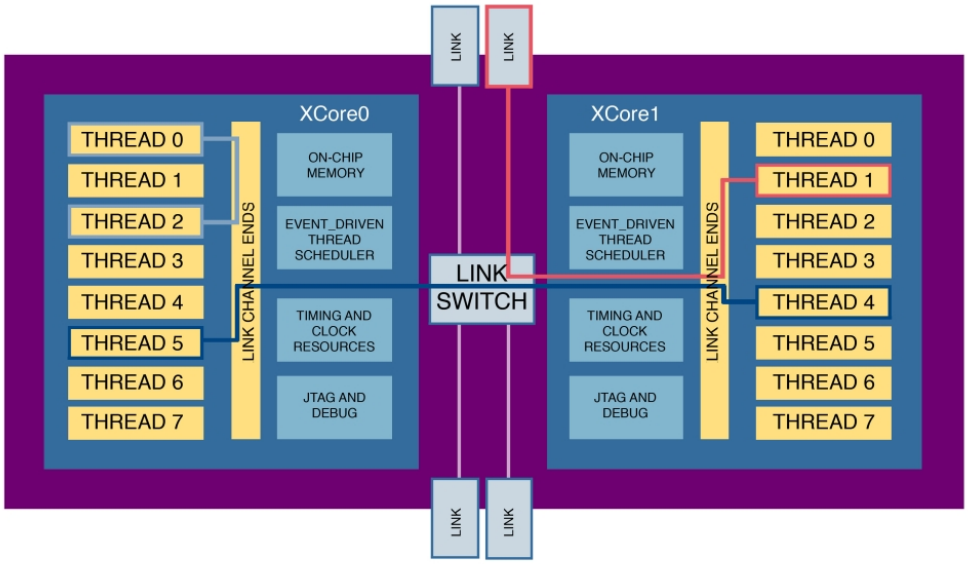

Figure 2: Inter-thread communication using channels. Source: XMOS

2.2.4 Resources

Typical embedded microprocessors use memory mapped I/O or registers to control peripheral devices. For example to set up a serial link one writes values to a specific control register. To send data over the serial link one writes the data to the data register. After a transfer has been completed a bit will be set in the status register. The application can either poll this bit to find out if it can start a new data transfer or it can set up an interrupt which will execute an interrupt handler when the previous transfer has been completed.

On the XMOS XS1 architecture specialised instructions are used that enable a thread to directly communicate with hardware resources [2]. These resources can be allocated and deallocated by a thread using two instructions. When allocating a resource, the instruction will return a resource handler when the allocation was succesfull. This resource handler must be used as operand when using the specialised instruction to control the resource. Resources used by the Xcore for concurrency are channel ends, threads (a thread itself is also a resource!), synchronisers and hardware locks. Resources for I/O and timing are ports, clock blocks and timers.

2.2.5 I/O

While many embedded microprocessors include dedicated hardware for peripheral devices to communicate with external components (i.e UART, SPI), the XS1 architecture only incorporates general purpose I/O (GPIO). However there are advanced mechanisms to implement a wide range of peripherals using software threads. With these advanced mechanisms it is possible to implement for example an ethernet MAC or USB MAC entirely in software.

The GPIO pins are controlled using ports [2]. These ports are of a certain width (1 to 32 bit) and multiple ports are multiplexed to the same physical pins. For example a collection of 32 GPIO pins can be controlled as one 32-bit port or two 16-bit ports. Ports support buffering and deserializing to offload the processor when processing high speed data streams. Ports can be connected to a clock block to generate accurate timing signals.

2.2.6 Programming XMOS XS1 processors

XMOS XS1 processors can be programmed in Assembly, C, C++ and XC.

2.3 The XC programming language

XC is a language created by XMOS to better exploit the features of their architecture. XC supports explicit paralellism and inter-thread communication using channels. Furthermore the language natively supports timers and ports. In this section we will briefly discuss the most important language constructs.

2.3.1 Concurrency

The par statement is used to create cuncurrent threads [3]. All statements within the par block will be executed in different threads. Code listing 3 shows an example of the usage of the par statement. The par statement will start four threads: two on core 0 and two on core 1. The on modifier is used to denote on which core the threads will run and is only needed in the main function. A par statement can be used anywhere in a program, however one has to make sure that the maximum number of threads per core will not be exceeded. A par statement uses fork-join parallelism: the parent thread that executes the par statement will wait until all statements in the par block have finished executing. The first statement that has to run on the same core as the parent thread will be executed by the parent thread itself.

Listing 3: par statement example

```
int main(void) {
    par {
        on stdcore[0] : function1();    // thread on core 0
```

```
        on stdcore[0] : function2();    // thread on core 0
        on stdcore[1] : function3();    // thread on core 1
        on stdcore[1] : function4();    // thread on core 1
    }
}
```

Code listing 4 shows an example of fork-join parallism. The statements in the first par block have finished executing when the sequental statements are executed. After the sequental statements, a second par statement executes the statements within that block in parallel.

Listing 4: Fork join example

```
void fork_join()
{
    int a, b, c;

    par {
        // parallel exectution of statements on three threads
        a = a + 5;
        b = b + 10;
        c = c + 15;
    }

    // sequental execution of statements
    a = 0;
    b = 0;
    c = 0;

    par {
        // parallel exection of statements on three threads
        a = a + 5;
        b = b + 10;
        c = c + 15;
    }
}
```

2.3.2 Inter-thread communication

In section 2.2.3 we discussed that threads can communicate with each other using channels. Channel communication is the only way for threads programmed in XC to communicate with each other. Channels provide a synchronous point-to-point communication between two threads over which data can be transferred. Code listing 5 demonstrates the use of channels in XC with a producer and a consumer thread. A channel between two threads is set up by declaring a chan variable and pass it to exactly two threads as argument. The data type of the thread parameter is chanend.

Data is written to a channel using the <: operator and data is read from a channel using the :> operator. Writing data to a channel causes a thread to block if the the receiving thread does not do a read operation. Similar, reading data from a channel causes a thread to block if no data is available.

Listing 5: par statement example

```
int main(void) {
    chan c; // channel declaration

    par {
        on stdcore[0] : producer(c); // create two threads and
        on stdcore[1] : consumer(c); // pass the channel variable to the two threads
    }
}

void producer(chanend ce) {
    int send_data = 0;

    while (1) {
        ce <: send_data;
        send_data++;
    }
}

void consumer(chanend ce) {
    int recv_data;

    while(1) {
        ce :> recv_data;

        // process data
    }
}
```

2.3.3 Timers

Each processor core provides a set of timers that provide a reference clock which threads can use to execute in a timely manner. The timers have a 32-bit counter and run on a 100MHz frequency by default. Threads can read the value of a timer and block until a particular timer value has been reached. Code listing 6 shows an example of using a timer.

Listing 6: An example of using a timer in XC.

```
timer t;         // timer declaration
unsigned time;   // variable to store timer value

t :> time;       // read timer value
time += 1000;    // increase timer value
t when timerafter(time) :> void; // block thread until timer has
                                 // reached te value saved in time

// continue executing
```

2.3.4 Ports

Each processor core has a number of physical I/O pins to control and communicate with external devices. These I/O pins are accessed using ports. Ports can have different

widths. These ports are multiplexed with the physical I/O pins. XC also uses ports to control the I/O pins. Code listing 7 shows an example of basic port usage in XC. To use a port, one has to declare a global port variable. Threads can change the state of the pins to low and high by writing a value to the port. Threads can sample the value of a port and save it to a variable.

Listing 7: An example of using ports in XC.

```
in port in_port = XS1_PORT_1A;      // declare a 1-bit wide input port
out port out_port = XS1_PORT_16A;   // declare a 16-bit wide output port

int main(void) {
    int read_input;

    out_port <: 0xBEEF;     // write a value to the output port
    in_port :> read_input;  // read a value from the input port

    return 0;
}
```

Ports provide special mechanisms to increase the performance and accuracy of the I/O control and to offload the processor:

- Threads can wait until a port has a specific value.
- Ports can generate a clock signal and strobe signal in conjunction with the outputted data.
- Ports can buffer the data of read and write operations.
- Ports can serialise and deserialise data.

It is out of the scope of this document to discuss these features of ports.

2.3.5 Multiple events

A thread can wait for multiple events at the same time and process the first occuring event using the select statement. The select statement is mapped by the compiler to a WAIT instruction (see section 2.2.2), creating a fast and predictable event handling mechanism. Code listing 8 shows an example of using the select statement. A default case can be used to avoid the thread from blocking and waiting until an event occurs.

Listing 8: An example of using the select statement.

```
in port in_port = XS1_PORT_1A;      // declare a 1-bit wide input port
timer tmr;  // declare a timer

int main(void) {
    unsigned time;

    tmr :> time;    // get the current value of the timer
    time += 1000;   // increase timer value to the value we wait to wait for

    while (1) {
```

```
        select {
            case in_port when pinseq(1) :> void :
                // process event
                break;
            case tmr when timerafter(time) :> void :
                time += 1000;
                break;
        }
    }

    return 0;
}
```

2.4 Operating Systems

An Operating System is a program or set of programs running on a computer and controls the execution of other programs. Often, Operating Systems implement multitasking whereby multiple programs can run concurrently. It provides services to other programs. Operating Systems often provide a set of system calls which programs can use to request service from the operating system, for example to read a file. An Operating System can be seen an extended machine [8]: it can hide the details of the hardware it is running on by presenting a simpler virtual machine to the programs running on it. For example, if a program wants to read a file from a disk, the program does not have to be concerned about how to communicate with the disk, this is done by the Operating System. It is also a resource manager: it makes it possible that multiple programs can share a resource and that every program will get its share. If for example multiple programs want to read files from the same disk, the Operating System will coordinate the operations.

3 Problem Statement

The XMOS XS1 microprocessor architecture has implemented functionality in hardware which are usually provided by an Operating System. These are:

- Multitasking by executing multiple threads concurrently.
- Thread synchronisation and locks.
- Inter-thread communication using channels.
- Timers.

These functions are implemented as resources. The number of resources provided are limited. The current XS1 architecture offers the following resources per Xcore:

- 8 Threads
- 7 Thread synchronisers
- 10 Timers
- 32 channel ends
- 6 clock blocks (for ports)
- 4 locks

The most important limitation is the number of concurrent threads. The number of other resources provided are balanced with the threads using them.

This limitation can be increased to a level where it will suffice for most applications by connecting multiple XS1 processors together using XMOS Links. This is however a waste of thread resources if the threads only utilize a small part of the available processing time. This aproach results in additional costs due to more components and a more complex PCB, a more complex system and a higher power consumption.

Some applications use a large number of tasks which do not have real-time constraints and do not need a lot of processing time or the processing time is dependent on external factors. Consider a networked application with a webserver, TCP stack, IP stack and Ethernet stack. This would be typically implemented on the XMOS XS1 architecture by assigning a hardware thread to each of the four components. The Ethernet stack has real-time constraints and should run in a dedicated hardware thread. The webserver, TCP stack and IP stack do not have real-time constraints and it is acceptable that they share a single hardware thread. An Operating System is needed to let multiple tasks share the same resources inside a single hardware thread.

4 Goals

In this section we will explain what we want to achieve with our project by giving an example.

4.1 An example case

Figure 3 on page 24 shows the software components of a ethernet enabled media device. One can send an audio and video stream to the device over a network and the device will play back the audio and video. The device also controls an RGB LED light which can light up in different colors. The device has a web interface to change settings, such as the audio volume and balancing.

The hardware of this device consists of a dual-core processor based on the XMOS XS1 architecture, an ethernet PHY, audio LC filters, a controllerless LCD display and the RGB LEDs. The hardware has been developed by the hardware engineers and cannot be changed at this point.

The system consists of 19 software components. All software components are usually programmed to run on separate hardware threads. The dual-core processor only supports 16 hardware threads. In this case it is not possible to implement the system by using a hardware thread for each software component.

Some of these components have real-time constraints and have to run on a hardware thread without sharing the processing time with other threads. These components have a light gray background in figure 3. The minimum amount of processing time that is guaranteed for a thread is 0.125 of the processing time available on a processor core. This is not enough for some software components. For example the display controller in figure 3 requires 1/4 of the processing time available to control a controllerless LCD display. If one thread on a processor core requires 1/4 of the processing time available, it is not possible to run more than 4 threads on that core. This will limit the software engineer even more as he has no longer 16 hardware threads available to implement the 19 software components but only 12 hardware threads!

Software components can be location dependent. They can be bound to a particular core because the hardware to which it interfaces is connected to that core. For example the two software components dealing with the ethernet PHY can only run at core 1 because the ethernet PHY is connected to core 1 and the software components that generate the PWM for the RGB LEDs can only run on core 0 because the LEDs are connected to that core.

Many software components are not CPU-bound and do not have real-time requirements. Multiple of these software components can share the processing time of a single hardware thread. It is desirable that multiple hardware threads can be configured to run multiple software modules. One of the reasons is that the processing time of a hardware thread is limited. An other reason is that software modules might be bound to a particu-

lar core. It is important that the software engineer can partition te processing resources in a fine-grain way.

Figure 3 shows that software components communicate with each other. Software components that can use a shared hardware thread may need to communicate with software components that also use a shared hardware thread. This can be the same hardware thread, a hardware thread on the same core, or on a different core. An example of this is the TCP module and the webserver.

There are also software components that use a shared hardware thread and need to communicate with a software component running on a hardware thread which is not shared with other software components. An example of this is the audio decoder which can run on a shared hardware thread and the modules that generate the PWM to generate sound.

