Abstract

This thesis project is part of the MANY-project hosted by ITEA2. The objective of Many is to provide developers with tools for developing using multi and manycore, as well as to provide a knowledge-base about software on manycore. This thesis project has the following objectives: to investigate the complex subject of effectively managing system memory in a manycore environment, propose a memory management technique for implementation in OSE and to investigate the Tilera manycore processor TILEPro64 and Enea OSE in order to be able to continue the ongoing project of porting OSE to TILEPro64.

Several memory management techniques were investigated for managing memory access on all tiers of the system. Some of these techniques require modifications to hardware while some are made directly in software.

The porting of OSE to the TILEPro64 processor was continued and contributions where made to the Hardware Abstraction Layer of OSE.
Acknowledgements

I would like to start by thanking all the people who helped me during this thesis. At KTH I would like to thank my examiner Ingo Sander and at Xdin I would like to thank Barbro Claesson and Detlef Scholle for giving me the chance to work on this project.

At Enea I would like to thank OSE chief architect Patrik Strömblad for his assistance as well as Peter Bergsten, Hans Hedin and Robert Andersson for helping me to evaluate memory technologies in OSE and Anders Hagström for guiding me through the intricate details of OSE’s interrupt-handling.

Lastly I would like to thank all my friends who have helped me by proofreading this report and special thanks to Chris Wayment for scrutinizing the language of this report.
Contents

List of Figures ix
List of Tables xi
List of Abbreviations xiii

1 Introduction 1
  1.1 Problem motivation 1
  1.2 Thesis motivation 1
  1.3 Thesis goals 2
    1.3.1 Memory management on manycore platforms 2
    1.3.2 Porting of OSE 2
  1.4 Method 2
    1.4.1 Phase one, pre-study 2
    1.4.2 Phase two, implementation 3
  1.5 Delimitations 3

2 Manycore Background 5
  2.1 Multicore architecture 5
    2.1.1 Heterogeneous and Homogeneous 5
    2.1.2 Interconnection Networks 6
    2.1.3 Memory Architecture 6
  2.2 Programming for multicore 9
    2.2.1 Symmetric Multi-Processing 9
    2.2.2 Bound Multi-Processing 10
    2.2.3 Asymmetric Multi-Processing 10
  2.3 Conclusions 11

3 Tilera TILEPro64 13
  3.1 Architecture Outlines 13
  3.2 Tiles 14
  3.3 iMesh 15
    3.3.1 iMesh Networks 16
    3.3.2 Hardwall protection 16
    3.3.3 Packets and Routing 16
  3.4 Memory Architecture 17
    3.4.1 Cache Architecture 17
    3.4.2 Dynamic Distributed Cache 18
    3.4.3 Memory Balancing and Striped Memory 19
  3.5 Conclusions 19

4 ENEA OSE 21
  4.1 OSE Architecture overview 22
## CONTENTS

4.1.1 Processes, blocks and pools ........................................ 22
4.1.2 Signals ............................................................. 22
4.1.3 Core components .................................................... 23
4.1.4 Portability and BSP .................................................. 23

4.2 OSE Multicore Edition .................................................. 24
4.3 Conclusions ............................................................... 24

5 Memory Management of Manycore Systems 25

5.1 Software-based techniques ............................................. 25
  5.1.1 Managing Cache ..................................................... 25
  5.1.2 Alloc ................................................................. 29
  5.1.3 Transactional Memory ................................................ 29
  5.1.4 Shared memory without locks and Transactional Memory ... 30
  5.1.5 Superpackets ........................................................ 31
  5.1.6 Routing of interconnection networks ................................ 31

5.2 Hardware-based techniques ........................................... 32
  5.2.1 Placement of memory controller ports ................................ 32
  5.2.2 Additional Buffers .................................................. 32
  5.2.3 Memory Access Scheduling ......................................... 33
  5.2.4 Scratchpad Memory .................................................. 33

5.3 Conclusions ............................................................... 34
  5.3.1 Software-Based techniques ......................................... 34
  5.3.2 Hardware-Based techniques ......................................... 35
  5.3.3 Answering Questions ............................................... 36

6 Memory Management on TILEPro64 37

6.1 TILEPro64 ................................................................. 37
  6.1.1 Discussion ........................................................... 37
  6.2 Cache Bins ............................................................... 38
  6.3 Conclusions ............................................................... 39