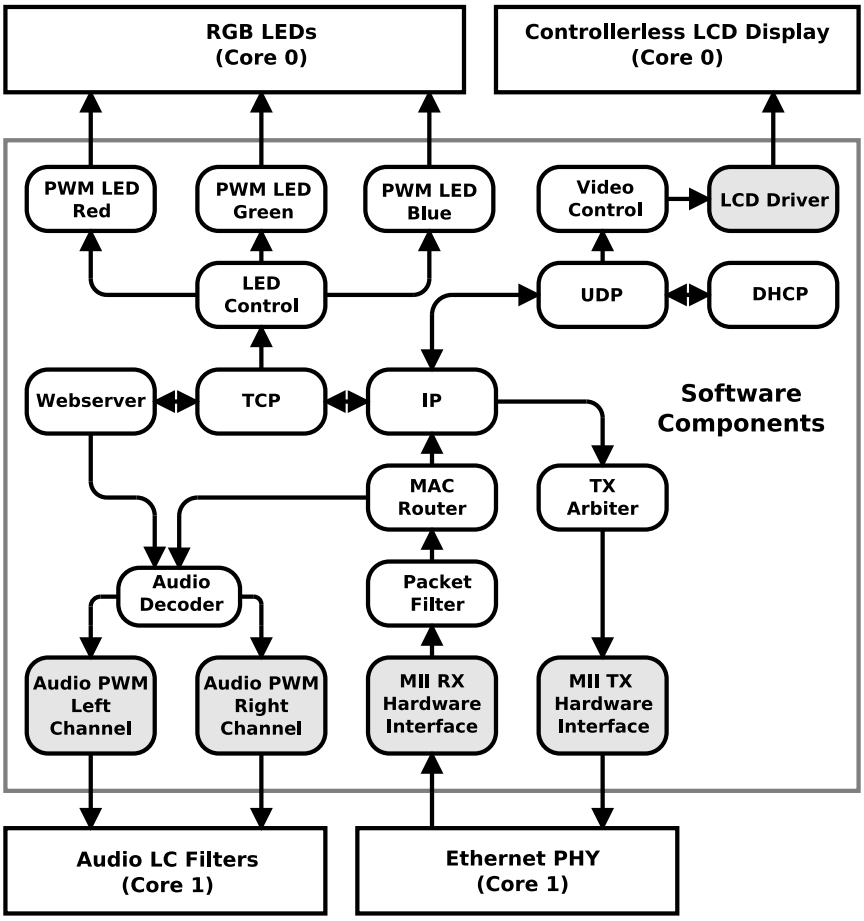

Figure 3: The software components of an ethernet enabled media device.

4.2 Partitioning a system

Figure 4 shows an example of how the system in Figure 3 can be configured. The first core is restricted to run a maximum of 4 threads because the LCD driver needs a minimum of 0.25 of the available processing time on that core. The LCD driver runs on its own hardware thread. Two other hardware threads are used to run the other software components on that core. The second core has four threads that need to run on unshared hardwarethreads. Three other hardware threads are used to run the other software components. Each core has one spare thread left that can be used if the current processing resources are not sufficient.

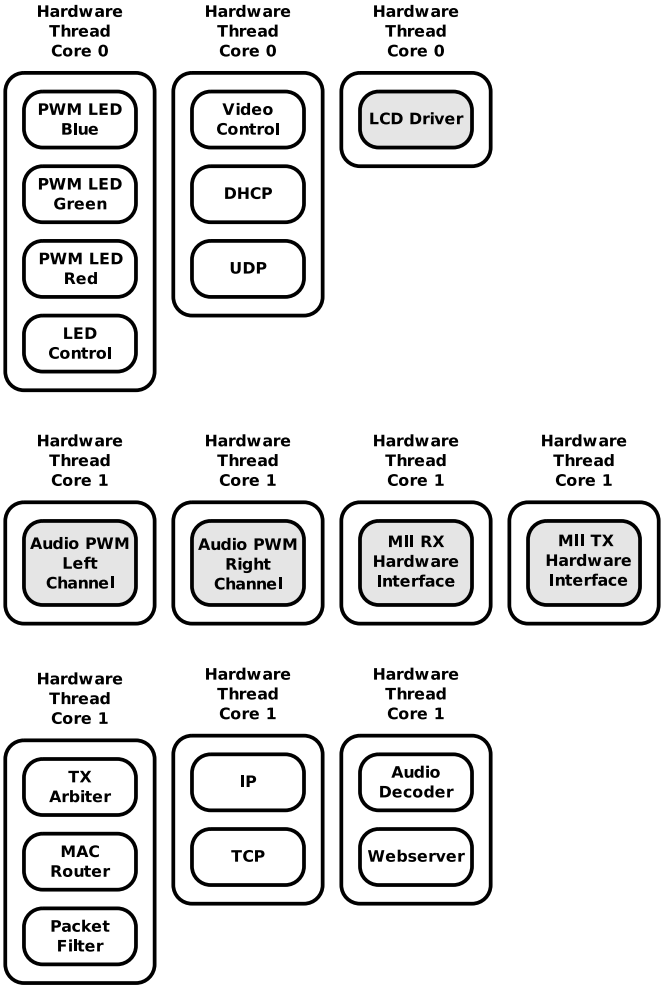

Figure 4: Partitioning the software of an ethernet enabled media device.

4.3 Summary

Our goal is to provide software engineers a tool by means of an Operating System which they can use to partition the processing resources in a more fine-grain way than that is possible with the features provided by the processor architecture. Multiple software modules that usually run on a separate hardware thread should be able to share the processing time of a single hardware thread. Software modules that cannot use a shared hardware thread can still run on its own hardware thread. Software components that use a shared hardware thread should have the ability to communicate with software components that run on their own hardware thread. Software components that both use a shared hardware thread should also have the ability to communicate with each other. The software engineer should have full control over where software components run.

5 Requirements

In this section we will discuss the requirements that we have set up for a new Operating System.

1. Sharing of hardware thread resources between tasks. The processor time available to a hardware thread can be shared between multiple tasks. Not all tasks are equal in importance, tasks have a priority assigned. Resource sharing should not be limited to a single thread. Multiple hardware threads can be set up to share the available processor time among tasks.

2. Tasks must be scheduled preemptively.

3. Configurability of shared hardware threads. The designer of a system must have full control over the number of threads configured to share the processing time between tasks. The designer also has to be able to determine on which cores or processors (in the case of multiple processors) the shared hardware threads run.

4. Tasks must have the ability to communicate with other tasks. The location of the other task can be one of the following:
   - On the same shared thread.
   - On another shared thread on the same core.
   - On another shared thread on a different core.
   - On another shared thread on a different processor.

5. Task running on a shared hardware thread must have the ability to create new hardware threads. The location of the newly created hardware thread can be one of the following:
   - On the same core.
   - On a different core on the same processor.
   - On a different processor.

6. Tasks running on a shared hardware thread must have the ability to communicate with a newly created hardware thread.

7. A hardware thread must be able to communicate with a task running on a shared hardware thread using XC language buildin primitives.

6 Design Rationale

Before designing a new Operating System we have looked at existing Operating Systems and if it is suitable to port an existing Operating System to the XMOS XS1 architecture. In particular we have looked at small embedded Operating Systems which can run on systems with little memory. Each XMOS processor core has 64 KB memory. Larger Operating Systems such as GNU/Linux, Solaris, VxWorks and QNX cannot be ported.

6.1 FreeRTOS

We have chosen to examine the FreeRTOS Operating System because it is small, it is well documented and is already ported to many architectures. FreeRTOS is a small real-time Operating System targeted at embedded systems. It offers the programmer to create tasks with preemptive and cooperative scheduling. Tasks can communicate with each other using message queues. Furthermore FreeRTOS provides semaphores and mutexes to synchronize tasks [5].

We have succesfully ported FreeRTOS to the XMOS XS1 architecture and came to the following conclusions:

- Most small embedded Operating Systems can be ported to the XMOS XS1 architecture.
- Tasks running on the Operating System cannot communicate with other hardware threads using mechanisms provided by the Operating System. The existing Operating Systems are not aware of the channels that the XMOS XS1 architecture provides for inter-thread communication.
- Tasks can communicate with other hardware threads using the XC or assembly language, however this is not recommended because these operations execute instructions that can block a hardware thread. This will block the current running task and also prevents any other task from running.
- The Operating System cannot be relocated. It can only run on the first core of a multi-core system. This is not a problem in the case of a singe-core system.
- It is not possible to run multiple independent instances of the same Operating System on a system. To be able to run multiple instances the Operating System should be relocatable (previous issue) and multiple Operating Systems may not share the same data structures. Most embedded Operating Systems are not made with the requirement to be able to run on multi-core systems. They often use global variables in C source files which will cause problems if multiple Operating Systems run on the same core as these variables will be shared.

6.2 Barrelfish

The Barrelfish Operating System is based on a new Operating System structure called the multikernel [1]. The multikernel is designed in order to address scalability issues in multi-core systems. Traditional kernels treat multicore systems as a single system. The kernel state in these Operating Systems is stored in shared memory, protected by locks. This introduces scalability issues in systems with a large number of cores because only one core can access the kernel data at a time.

The multikernel model treats a each core in a multi-core system as an independent system. The kernel state is not shared by all the cores: each core has its own copy of the kernel state. Changes to the kernel state are updated by explicitly passing messages from one kernel to another. Shared memory message channels are the only shared data structures in systems which do not provide message passing in hardware.

The data structures in the multikernel model are hardware neutral. Only the message passing implementation and device drivers are hardware specific. This makes it possible for a single Operating System to control a heterogeneous system.

The multikernel approach is very attractive for the XMOS XS1 processor architecture as many systems build around this architecture consist of multiple cores. Each core is fully independent and has its own memory. A shared memory model to control a multicore operating system is not possible. However the processor architecture provides excellent mechanisms for threads on different cores to communicate explicitly with each other using communication channels. This makes it possible to create a distributed Operating System using the multikernel model.

We have decided not to port Barrelfish to the XMOS XS1 architecture because it is unclear if Barrelfish can be ported at all. At the time (February 2010) the documenation regarding implementation was poor. A major concern is whether Barrelfish can run within the 64 KB memory that each Xcore offers. Existing ports of Barrelfish (ARM, x86) are targeted at platforms with more memory.

7 Design

In this section we discuss the design of a distributed Operating System for the XMOS XS1 architecture.

7.1 Design considerations

In this section we discuss several decisions we have made concerning the design of the Operating System.

7.1.1 Kernel model

In this section we discuss two approaches to design a kernel to share multiple hardware threads on a processor core.

Single kernel per processor core
In this approach a single kernel is used for all hardware threads on the processor core. All threads run the same same kernel code however, the kernel state is shared by using shared memory. Because the hardware threads are running concurrently, the kernel state may only be accessed by one hardware thread at a time. This can be achieved by using a hardware lock. This affects the performance of the system (called a software lockout [7]). Context switches and kernel calls will take longer on average compared to the approach below.

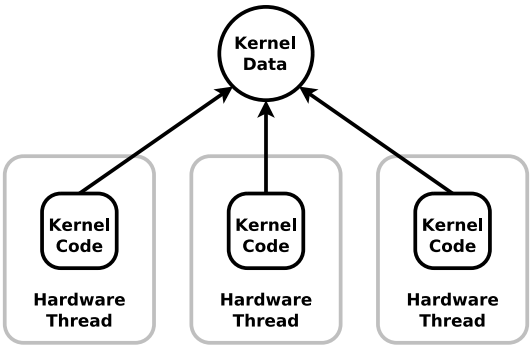

Figure 5: Kernel design approach using a single kernel per processor core.

Multiple kernels per processor core
In this approach each hardware thread is treated as an independent system. All hardware threads on a processor core execute the same kernel code, however each hardware thread has its own kernel data which is not shared with other hardware threads. We can consider this as each hardware thread running an independent kernel. A major advantage is

that the kernel data is always accessible by the kernel. This improves performance and determinism. Another advantage is that each hardware thread can run a different task scheduling algorithm that is optimal for the set of tasks running on the particular hardware thread. A disadvantage is that communication between tasks running on different hardware threads is more complicated compared to the single kernel model, however the implementation of the kernel itself is more simple.

This approach differs slightly from the multikernel model described in section 6.2. In the multikernel model each core has a copy of the kernel data while in this approach each kernel is truly independent. We do think that there are major advantages to having one kernel state. For example a kernel on one core does not need to know the state of a task running on a different core. Barrelfish is not aimed at systems with little memory (such as the 64 KB each Xcore provides). Saving the state of tasks and other components that are not running on the same hardware thread is too expensive in terms of memory.

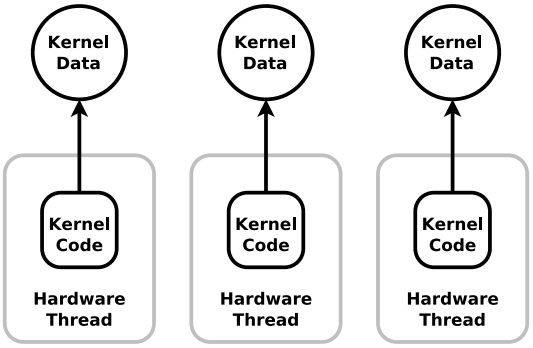

Figure 6: Kernel design approach using multiple kernels per processor core.

Decision
Our design uses multiple kernels per processor core. The major reason for it is that this approach is easier to implement and requires fewer test corners: testing shared memory with locks is error prone and it is easy to forget a corner case.

7.1.2 Communication

Tasks should have the ability to communicate with hardware threads using channels. This can be implemented in several ways. We will discuss three different approaches in this section.

Direct communication
In this approach the Operating System does not implement any specific features regarding communication between tasks and hardware threads. A task can directly communicate to a hardware thread using a channel. The major concern with this approach is

that instructions dealing with channel communication can block the hardware thread on which the tasks runs. This would prevent the kernel and other tasks from running. The current products of XMOS (XS1-G and XS1-L family) support dealing with interrupts while a thread was blocked waiting for an event from a channel. In this way the kernel can gain back the control over the processor resources and run a different task. However using interrupts and events at the same time is not recommended by XMOS. It is not part of the architecture specification and might not be supported in future products. A second disadvantage is that a task which does a channel operation and blocks will occupy the processor waiting for an event until the timer interrupt handles the processor control over to the kernel. Other tasks might be ready to run and thus this is a very inefficient method to deal with communication with other hardware threads.

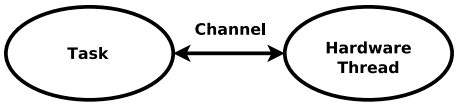

Figure 7: Hardware thread communication approach with a task directly communicating with a hardware thread.

Kernel manages communication

In this approach the kernel handles all communication between tasks and hardware threads. A task that wants to send data to a hardware thread uses a kernel call and supplies the address of a buffer. The kernel handles this call and transmits the data from the buffer to the hardware thread. A disadvantage of this is that transmitting data over a channel uses instructions that might block the thread on which the kernel is running. If the hardware thread is not ready to receive data, the kernel will be blocked (and no task can run) until the hardware thread is ready. Incoming data from the hardware thread will be processed by an interrupt handler and stored in a buffer. This is an inefficient way to deal with incoming data. For each received word the interrupt handler has to be executed. This interrupt handler first has to save the current context (partially). After that it has to find the right buffer and location in the buffer to store the word. Finally it has to restore the context. This involves a lot of code to store a single word. If the hardware thread is continuously sending data, the system will spend more time spending interrupts than executing tasks.

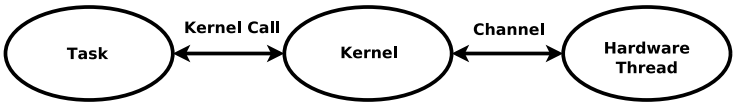

Figure 8: Hardware thread communication approach in which the kernel manages the communication between a task and a hardware thread.

Kernel offloading by using a communication server.
In this approach efforts are done to offload handling the communication between tasks and hardware threads from the kernel. We do this by introducing a communication server. The task of the communication server is to handle all communication between tasks and hardware threads. The communication server will manage a number of buffers which are used to either transmit or receive data. A task can read incoming data by asking the communication server if there is data available. A task can transmit data by telling the communication server that a particular buffer is ready to be transmitted. Tasks however should not directly communicate with the communication server. This should be done by the kernel whereby tasks use kernel calls to instruct the kernel about actions to be performed. This approach will perform better than the previously discussed approaches when dealing with large amounts of data as data can be received or transmitted concurrently with task execution. A disadvantage is that the communication server requires to run on its own hardware thread.

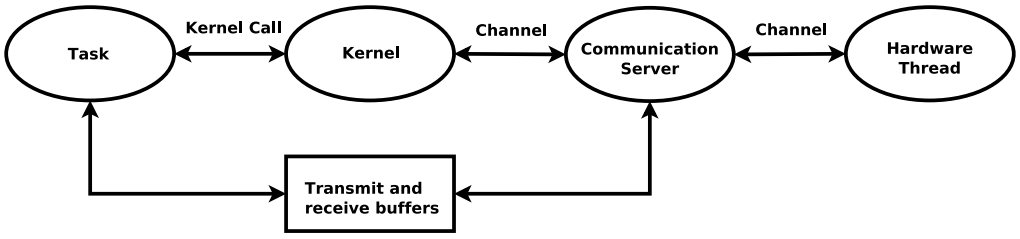

Figure 9: Hardware thread communication approach in which the kernel is assisted by a communication server to enable communication between a task and a hardware thread.

Decision
Our design uses the approach in which the kernel is assisted by a communication server to enable communication between tasks and hardware threads because it provides the best performance.

7.2 XC and Tasks

Applications for the XMOS XS1 architecture are often programmed in the XC language because the C and C++ language cannot exploit specific processor features such as channel communication, timers and ports. It is desirable that tasks running on the Operating System can be programmed in XC and make use of those processor features. However many language constructions in XC can block the hardware thread on which the task is running. This may prevent the kernel from gaining control over the hardware thread or inefficient use of the processing time because no other task can run while the hardware thread is blocked and that is undesirable.

7.2.1 Safe XC language constructions

Language constructions that do not block a hardware thread can be freely used. These are:

- Writing to a normal port.
- Reading from a normal port.
- Reading the value of a timer.

In section 2.3.5 we discussed that a default case can be used within a select statement to prevent a hardware thread from blocking until an event occurs. Even though this language construction does not block a hardware thread, it is unsafe to use it within tasks. The default case is implemented by globally enabling events in the status register of the hardware thread and directly disabling events. The behaviour of globally enabling events is similar to interrupts and it does not require any special instruction to wait for an event. enabling global events and directly disabling global events is not an atomic operation. A task can be descheduled just after globally enabling events and thus not having the chance to disable events. An event can occur when the kernel or an other task is executing. Execution will continue at the event handler set by the task that has been descheduled. This results in undefined behaviour.

7.2.2 Retargeting the compiler

To fully support the XC language for tasks, the XC compiler has to be retargeted. The Operating System will have to provide an API that deals with all the features that XC offers. The XC compiler will have to generate code that makes use of the Operating System API instead of directly accesing the processor. It is out of the scope of this project to retarget the XC compiler. To prove the feasibility of implementing an API that supports functions provided by XC we have implemented API functions regarding timing and channel communication.

7.3 Architectural overview

The Operating System that we designed comprises several major components. In this section we present an overview of those components and other aspects.

The multikernel model
In our system multiple hardware threads can be set up to share the processing time of a thread between multiple tasks. Each thread that shares the processing time is managed by a fully independent kernel. The kernel shares the processing time between tasks using a scheduling algorithm. Furthermore the kernel provides a set of system calls to tasks. These system calls provides mechanisms for communication and hardware abstraction.

Dedicated hardware threads
Tasks can execute programs on dedicated hardware threads. These programs do not run on a Operating System but directly on the processor. The processor architecture provides mechanisms to manage hardware threads. A task can start a program on a local hardware thread (on the same core) or on a remote hardware thread (on a different core).

Communication server
The communication server plays a key role in the Operating System. Each kernel is connected to a communication server by a management channel. A communication server can serve all the kernels on the same core (up to seven kernels). The communication server runs on a seperate hardware thread. A minimal setup of the Operating System is a single kernel and a communication server, using two hardware threads together. The communication server makes it possible for the kernels to communicate with each other. The communication server also manages the dedicated hardware threads on which programs started by tasks run. These dedicated hardware threads can communicate with the tasks via the communication server.

Ringbus
Larger systems with multiple cores run a communication server and one or more kernels on each cores. To enable communication between the cores the communication servers are connected to each other using a ringbus. Not neccesary all communication between cores goes through the ringbus. Direct channels can be set up between threads on different cores to enable lower latency communication.

Configurability
The Operating System is configurable. The programmer of the system decides which cores are used by the Operating System and which threads are set up to share the processing time by running multiple tasks. Processor cores that are not used by the Oper-

ating System can run real-time tasks without any interference of the Operating System, however they will not be able to communicate with tasks running on the Operating System. Figure 10 shows an example configuration. Processor core 0 is configured for two tasks to run tasks. The two kernels are connected to a Communication Server. Processor core 1 is configured to let one hardware thread share the processing time. It also has one dedicated hardware thread. Processor cores 2 and 3 do not have threads that share the processing time. However both run a Communication Server with two dedicated hardware threads that are controlled by tasks on an other core.

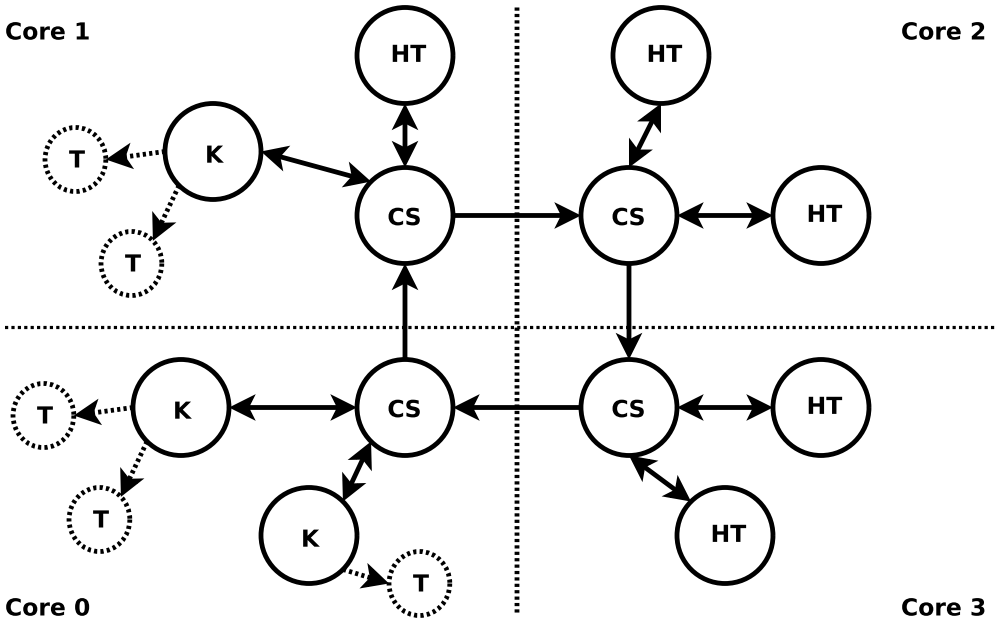

Figure 10: An example OS configuration running on four cores. Legenda: CS: Communication Server; HT: Hardware Thread; K: Kernel; T: Task running on Operating System.

7.4 Kernel

In this section we discuss the design of the kernel proper.

7.4.1 Tasks

A task in our system is a proces running under control of the Operating System that makes use of a shared hardware thread. Each task has a stack on which it can temporary save data. A task can be in three states: ready, running and blocked. A task that is ready is able to execute but the processor is taken by an other task or the kernel. A task in the running state is currently executing its program. A task in the running state is currently executing its program. A task in the running state can be interrupted by resources that generate interrupts on which an interrupt vector is executed. Tasks in the blocked state cannot continue executing because it is waiting for a service from the Operating System or is blocked on purpose (delays for example). Figure 11 shows the possible transitions between states.

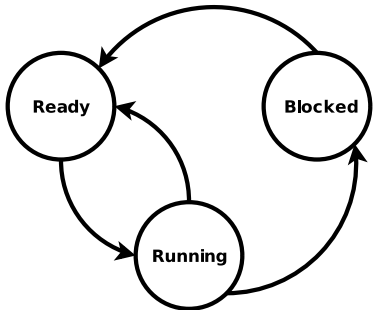

Figure 11: Task states and state transitions.

7.4.2 Scheduling

If multiple tasks share the processing time of a single hardware thread there is need for a mechanism which determines when each task may run. This mechanism is the task scheduler. Tasks do not run till completion before releasing the processor. Tasks can be preempted in favour of an other task. Each task may run for a certain period before it has to release the processor. To preempt a task the kernel has a clock which creates an interrupt at a determined interval. The interrupt routine will save the context of the task. The context of a task is the current current state of a task and consists of the program counter, stack pointer and registers. After the context has been saved the task scheduler will be called to determine which task may run next. After that the context of the process that may run next is restored and execution of this task starts.

A scheduler uses a scheduling algorithm to choose the next runnable task. Our scheduler uses a multi-level queue scheduling algorithm. We have chosen this algorithm because it is simple to implement. Tasks are scheduled based on their priority. The priority is the relative importance of a task. The scheduler has a queue for each priority. A task will be assigned to a queue upon creation. This queue corresponds with the priority of the task. To pick a new task the scheduler checks the queues from the highest priority to the lowest priority until it finds a process in one of the queues.

7.4.3 Delays

Tasks should have the ability to delay for a certain amount of time. For example a task that toggles a LED at a certain frequency can be implemented in the following way:

1. Turn LED on.
2. Delay for a certain amount of time.
3. Turn LED off.
4. Delay for a certain amount of time.
5. Execute item 1.

The smallest measurable amount of time in our Operating System is the tick. Tasks can request to delay for a certain amount of ticks. To do so the task has to execute a kernel call. The kernel will prevent the task from executing untill the number of ticks has passed. The kernel keeps a list of processes that are delayed. The kernel also keeps the number of ticks passed since the system has started. The list of processes is sorted by absolute time: the number of ticks since the system has started on which the task can continue. On each clock interrupt the kernel checks the front of the list for processes that can continue executing. If one or more processes are found they will be removed from the list and added to the scheduling queue so they can resume executing their program.

7.4.4 Kernel entry points

A kernel entry point is a part of the kernel that first starts executing when the control over a hardware thread is deferred from a task to the kernel. There are four kernel entry points. The first entry point is the timer interrupt. A task that is running will be preempted and the scheduler will be invoked to determine which task may run next. The second entry point are other interrupts that service the kernel such as an interrupt on a channel end noting that there is data available. In that case the running task will also be preempted and after processing the interrupt the scheduler will be invoked to determine which task may run next. The third entry point are kernel calls (also called kernel traps). A task can execute a special instruction (KCALL) that generates an interrupt. By executing this instruction the processor continues execution at a known memory location in the kernel. The program can supply a numeric operand to the KCALL instruction to

differentiate multiple kernel calls from each other. The last entry point are exceptions. If a task executes an illegal instruction or combination of instruction and operand, the processor continues executing at a known memory location in the kernel that can process the exception.

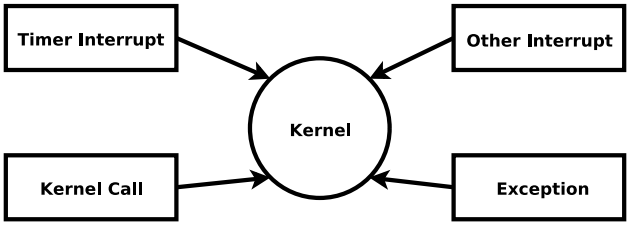

Figure 12: Kernel entry points.

7.4.5 Kernel states

The kernel can be in three states: running, inactive and blocked. The kernel is inactive if a task is running. The kernel is in the running state when it starts executing by entering one of the kernel entry points. While executing the kernel can be blocked by the hardware thread scheduler if it executes instructions that can cause a thread to block. In the current design communication with the Communication Server can cause the kernel to block if the Communication Server is not ready to process the request from the kernel because it is processing an other event. However the blocked state is fully transparant for the Operating System programmer: switching from blocked and running is controlled by the hardware thread scheduler and the programmer does not have to account it. However it introduces latencies. Figure 13 shows the possible transitions between the states.

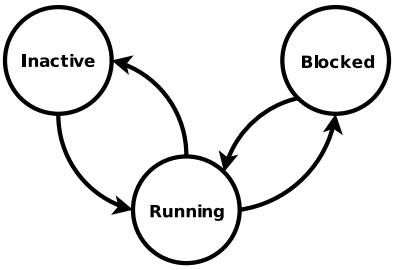

Figure 13: Kernel states and state transitions.

7.4.6 Multikernel

Each hardware thread that is configured to share the processing time runs its own kernel. The kernel is independent in the sense that it does not need other parts of the operating system to control the tasks running on it. However as there are multiple kernels in the system it is desirable that kernels can exchange data with each other. The process of exchanging data between kernels is managed by the Communication Server. Section 7.5.1 discusses the communication between a kernel and a Communication Server.

7.4.7 Dynamic data structures

Small embedded Operating Systems often make use of global variables to save the state of the Operating System. In our case each hardware thread is managed by its own independent kernel. However all the hardware threads on the same core share one memory space. This introduces problems when one runs multiple kernels on the same core as they try to read and write to the same variables. To avoid this problem the kernel data structures will not be allocated at compile time as global variables but at runtime by allocating a chunk of memory.

7.5 Communication Server

The communication server is an essential part of the Operating System. It coordinates all communication from a kernel to other parts of the system.

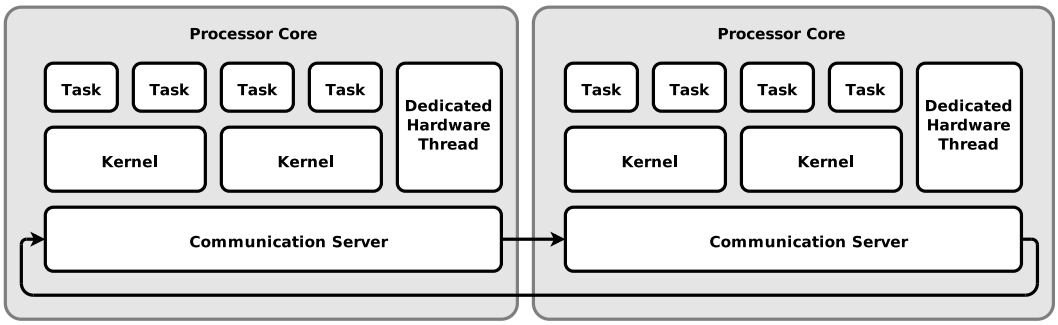

Figure 14: Communication stack.

Figure 14 shows the communication stack. The communication server directly communicates with kernels and dedicated hardware threads on the same core. Tasks that are managed by a kernel cannot directly communicate with the communication server. They can only communicate with the kernel that manages it by using kernel calls. The kernel can defer the request from a task to the communication server.

7.5.1 Kernel Interface

Each kernel is connected to a Communication Server through two channels called the Management Channel and the Notification Channel.

The Management Channel is used for synchronised communication. This is either a request from a kernel on which the kernel does not expect a reply or a request from a kernel on which the Communication Server can directly send a reply back.

The Communication Server does not send a reply back over the Management Channel for requests from a kernel that takes a considerable amount of time to process. For example requests that involve the ringbus takes a lot of time. Instead it sends an asynchronous notification through the Notification Channel to the kernel. The notification indicates that a reply is available and buffered at the Communication Server. This will interrupt the hardware thread that is controlled by the kernel. The kernel will send a request over the Management Channel to receive the reply. In this way the kernel will not be blocked waiting for a reply from the Communication Server and a task can run in the time that the request is processed by the Communication Server.

Our initial implementation did not use notifications. Instead it sent the reply back over the channel that is now called the Notification Channel. This introduced a situation in which a deadlock could occur. Channel communication can block a hardware thread. If a hardware thread will block when sending data over a channel depends on the availability of free buffers in either the switch or the channel end of the receiver. Each channel end has a buffer that fits 8 tokens (8 bytes). This is not enough for the kernel to buffer a reply from the Communication Server. A hardware thread controlled by a kernel that is not executing a task is in kernel mode. In kernel mode interrupts are disabled. If the kernel sends a request to the Communication Server and at the same time the Communication Server sends a reply over the other channel a deadlock occurs. The kernel cannot read the reply because the interrupt is masked in kernel mode. The Communication Server is blocked waiting to complete the transmission of the reply. The kernel is blocked waiting to complete sending the request to the Communication Server. To solve this problem we now use single token notifications. Because each channel end has a buffer for 8 tokens, a Communication Server can send up to eight notifications without blocking.

Even though channels are bidirectional we need two channels if we want to use asynchronous communication. This is due to the way objects are transmitted over a channel using the specification described in the XMOS Application Binary Interface [**?**]. Before and after an object is transmitted the sender and receiver synchronize with each other using special control tokens. This uses both directions of a channel. The sender can only transmit an object if it is sure the receiver is waiting for it, otherwise the synchronisation before a transmission of an object will fail.

The kernel always initiates communication. It sends a messages of a fixed length over the Management Channel. This message contains a command field and several

parameter fields. The Communication Server processes this message. It looks at the command field to determine if it should reply directly on the Management Channel or if it should send a notification when the request has been processed.

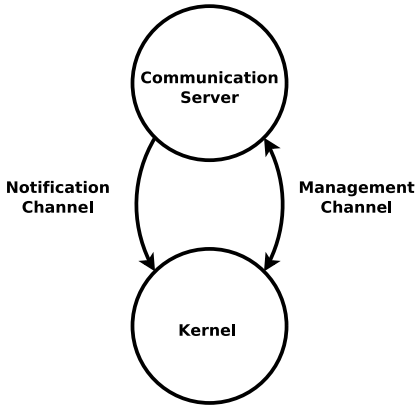

Figure 15: The communication channels between a kernel and a Communication Server.

7.5.2 Ringbus

In a system with multiple cores, each core runs a communication server. To enable communication between the communication servers, they are connected by channels forming a bus in a ring network topology. Communication over this ring is single direction. Each communication server is identified by a number.

Over this ringbus packets with a dynamic length are send. Each packet contains the following fields:

- Sending communication server id
- message type
- status
- payload size
- payload (optional)

Communication over the ringbus is asynchronous. Multiple packets from the same or different communication servers can be on the bus at the same time. To be able to transmit multiple packets over the ringbus and process the returning packets with the reply from the ringbus, the communication server has to save information about each request with a pending reply. The communication server keeps a queue with data structures containing information about each request. Due to the nature of the ringbus topology it is guaranteed that replies from the ringbus are received in the order of the send requests. For each new request send over the ringbus, the communication server

adds a new data structure at the end of the queue. The data structure at the front of the queue will be processed and removed for each incoming reply.

7.6 Inter-task communication

Tasks can communicate with each other by passing messages. Inter-task communication is handled by the Communication Server. To be able to do that the Operating System has to be able to identify tasks. The programmer has to be aware of the identification process as the programmer decides which two tasks communicate with each other. A task that wants to communicate with other tasks has to register itself to the kernel and the communication server. The registration ID is a unique number on the system and created by the programmer. An inbox and outbox for the task is created when it registers. A task can register multiple times to seperate communication with multiple tasks. After registration a task can send or receive to other tasks using the registration ID.

Inter-task communication is fully synchronous. This assures that when a message is delivered it will be processed directly. If queues or other buffering methods are used, it is unknown when a message will be read, if read at all. In our Operating System the recipient executes an API call to receive a message. The kernel notifies the communication server that the task wants to receive a message and blocks the task. If a sender sends a message, the kernel notifies the Communication Server that the sender wants to send a message. After that it will block the sender. The Communication Server will check if the recipient is on the same core or not. If the registration ID cannot be found on the Communication Server, it creates a ringbus packet and transfers the message over the ringbus to Communication Servers running on other cores.

The Communication Server that has the registration ID of the recipient will first check if the recipient is waiting for a message. If the recipient is ready, the message will be copied from the outbox of the sender to the inbox of the recipient and the recipient. After that the Communication Server instructs the kernel on which the recipient runs that the recipient has to be unblocked. If the recipient is not waiting for a message, the sender will stay blocked. When the recipient requests to receive a message and a previous delivery attempt of a sender failed, the Communication Server will try to find the sender in its register. If the sender is found, the message will be copied from the outbox of the sender to the inbox of the recipient and the kernels on which the sender and recipient run will be instructed to unblock the tasks. If the sender was not found, the Communication Server will send a packet over the ringbus to instruct other Communication Servers that the recipient is waiting for a message. The Communication Server with a pending delivery will resend the message over the ringbus to the Communication Server that deals with the recipient.

7.7 Local Dedicated Hardware Threads

Tasks can create dedicated hardware threads to run programs. A local dedicated hardware thread is a hardware thread running on the same core as the task that creates it. To create a new local dedicated thread the task will perform a kernel call with the parameters needed to create the new thread, such as the address of the function to execute and the stack size. The kernel will perform a request to the Communication Server to create this new thread. The Communication Server will create a new thread and initializes the context of the thread. It also creates a channel between itself and the local dedicated thread to communicate with each other. It returns a handle for the task to be used as identification of the dedicated thread.

The task that created the dedicated thread can transmit and receive data to the dedicated thread. However tasks do not directly communicate with the dedicated thread. The Communication Server manages two transmit and two receive buffers for each dedicated thread. Using shared memory a task can request for a transmit buffer and fill it. After the buffer is filled it instructs the kernel to request to the Communication Server to send the buffer to the dedicated thread. The Communication Server replies immediately with the second transmit buffer (if it is available). The task can now fill the new transmit buffer if it needs to. In this way transmitting a buffer and filling a buffer can be executed concurrently.

To receive data a task has to perform a kernel call. The kernel will instruct the Communication Server that the task has made a request for receiving data. There are two types of requests: receive a full buffer or receive a partially filled buffer with a given minimum size. The Communication Server will reply to the kernel with a buffer if the condition is met. Otherwise it will reply that no data is available. In the first case the task can continue running. In the latter case the task will be blocked until data is available. The Communication Server saves the request and as soon as the condition is met it will instruct the kernel that a receive buffer is available to read. The kernel will unblock the task in that case.

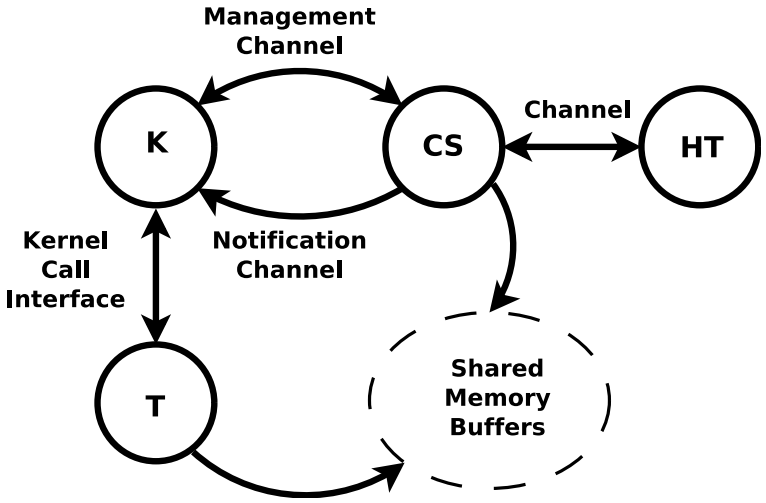

Figure 16: Interface between a task and a local dedicated hardware thread. Legenda: CS: Communication Server; HT: Hardware Thread; K: Kernel; T: Task running on Operating System.

7.8 Remote Dedicated Hardware Threads

To create a remote thread the Communication Server first creates a new channel end to communicate with the dedicated hardware thread that yet has to be created. After that it will send a request over the ringbus to create a new thread (Figure 17 (1)). This could be addressed to a particular Communication Server, or the first one that has the free resources to create a remote hardware thread. The channel end id of the created channel end is one of the parameters of the request. The Communication Server that creates the dedicated thread also creates a new channel end and sets the destination of that channel end to the channel end id that was supplied in the request. After that it will send a reply back containing the channel end id of the created channel end. The requesting Communication Server sets the destination of the created channel end to the one received in the reply to establish a bidirectional direct link (Figure 17 (3)). After initialisation communication is handled is exactly the same way as with local threads.

Setting up the program to run on the dedicated thread is more complicated than on local threads. The code of the program might not be available on the remote core or the location might be unknown. There are two solutions for this problem.

The simplest solution is to load the program at a known location on every core that potential has to run it. A major disadvantage of this approach is that it wastes memory on the cores that never run the program.

An other approach is to dynamically load the code to the core that has to execute the program. To be able to load programs to other cores, the programs should be compiled

in a fully independent capsule. A major disadvantage is that existing tools are not able to create them automatically.

We use the first approach until the tools are able to create independent program capsules.

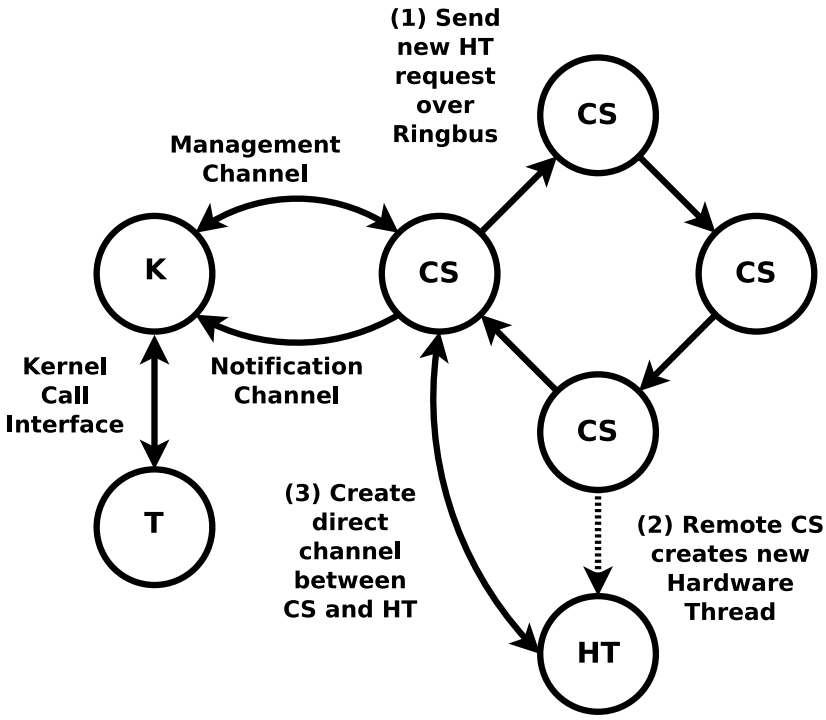

Figure 17: Creation of a dedicated remote hardware thread. Legenda: CS: Communication Server; HT: Hardware Thread; K: Kernel; T: Task running on Operating System.

8 Testing

To test our Operating System we have used development boards and a simulator. With the simulator you can follow the program execution at the instruction level, which was often very useful during the development process. Simulation however is slow compared to execution on the hardware. The Operating System is intended to run on real hardware and we have tested our Operating System on different development boards.

We have focussed testing on features implemented in the Operating System. We have created unit tests to test the operating system in a organised way. A unit in our project is a particular function provided by the Operating System. Each unit test is an application that has to be build together with the Operating System.

The following features have been tested and passed the test:

- The ability to deploy the Operating System on a system with one kernel and one communication server on the same core.
- The ability to deploy the Operating System on a system with multiple kernels and one communication server on the same core.
- The ability to deploy the Operating System on a system with multiple kernels and communication servers on different cores.
- The ability for an initially created task to create new tasks.
- The ability for a task to take parameters.
- The ability for a task to run a program on a dedicated hardware thread on the same core.
- The ability for a task to run a program on a dedicated hardware thread on a different core.
- The ability for a task to transmit data to a dedicated hardware thread.
- The ability for a task to receive data from a dedicated hardware thread.
- The ability for a task to delay for a given number of ticks.
- Inter-task communication.

9 Performance Evaluation

We have performed various measurements to our Operating System and where possible also to FreeRTOS. In this section we summarize the results. The measurement methods and results are discussed in detail in Appendix B.

Context Switch Time
The context switch time is the time it takes for an Operating System to stop the execution of one process and resume the execution of another process. We have measured the context switch time of both our Operating System and FreeRTOS under equal conditions. A context switch in our Operating System uses 129 instructions while a context switch in FreeRTOS uses 102 instructions. This can be partly explained by the fact that our Operating System uses dynamically allocated memory for the main kernel data structure. It takes more instructions to access it.

Memory Footprint
The memory footprint is the amount of memory used by the Operating System. We attempted to measure it at run time. Unfortunately our method was not accurate. We can confirm that our Operating System uses less than 13 KB of memory. The actual memory footprint is estimated at roughly 6 KB.

Inter-task communication
We have measured the throughput and latency of inter-task communication within the same core for our Operating System and FreeRTOS. We also measured the throughput of our Operating System in 2-core and 4-core systems. The throughput of FreeRTOS within the same core is higher than in our Operating System using small messages. The throughput of our Operating System is higher with messages larger than 300 bytes. The maximum throughput within the same core for our Operating System is roughly 70 MB/s. The latency of message passing in FreeRTOS is 4.54 microseconds. The latency of message passing within the same core in our Operating System is 11.16 microseconds. The higher latency can be explained by the different ways inter-task communication is implemented on both Operating Systems.

Communication between tasks and hardware threads
We have measured the throughput and latency of communication with hardware threads. The maximum throughput of receiving data from a hardware thread is 20 MB/s and sending data 22 MB/s. The minimum latency for sending data to a hardware thread is 2.86 microseconds. The minimum latency for receiving data from a hardware thread is 10.60 microseconds.

Scalability
We have succesfully deployed our Operating System on a system with 64 processor cores.

Conclusions
FreeRTOS is performing better in communication within the same hardware thread. However FreeRTOS provides less functionality than our Operating System. For example FreeRTOS does not provide mechanisms to communicate with hardware threads. Also, FreeRTOS can only control a single hardware thread. Our Operating System is performing good enough to implement a system as described in section 4 on page 23.

10 Conclusions and Recommendations

The XMOS XS1 architecture is a microprocessor architecture designed to execute multiple tasks at the same time. The number of tasks that can be executed concurrently is limited. To be able to run more tasks concurrently an Operating System is needed.

We have researched if existing embedded Operating Systems such as FreeRTOS are suitable to manage the resources. We found out that existing Operating Systems can be ported to the XMOS XS1 architecture, but have limitations. Existing Operating Systems cannot make use of multiple hardware threads, nor it is configurable on which core the Operating System should run. Furthermore it is not possible for existing Operating Systems to manage multiple cores or even multiple processors. Tasks running on the Operating System are not able to communicate with programs running on other hardware threads in a safe way. The source of these limitations is that the XMOS XS1 processor architecture differs from most common processor architectures. Furthermore most existing small embedded Operating Systems are designed for single-threaded processors.

The XMOS XS1 processor architecture has instructions that can cause a hardware thread to block. We argue that tasks may not use these instructions directly.

We have proposed a new Distributed Operating System design that uses multiple kernels. Each hardware thread is managed by its own independent kernel. The kernels on each core are connected to a communication server. Tasks on the Operating System can run programs on a dedicated hardware thread which may reside on the same or a different core. The communication servers are connected by a ringbus which enables communication between Operating System components residing on different cores.

We have implemented a new Operating System which applies this design. We have tested and evaluated the Operating System and conclude that it is scalable and configurable. It can run on small systems incorporating one core to large systems with 64 cores. The programmer that deploys the Operating System has full control over which cores are controlled by the Operating System and how many threads on each core is used for timesharing between multiple tasks. The XMOS XS1 architecture is suitable for distributed Operating Systems due to the provided mechanisms which hardware threads can use to communicate with each other in an explicit way with low latencies.

Our focus have been on delivering a proof-of-concept. Efforts have to be done to improve the performance and reliability before our system is usable for real products.

11 Future Work

Task removal
In our current implementation tasks can be created. This can be initial tasks, which start running when the system starts up, or tasks created by other tasks. Some tasks will have to run as long as the system is running, while other tasks complete their job early. Tasks who complete their job should be removed from the Operating System. This enables other tasks to use the freed resources.

Task migration
Each hardware thread is managed by its own kernel. A task has an affinity with this kernel and cannot run on a different kernel. It is desirable that a task can be migrated from one kernel to another. This is easy to achieve if a task has to be migrated to a kernel on the same core. The kernel on which the task previously runned has to send the address of the task_entry data structure to the kernel on which the task continues to run. Migrating a task to a different core is more complicated as the code and data used by the task also has to be migrated.

Improve reliability
Our implementation has been focussed on providing a proof-of-concept. Because of our limited time frame, we were not able to fully implement checks on parameters passed by tasks using kernel calls. It is desirable that the Operating System will be more robust and that the programmer will be protected or noticed of any mistakes where possible.

Improve performance
The performance of an Operating System is important. The Operating System provides a service to the tasks running on it. A more efficient Operating System will directly affect the performance of the tasks. Performance has not been one of our primary focusses. Is is desirable that that the performance of the Operating System will be improved where possible without making any concessions.

More efficient bus system between communication servers
Our current implementation uses a ring bus. One of the main reasons for it is the simplicity to implement a ring bus. However a ring bus is inefficient when the Operating System system manages a system with many cores. It might be worth to take a look at other networ topologies such as a mesh or tree network.

Code migration
Tasks can run programs on dedicated hardware threads located on remote cores. One of the issues of running a program on a remote core is that the program might not be located on that core. In our current implementation, all programs that can be started on

a remote core are loaded into the memory of all cores. This is a waste of memory space if most cores will never run that program. It is desirable that a program will be loaded to the core when a task wants to run the program on that speific core. The program has to be encapsulated in a standalone package and transmitted over the ringbus to the right core.

References

[1] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. *The Multikernel: A new OS architecture for scalable multicore systems*. [online] http://www.barrelfish.org/barrelfish_sosp09.pdf.

[2] XMOS Ltd. David May. *The XMOS XS1 Architecture*. [online] http://www.xmos.com/published/xs1_en.

[3] XMOS Ltd. Douglas Watt. *Programming XC on XMOS Devices*. [online] http://www.xmos.com/published/xc_en?ver=xcuser_en.pdf.

[4] XMOS Ltd. Douglas Watt et all. *XS1 Assembly Language Manual*. [online] http://www.xmos.com/published/xas_en.

[5] FreeRTOS. *FreeRTOS*. [online] http://www.freertos.org.

[6] XMOS Ltd. *XMOS XS1 32-Bit Application Binary Interface*. [online] http://www.xmos.com/published/abi32.

[7] Stuart E. Madnick. Multi-processor software lockout. In *ACM '68: Proceedings of the 1968 23rd ACM national conference*, pages 19–24, New York, NY, USA, 1968. ACM.

[8] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall, January 2006.

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CS	Communication Server
I/O	Input/Ouput
IPC	Inter-Process Communication
KB	Kilobyte
MB	Megabyte
MIPS	Millions Instructions Per Second
OS	Operating System
PID	Process Identifier
RTOS	Real-time Operating System

Glossary

Blocked
: The state of a process in which it can not run because it is waiting for a particular event or needs service from the operating system.

Context Switch
: The process of switching from one task to another in a multitasking operating system. A context switch involves saving the context of the running task and restoring the previously saved context of the other.

Embedded System
: A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function.

Inter-task Communication
: A technique to exchange data between tasks.

Kernel
: The lowest part of an operating system that deals with process scheduling, memory management and process scheduling.

Kernel Call
: A request by a task for a service performed by the kernel.

Operating System
: An operating system is the software that controls a computer and computer software and governs how they work together.

Preemptive
: A process is preemptive when it can be interrupted without its cooperation.

Priority of a Process
: Measurement of importance of a process.

Ready
: The state of a process in which it can run but is not running.

Real-Time System
: A system from which the correctness depends on the timeliness in which computations are finished..

Scheduler
: A piece of software that decides which process may run.

Scheduling
: The way tasks are assigned to run on the CPU.

Task
: A task is a process running under control of our Operating System or FreeRTOS. Depending on the context a task can also be a piece of software with a defined function, usually running on a hardware thread.

Appendices

A Project Management and Schedule

A.1 Project Management

As project management methodlogy I used a modified version of Extreme Programming. I am the only person contributing directly to the project. It was important that the project management methodology was not a burden to me. Methodologies such as Team Software Process or Scrum are not very suitable for single person teams. The role of client is played by myself and my supervisor. We have set up the requirements together at the start of the project. I used multiple short iterations. Each iteration consists of a design, implementation and test part. Each iteration results in a working product (usually a new feature of the Operating System). After each iteration I discussed the approach for the next iteration with my supervisor. I did not use pair programming because I am the only contributor to the project.

A.2 Milestones

I have defined the following milestones in my project:

- FreeRTOS port
- Design and implementation of a basic kernel (task control)
- Design and implemenation of the Communication Server
- Design and implementation of local dedicated hardware threads
- Design and implementation of receiving data from a hardware thread
- Design and implementation of sending data to a hardware thread
- Design and implementation of a ringbus between Communication Servers
- Design and implementation of remote dedicated hardware threads
- Design and implementation of inter-task communication within the same core
- Design and implementation of inter-task communication between tasks on different cores
- Performance evaluation
- Thesis
- Thesis defense

A.3 Project Boundaries

In the scope of this project are task control, the ability for tasks to communicate with each other and hardware threads. Outside the scope of this project are modifications to the compiler or other tools to achieve my goals. For example modifying the compiler to generate loadable code for remote dedicated hardware threads is outside the scope.

A.4 Project Schedule

My project was planned to start in February and finish in June. At the start of my project I made a project schedule. After a few months this schedule is adjusted to its final version. This is done because the initial schedule was not achievable within the time frame. Figure 18 on page 60 visualizes the schedule in a gantt chart. The deadline for my thesis in June 2010 was not met.

Not all the milestones are visualized in Figure 18: all the milestones related to the Communication Server[2] are congregated.

[2] named Channel Handler in Figure 18

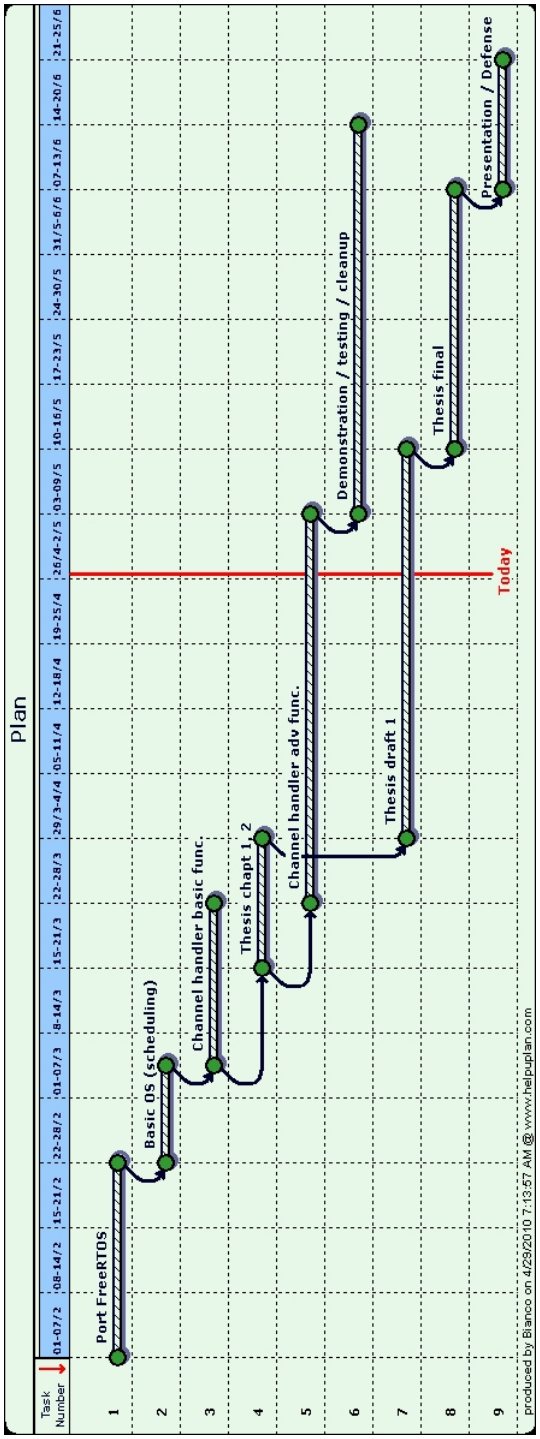

Figure 18: Gantt chart of the schedule.

B Performance Evaluation

We have performed various measurements of our Operating System and FreeRTOS. The methods and results of the measurements are discussed in this section.

B.1 Context Switch Time

The context switch time is the time it takes for an Operating System to stop the execution of one process and resume the execution of another process. Depending on the state of the processes and the scheduling algorithm, the other process can be the process that was just stopped. A lower context switch time is better. We have measured the context switch time of both our Operating System and FreeRTOS.

The context switch time can vary depending on the state of the processes. For example the number of processes that are ready and the priority of the process that should run next may have a large influence on the context switch time. To measure the difference in context switch time between our Operating System and FreeRTOS it is important that we apply the same test conditions. We have done this by running two processes on the system with an equal priority. The priority of the processes is the highest available in the Operating System. The context switch time is measured by running the application in the simulator. Using the simulator we can trace the application at the instruction level and count the number of instructions needed for a context switch.

A context switch in our Operating System uses 129 instructions in the scenario described above. On a 100 MIPS hardware thread a context switch on our Operating System takes 1.29 microseconds. A context switch on FreeRTOS uses 102 instructions. On a 100 MIPS Hardware Thread a context switch on FreeRTOS takes 1.02 microseconds.

The higher number of instructions used by our Operating System can be partly explained by the way the two Operating Systems access the kernel data. FreeRTOS uses global variables to store the kernel state. At the instruction level global variables are accessed using a data pointer with an offset. Our Operating System dynamically allocates memory to save the kernel state. A pointer to the main kernel data structure is saved on the kernel stack. To access the kernel data one first has to switch from the regular stack to the kernel stack and retrieve the pointer to the kernel data structure. This uses considerable more instructions.

B.2 Memory Footprint

The memory footprint is the amount of memory used by the Operating System. This is an important factor in embedded systems because they are often limited in memory capacity. Each processor core based on the XMOS XS1 architecture has 64 KB memory. The Operating System must fit in 64 KB of memory and leave enough space for applications. We attempted to measure the memory footprint at run time by repeatedly

dynamically allocating a block of memory until it fails because the system is running out of memory. Unfortunately the malloc library function we used does not allocate memory in the lowest regions of the available memory. This makes our method not very usable for an accurate estimation for the memory footprint. However using this method we were able to confirm that our Operating System uses less than 13KB memory per processor core in a minimal configuration. An estimation of the actual used memory is 6 KB.

B.3 Inter-task communication

Tasks can communicate with each other by passing messages. Two important factors determine the performance of an inter-task communication implementation: throughput and latency. A high throughput in combination with a low latency is ideal.

B.3.1 Throughput of inter-task communication

Inter-task communication is implemented different on our Operating System and FreeRTOS. Our Operating System uses fully synchronized message passing while FreeRTOS uses message queues. We have measured the throughput by repeatedly (1000 times) sending a message to an other task. Before start sending messages the start time is saved by the sender using a 100MHz timer. After the last message has been sent the end time is also saved.

FreeRTOS is tested with the message queue depth set to four messages to gain a higher throughput in comparison with a queue depth of one message. Our Operating System is tested on one core with the sender and recipient on different kernels, two cores and four cores. The tests are repeated with increasing message sizes (data buffer sizes). Figure 19 shows the results.

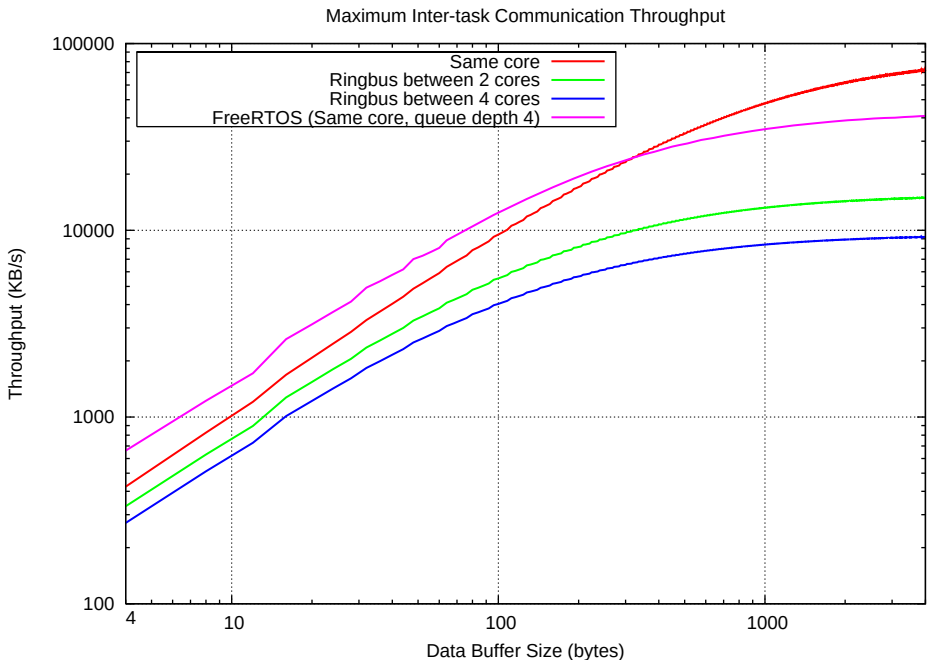

Figure 19: Throughput of inter-task communication using different configurations.

Figure 19 shows that FreeRTOS is performing better with buffer sizes up to 300 bytes. The maximum throughput is about 40 MB/s using messages of 4096 bytes. With message sizes above 300 bytes, our Operating System is performing better when the sender and recipient are on the same core. The maximum throughput is about 70 MB/s using messages of 4096 bytes.

The good performance of FreeRTOS can be explained by two reasons. First, FreeRTOS makes use of message queues which enables the sender to send up to four messages (in the test configuration) without the recipient taking any action. In our Operating System the recipient will have to take action before a message can be received. The recipient has to be blocked waiting for a message before the sender is able to deliver the message. If the sending task tries to send a message while the recipient is not ready, it will be blocked until the recipient becomes ready. Second, inter-task communication is fully handled by the FreeRTOS kernel, while in our Operating System it is handled by the Communication Server. Communication between the kernel and the Communication Server introduces more overhead. We currently do not have an explanation why our Operating System is performing better with large message sizes.

Figure 19 also shows the throughput of our Operating System in small multicore configurations with two and four cores. These tests are not performed for FreeRTOS because it cannot control more than one hardware thread and it does not support multiple

cores. Messages transmitted between tasks on different cores are send over a ringbus which has more overhead in comparison to communication between tasks on the same core. Using a ringbus between four cores has a lower throughput than a ringbus between two cores. These results confirm our expectations.

Figure 19 shows a clear relation between the throughput and the message size. A larger message size enables a higher throughput. When transferring small messages, a large amount of time is used for context switches, finding the destination task etc. causing a major overhead. Only a very small fraction of the time is used to actually transfer the message. When the message size is increased, the overhead decreases and the throughput increases.

B.3.2 Inter-task Communication Latency

We have measured the inter-task communication of our Operating System and FreeRTOS within the same core. The inter-task communication latency is the time measured between the execution of the first instruction of the API function to send a message by the sender and the last instruction executed of the API function to receive a message by the recipient. The time is measured in the simulator which enables us to follow the execution at the instruction level. In the test a message of 4 bytes is transmitted. The two tasks on our Operating System are running on different kernels. Assumed are that all the involved hardware threads have a guaranteed availability of 100 MIPS processing power.

The latency in our Operating System is 11.16 microseconds. The latency in FreeRTOS is 4.54 microseconds.

The higher latency in our Operating System can be explained by the way inter-task communication is implemented in FreeRTOS and our Operating System. In FreeRTOS, inter-task communication is handled by the kernel while in our Operating System it is handled by the Communication Server. The communication between the kernel and the Operating System introduces overhead.

B.4 Communication Between Tasks and Hardware Threads

Tasks can communicate with dedicated hardware threads. In this section we discuss measurements performed on the latency and throughput of communication between tasks and dedicated hardware threads. Sending and receiving are measured seperately. This is done because tasks and dedicated hardware threads are heterogeneous entities which deal with communication in different ways. Tasks communicate with dedicated hardware threads using buffers. Dedicated hardware threads communicate with tasks using channels. Sending and receiving data are implemented in different ways and might differ in performance.

B.4.1 Throughput of Sending Data to a Hardware Thread

The throughput of sending to a hardware thread is measured by repeatedly (1000 times) transferring a data buffer from a task to a hardware thread. Before start sending data to the hardware thread the start time is saved by the task using a 100MHz timer. After the last data buffer has been transferred the end time is also saved. The tests are repeated with different data buffer sizes and two different object sizes. Figure 20 shows the results of the measurements.

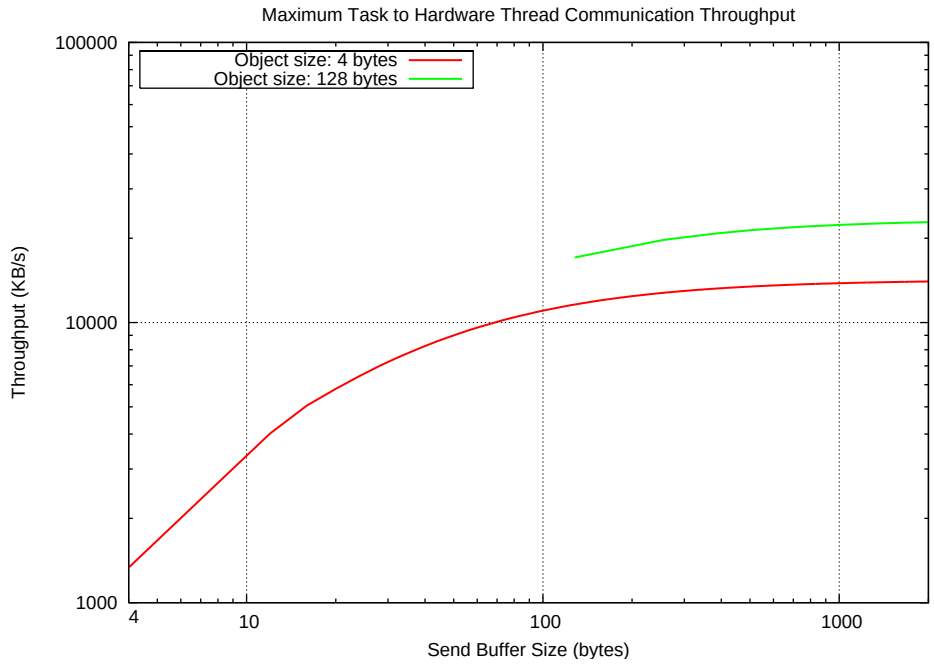

Figure 20: Throughput of a task sending data to a hardware thread.

Figure 20 shows a clear relation between the data buffer size and the throughput. Also a relation between the object size and the throughput is visible. Larger objects and larger buffers decrease overhead, which has a positive effect on the throughput. The maximum throughput is about 22MB/s using 128 byte objects and a data buffer of 2048 bytes.

B.4.2 Latency of Sending Data to a Hardware Thread

The latency of sending data from a task to a hardware thread is measured by sending a buffer of 4 bytes with an object size of 4 bytes to a hardware thread. Before sending the buffer, the task saves the start time using a 100MHz timer. The hardware thread saves

the end time after receiving the object. The measured latency is 2.86 microseconds with all the involved hardware threads having a guaranteed 100 MIPS processing power.

B.4.3 Throughput of Receiving Data from a Hardware Thread

The throughput of sending to a hardware thread is measured by repeatedly (1000 times) receiving a data buffer filled by the Communication Server with data received from a dedicated hardware thread. Before start receiving data the start time is saved by the task using a 100MHz timer. The end time is saved after the last buffer has been received. The tests are repeated with different data buffer sizes and two different object sizes. Figure 21 shows the results of the measurements.

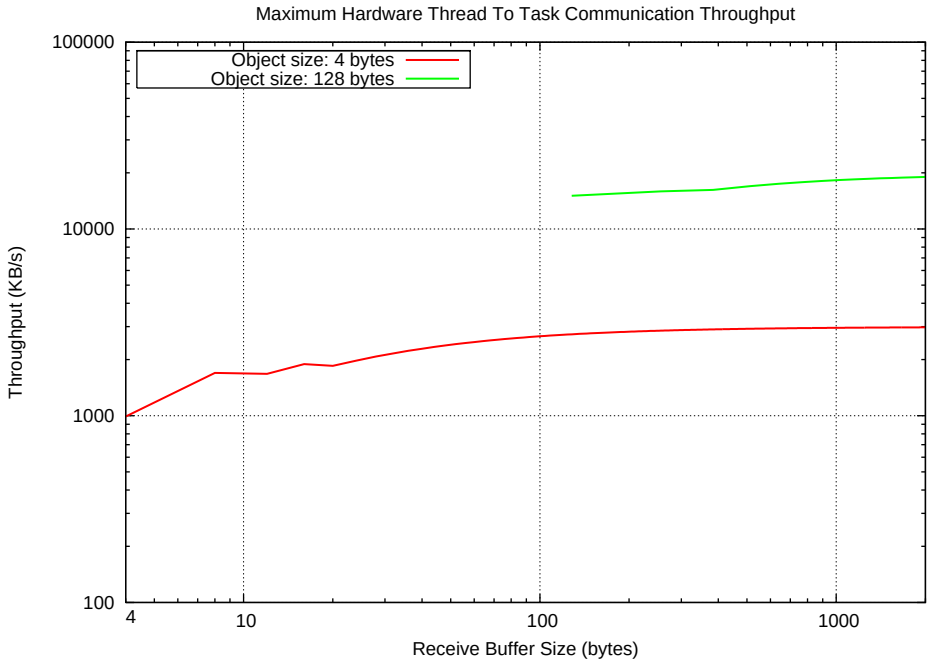

Figure 21: Throughput of a task receiving data from a hardware thread.

B.4.4 Latency of Receiving Data from a Hardware Thread

The latency of a task receiving data from a hardware thread is measured by sending an object of 4 bytes (i.e. an integer) to the task. Before sending the object, the hardware thread saves the start time using a 100MHz timer. The task saves the end time after receiving the object. The measured latency is 10.60 microseconds with all the involved hardware threads having a guaranteed 100 MIPS processing power.

B.5 Scalability

We have deployed our Operating System on the XMOS XK-XMP-64 development board to confirm that our Operating System is able to manage large scale systems. This development board comprises 16 quadcore XS1-G4 processors. The processors are connected to each other using external XMOS links in a hypercube shape (See Figure 22). Each edge of the hypercube provides a maximum bandwidth of 1.6 Gigabit/s. The development board has a total of 64 processor cores and can run 512 hardware threads concurrently. We have measured latency and throughput of inter-task communication in various configurations. In all scenarios each involved processor core runs one Communication Server and one kernel.

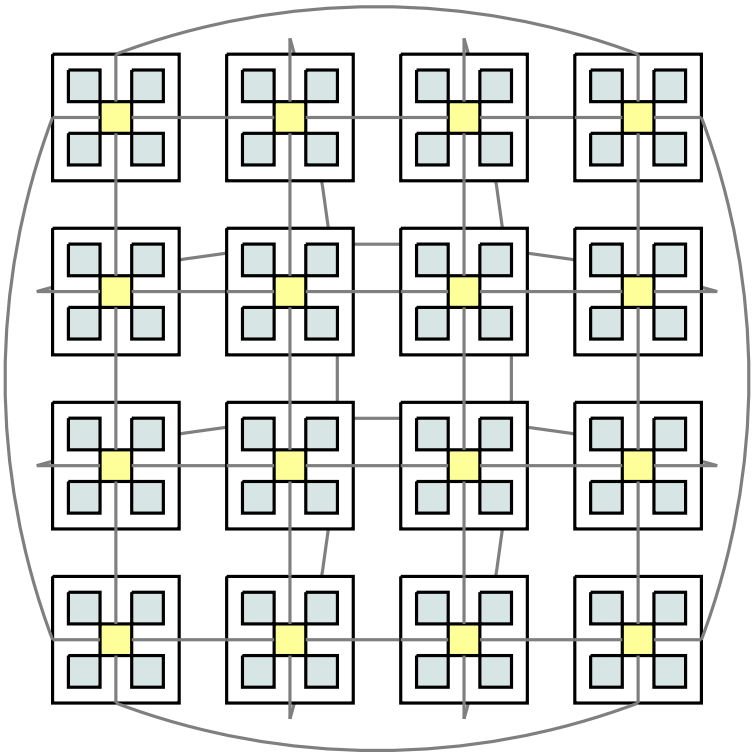

Figure 22: XK-XMP-64 processor interconnections. Blue rectangles are processor cores. Yellow rectangles are communication switches. Grey lines are the processor interconnections.

B.5.1 Throughput on Large Scale Systems

The throughput of inter-task communication is measured in Operating System configurations ranging from 4 cores up to 64 cores. The throughput is measured by repeatedly

sending a message from one task to another. The start time is saved before sending messages using a 100MHz timer. The end time is saved after the last message has been sent.

We have used the worst case scenario for the location of the sender and the recipient: the recipient is on the last node (Communication Server) on the ringbus. Figure 23 illustrates this. Inter-task communication is fully synchronous. A packet transmitted over the ringbus will always return to the Communication Server that has sent it. The sending task will be blocked waiting for it. The packet contains a state, which indicates whether the inter-task message has been delivered or not. A packet transmitted over the ringbus will pass all the ringbus nodes. The payload (the inter-task message itself) will be omitted for the next ringbus nodes if the inter-task message has been delivered. However if the recipient is on the last ringbus node, the payload passes all the ringbus nodes which increases the latency and reduces the throughput.

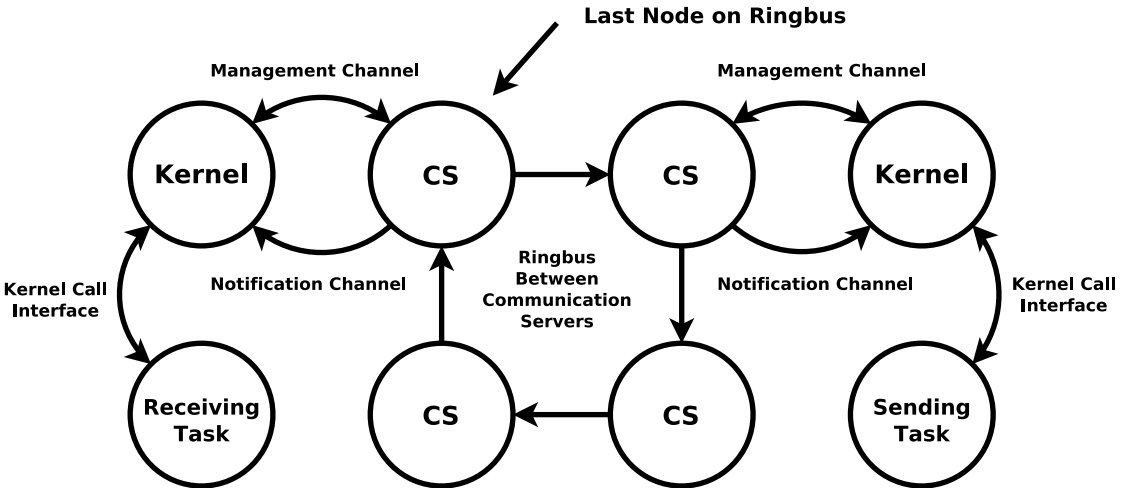

Figure 23: The last node on the ringbus, relative from the Communication Server dealing with the sending task.

Figure 24 shows the throughput on systems ranging from 4 to 64 cores. It clearly shows that the throughput decreases when the number of ringbus nodes increases. The tests are performed using two different message sizes. It shows that a larger message size has a positive influence on the throughput. The throughput of inter-task communication measured in this test is not the maximum throughput of the ringbus itself because multiple packets can be transmitted over the ringbus simultaneously.

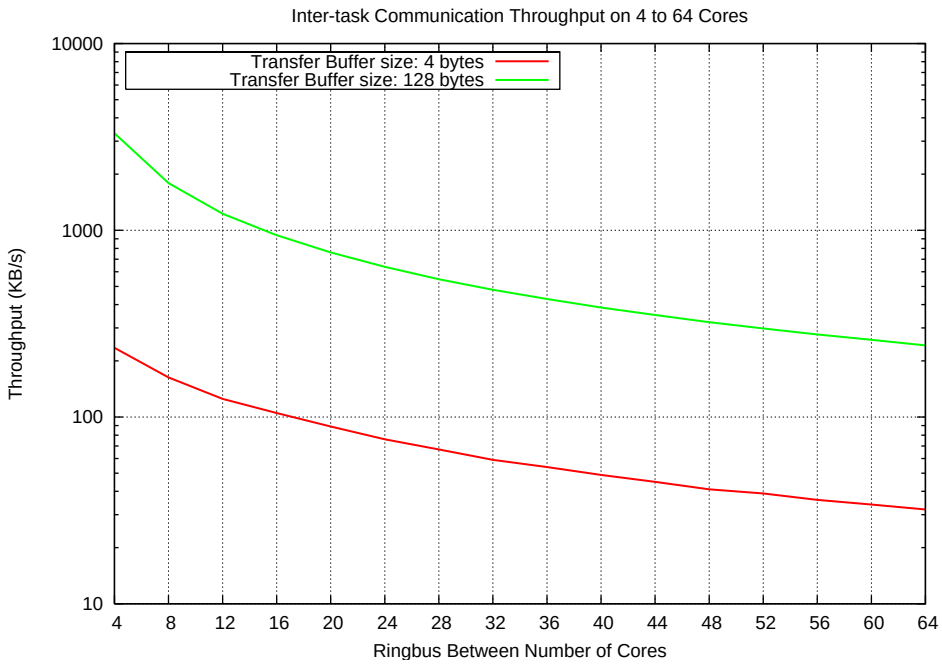

Figure 24: Throughput of inter-task communication on large scale systems.

B.5.2 Latency on Large Scale Systems

The latency of inter-task communication is measured in a different way than in section B.3.2. In section B.3.2 the latency is measured as the time between the sending task calling the send API function and the recipient returning from the receive API function. This was measured in the simulator by following the instruction trace. The latency will vary largely if we measure it in the same as in section B.3.2 depending on the location of the ringbus node dealing with the recipient. In this test we have measured the latency as the time between the sending task calling the send API function and returning from it. This incorporates the latency of sending the ringbus packet through all ringbus nodes. The ringbus node dealing with the recipient is always the last node on the ringbus relative from the node dealing with the sender. All involved hardware threads have a guaranteed availability of 100 MIPS processing power.

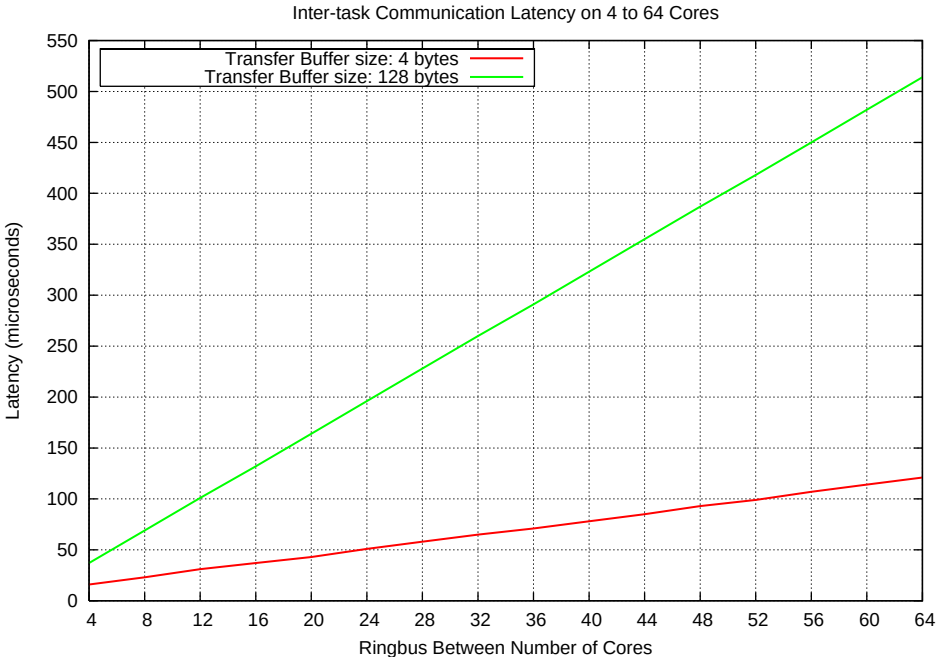

Figure 25: Latency of a task receiving data from a hardware thread using different numbers of cores and two message sizes.

Figure 25 shows the latency on systems ranging from 4 to 64 cores. Two different message sizes are used. It shows that in a best case scenario (when all Communication Server are ready to process ringbus packets) the minimum latency for a small message (4 bytes) is about 120 microseconds. Furthermore the latency increases linearly with an increasing number of processor cores.

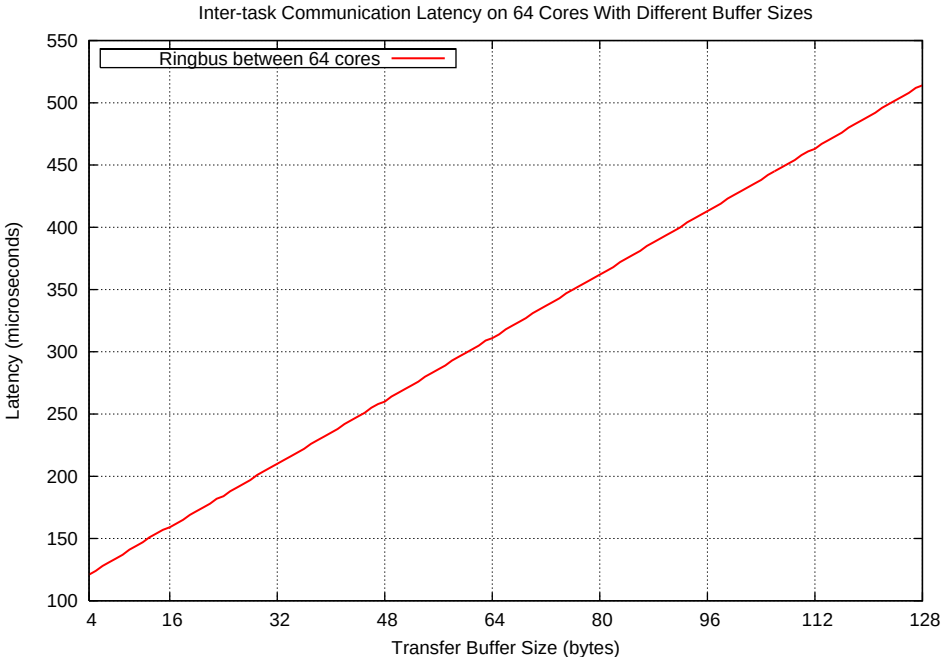

Figure 26: Latency of a task receiving data from a hardware thread using different message sizes.

In Figure 26 the latency was measured on a system with 64 cores using different message sizes. It shows that the latency increases linearly with increasing message sizes.

B.6 Conclusions

We have performed various measurements of our Operating System and FreeRTOS. FreeRTOS is performing better with inter-task communication latency and throughput when small messages are used. FreeRTOS also has a smaller context switch time. However FreeRTOS provides less functionality than our Operating System. For example FreeRTOS does not provide functionality for tasks to communicate with hardware threads. Furthermore FreeRTOS can only manage a single hardware thread. It is a trade-off between performance and functionality. Our Operating System is designed in favor of functionality. Our system is performing well enough to implement a system such as described in section 4 on page 23. We have proven that our Operating System is able to manage large scale systems by deploying our Operating System on the XMOS XK-XMP-64 development board with 64 cores.

C Implementation

In this appendix we discuss parts of the implementation that needs to be more elaborated. This appendix only discusses a part of the implementation.

C.1 Kernel

C.1.1 Data Structures

In this section we will discuss the data structures used by the kernel.

struct kdata
The struct kdata data structure is the main data structure in the kernel. Code listing 9 shows the data structure. The kernel saves the current running task in the current_task field. Tasks that are ready to run are saved in the scheduling queues. There are eight scheduling queues.

There are four fields concerned with time management. First the timer resource id has to be saved, so that the timer can be read and controlled. The number of timer cycles between each cycle is stored in the timer_cycles field. Because the timer cannot be stopped or reset, the value of timer to create the next interrupt has to be saved. On each timer interrupt this field is incremented with the value stored in the timer_cycles field to calculate the time at which the next interrupt will happen. The time field stores the system uptime in ticks. At each timer interrupt this field will be incremented by one.

The head of the list of delayed tasks is saved in the delay_head field. The head of the list of blocked tasks is saved in the block_head field. The last two fields save the resource IDs of two channel ends used for the management channels to communicate with the Communication Server.

Listing 9: struct kdata.

```
struct kdata {
    struct proc_entry * current_task;   /* current running task */
    struct proc_entry * sched_head[8];  /* heads of the ready queues */
    unsigned int timer_res;             /* timer resource handle */
    unsigned int timer_cycles;          /* number of timer cycles per tick */
    unsigned int timer_int;             /* timer value of next interrupt */
    unsigned int time;                  /* time in ticks */
    struct proc_entry *delay_head;      /* start of delayed tasks list */
    struct proc_entry *block_head;      /* start of blocked tasks list */
    unsigned int ch_read;               /* asynchronous management channel */
    unsigned int ch_write;              /* synchronous management channel */
};
```

struct task_entry
The struct task_entry data structure saves the state of a task. Each task has one task_entry data structure. It saves the tasks stack pointer and the stack size. The top of the stack is also saved, which is the lowest memory address that may be used by the stack (the stack

grows downward). The priority of the process and the process identification are also saved. The delay field is used to save the time at which a task can continue executing when it is delayed. The kcall_params and kcall_nr fields are used to save the kernel call state if a task is blocked due to a kernel call. The next field is used to link multiple struct task_entry data structures together to form lists. Each struct task_entry is exactly in one of the following data structures of the struct kdata data structure: the current task, the scheduling queues, the delay list or the block list.

Listing 10: struct task_entry.

```
struct task_entry {
    unsigned long *sp;                  /* task stack pointer */
    unsigned long *bottom_stack;        /* stack top */
    unsigned int stack_size;            /* size of stack */
    unsigned int priority;              /* priority of task: 0-7 */
    unsigned int pid;                   /* process id */
    unsigned int delay;                 /* ticks to delay */
    struct kcall_data *kcall_params;    /* save kcall params when blocked */
    unsigned int kcall_nr;              /* save kcall nr when blocked */
    struct proc_entry *next;            /* pointer to next process for queues */
};
```

C.1.2 Initialization

To initialize a kernel on a hardware thread an initialisation function has to be executed. The function prototype is showed in code listing 11. It takes two function pointers: a pointer to a function which performs the task initialisation and a pointer to the idle task, a task that runs if no other task is ready. The timer_cycles argument provides the number of timer cycles between each timer interrupt. The last two arguments are the two management channels. The start_kernel() function first dynamically allocates memory for the kdata data structure and initializes the fields in this data structure. After that it initializes the kernel data. It allocates memory for the kernel stack and sets the kernel stack pointer. It sets the kernel entry point address. Exceptions jump to this address and kernel calls jump to this address with an offset of 128 bytes. The address of the kdata data structure is saved on the kernel stack which enables the kernel to find the location of this data structure. After that it creates the initial tasks and the idle task (section C.1.3 discusses the creation of tasks). It invokes the scheduler to choose the first task to run. After that it initializes the asynchronous management channel by enabling interrupts and setting up the interrupt routine. After that it configures the timer and timer interrupt. The kernel is now configured and the first task will be started by restoring the context of that task.

Listing 11: Initialize kernel function.

```
typedef void (*task_code)(void *);
typedef void (*init_code)(void);

start_kernel(init_code init_tasks, task_code idle_task,
```

```
unsigned int timer_cycles, chanend ch_read, chanend ch_write);
```

C.1.3 Task creation

To create a task on the Operating System first a new struct task_entry data structure is allocated and the fields initialized. After that, memory for the stack of the task is allocated. The stack will be initialized as if it were running and the context was saved. To start the task, only the context have to be restored. Finally the task will be added to the scheduling queue corresponding with its priority.

C.1.4 Context Switching

Saving the context

The context of a task is saved on the stack of the task. The context consists of the program counter, status register, exception data register, event type register, data pointer, constant pool pointer, link register and 12 general purpose registers. After saving the context on the stack, the stack pointer of the task is saved in the task_entry data structure of the task. Code listing 12 shows the macro used to save the context.

Listing 12: Saving the context.

```
#define SAVE_CONTEXT                                                          \
extsp 20;              /* make room on stack to save context */               \
stw   spc,  sp[1];  /* save the saved program counter (must be sp[1]!) */     \
stw   ssr,  sp[2];  /* save the saved status register (must be sp[2]!) */     \
stw   sed,  sp[3];  /* save the saved exception data register (must be sp[3]!) */\
stw   et,   sp[4];  /* save the event type register (must be sp[4]!) */       \
stw   dp,   sp[5];  /* save the data pointer */                               \
stw   cp,   sp[6];  /* save the constant pool pointer */                      \
stw   lr,   sp[7];  /* save the link register */                              \
stw   r0,   sp[8];  /* save the general purpose registers r0-r11 */           \
stw   r1,   sp[9];                                                            \
stw   r2,   sp[10];                                                           \
stw   r3,   sp[11];                                                           \
stw   r4,   sp[12];                                                           \
stw   r5,   sp[13];                                                           \
stw   r6,   sp[14];                                                           \
stw   r7,   sp[15];                                                           \
stw   r8,   sp[16];                                                           \
stw   r9,   sp[17];                                                           \
stw   r10,  sp[18];                                                           \
stw   r11,  sp[19];                                                           \
ldaw  r10,  sp[0];  /* get value of current stackpointer */                   \
kentsp  0;            /* switch to kernel stack */                            \
ldw   r11,  sp[1];  /* address of kdata */                                    \
ldw   r11,  r11[0]; /* address of current_task task_entry */                  \
stw   r10,  r11[0]; /* store task stack pointer in the task entry */          \
krestsp 0             /* return from kernel stack */
```

Restoring the context

Code listing 13 shows the context restoring macro. It first sets the stack pointer to the

stack pointer of the task by loading the stack pointer address from the current_task field in the kdata data structure. After that it restores all registers except the last two general purpose registers. The reason is that after restoring the context the stack pointer has to be decreased and the instruction set does not provide an instruction to decrease it. The stack pointer is decreased by saving the value in a general purpose register and calculate the new value. After that the new stack pointer can be set. The last two general purpose registers are used to calculate the new value. The last two general purpose registers are at a known offset from the new stack pointer and is restored using absolute addresses calculated from the new stack pointer.

Listing 13: Restoring the context.

```
// expects address of kdata structure in r11!
#define RESTORE_CONTEXT                                                    \
ldw  r11,   r11[0]; /* address of current_task task_entry */               \
ldw  r11,   r11[0]; /* stack pointer of current_task task_entry */         \
set  sp,    r11;    /* set the SP to the SP of the task we restore */      \
ldw  spc,   sp[1];  /* restore saved program counter */                    \
ldw  ssr,   sp[2];  /* restore saved status register */                    \
ldw  sed,   sp[3];  /* restore saved exception data */                     \
ldw  et,    sp[4];  /* restore exception type */                           \
ldw  dp,    sp[5];  /* restore data pointer */                             \
ldw  cp,    sp[6];  /* restore constant pool pointer */                    \
ldw  lr,    sp[7];  /* restore link register */                            \
ldw  r0,    sp[8];  /* restore GP registers r0-r9 */                       \
ldw  r1,    sp[9];                                                         \
ldw  r2,    sp[10];                                                        \
ldw  r3,    sp[11];                                                        \
ldw  r4,    sp[12];                                                        \
ldw  r5,    sp[13];                                                        \
ldw  r6,    sp[14];                                                        \
ldw  r7,    sp[15];                                                        \
ldw  r8,    sp[16];                                                        \
ldw  r9,    sp[17];                                                        \
ldc  r10,   80;           /* stack frame is 20 words */                    \
add  r11,   r11,    r10; /* r11 holds the SP, add 80 to it */             \
set  sp,    r11;          /* set the new SP */                             \
sub  r11,   r11,    8;    /* find the position of r10 relative to the new SP */ \
ldw  r10,   r11[0];       /* restore r10 */                                \
ldw  r11,   r11[1]        /* restore r11 */
```

C.1.5 Kernel Calls

Code listing 14 shows the kernel. The first 64 bytes of the kernel deals with exceptions. In our current implementation exceptions will cause the kernel to execute an infinite loop. When a KCALL instruction is executed by a task, execution starts at the kcep label. It first saves the full context of the task. After that it has to change the saved program counter on the stack. The saved program counter register does not contain the address of the instruction that has to be executed next, but the address of the KCALL instruction executed by the task. To prevent executing this instruction for a second time, the saved program counter on the stack is increased with two bytes (the size of

the KCALL instruction). Kernel calls are processed by a function written in C. Before invoking this function it prepares the three parameters that the function takes. The calling task provides a pointer to a data structure containing the arguments for the kernel call. This is one of the parameters passed to the C function. The C function saves the result of the kernel call back to the data structure passed by the task. When it returns from the C function it restores the context and the task that called the kernel or an other task continues (if the calling task has been blocked) executing. The calling task can examine the result of the kernel call from the data structure it passed to the kernel.

Listing 14: Kernel.

```
.align 128              // align the exception section on 128 bytes
kep:                    // entry point for exceptions
  bu kep                // infinite loop

.align 64               // kernel must be aligned on 64 bytes
kcep:                   // entry point for kernel calls

  SAVE_CONTEXT          // save context of caller process

  ldw r0, sp[1]         // the saved PC does not contain the next instruction
  ldc r1, 2             // but the address of the KCALL instruction.
  add r0, r0, r1        // Add two to the saved PC that was saved on the stack
  stw r0, sp[1]         // to jump over the KCALL instruction.

  get r11, ed           // the kernel call nr is in the exception data register
  add r0,  r11, 0        // store kernel call nr in r0

  ldw r2,  sp[8]         // store pointer to kcall_data structure in r2

  kentsp   1             // switch to kernel stack pointer
  ldw r1,  sp[2]         // get pointer to kdata structure

  bl kcall_handler       // jump to C function to process kcall

  ldw      r11, sp[2]   // load pointer to kcall data in r11
  krestsp  1

  RESTORE_CONTEXT       // restore context of the next running process
  kret                  // handle over the processor to this process
```

C.2 Communication Server

C.2.1 Event-driven state machine

The Communication Server is implemented in a eent-driven way on a seperate hardware thread. After initialisation it waits for events. If there are no events available, it will block untill an event occurs. Events can be of three types. All the kernels on the same core are connected to the Communication Server. A kernel can make a request to the Communication Server using the synchronous management channel. In that case an event occurs and the Communication Server will process the request. Some requests

requires the Communication Server to send a message over the ringbus to other Communication Servers. This will trigger an event at the other communication server. The last type of event is related to the communication between tasks and dedicated hardware threads. Incoming data from a dedicated hardware thread triggers an event that has to be processed. Figure 27 shows the states and transitions between states.

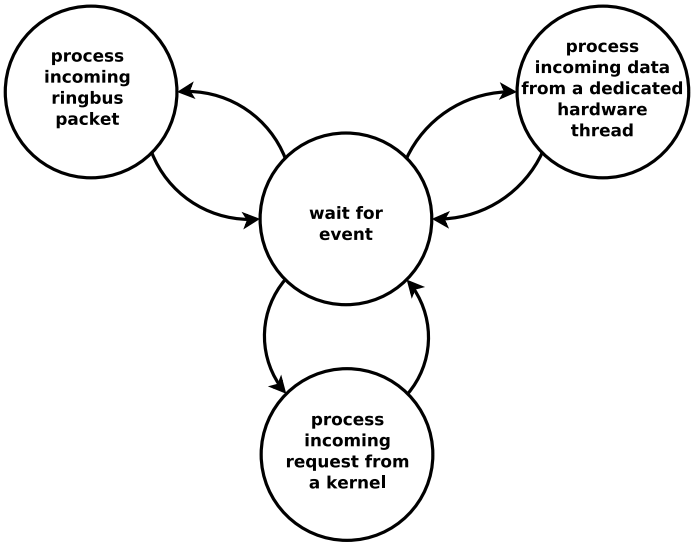

Figure 27: Communication Server states.