7 Porting Enea OSE to TILEPro64 ......................................... 41

7.1 Prerequisites and previous work ....................................... 41
7.2 Design Decisions and Goals ............................................ 41
7.3 Implementation .......................................................... 42
  7.3.1 Implementing Cache Bins .......................................... 42
  7.3.2 Hardware Abstraction Layer ....................................... 42
  7.3.3 zzeexternal_interrupts ............................................ 43
  7.3.4 Interrupts ............................................................ 46

7.4 Conclusions ............................................................... 46

8 Conclusions and Future Work ......................................... 47

8.1 Conclusions ............................................................... 47
CONTENTS

8.2 Future work ......................................................... 47
8.3 Discussion and Final Thoughts ................................. 48

Bibliography ......................................................... 49
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Different types of interconnection networks [1]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Examples of cache architectures in a multicore processor [1]</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>The TILEPro64 processor with 64 cores [2]</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Tile in the TILEPro64 processor [2]</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Hardwall Mechanism [3]</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>The layers of OSE</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>Processes in Blocks with Pools in Domains</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>The OSE Kernel and Core Components</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Cooperative Caching [4]</td>
<td>26</td>
</tr>
<tr>
<td>5.2</td>
<td>The overall structure of ULCC [5]</td>
<td>28</td>
</tr>
<tr>
<td>5.3</td>
<td>Different placements of memory controller ports (gray) as proposed and</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>simulated in [6]</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>The architecture upon which [7] has based their simulations</td>
<td>38</td>
</tr>
<tr>
<td>6.2</td>
<td>The microarchitectural layout of dynamic 2D page coloring [7]</td>
<td>38</td>
</tr>
<tr>
<td>7.1</td>
<td>The HAL layer of OSE</td>
<td>43</td>
</tr>
<tr>
<td>7.2</td>
<td>Functionality of the HAL-function zzexternal_interrupt</td>
<td>45</td>
</tr>
</tbody>
</table>
List of Tables

3.1 A detailed table of the cache properties of TILEPro64 chip [3] 18
7.1 The design-decisions made in cooperation with [8] 42
7.2 Section of assembler code in zzexternal_interrupt where zzinit_int is called and returned 44
List of Abbreviations

OS  Operating System
OSE  Operating System Embedded
BMP  Bound Symmetric Multiprocessing
MMU  Memory management unit
BSP  Board Support package
FIFO  First in, first out
DRAM  Dynamic Random Access Memory
SMP  Symmetric multiprocessing
AMP  Asymmetric multiprocessing
STM  Software Transactional Memory
TLB  Translation Lookaside Buffer
RTOS  Real-time Operating System
CB  Cache Bins
CE  Cache Equalizing
CC  Cooperative Caching
ULCC  User Level Cache Control
SIMD  Single Instruction, Multiple data
GPU  Graphical Processing Unit
TM  Transactional Memory
STM  Software Transactional Memory
HTM  Hardware Transactional Memory
CDR  Class-based Deterministic Routing
DSP  Digital Signal Processing
DMA  Direct Memory Access
API  Application Programming Interface
I/O  Input / Output
SPR  Special Purpose Register
DDC  Dynamic Distributed Cache
IPC  Inter-processor Communication
HAL  Hardware Abstraction Layer
UART  Universal Asynchronous Receiver/Transmitter
Chapter 1

Introduction

This thesis project was conducted at Xdin AB, a global services company focused on solutions for communication-driven products. The thesis project is part of the MANY project hosted by ITEA. The objective of MANY is to provide embedded system developers with a programming environment for manycore.

1.1 Problem motivation

The current CPU development has shifted focus from increasing the frequencies of CPUs to increasing the number of CPU cores working in parallel, the most obvious advantage being increased parallelism. Disadvantages are, amongst others, increased power consumption and the difficulties in writing efficient parallel code. Another problem that arises when shifting from single-digit number of cores (singlecore or multicore platforms) to double or even triple-digit number of cores (manycore platforms) is how data should be handled, transported and written by a CPU, or rather, memory management. In a manycore system with a large number of cores and small number of memory-controllers congestion can easily arise.

1.2 Thesis motivation

This thesis project focuses on memory management of manycore based systems on Enea OSE because it is an embedded OS designed by one of the partners of the MANY project. Within the MANY project, there is a ongoing effort (initiated by [9]) of porting Enea OSE to the manycore platform TILEPro64, which is produced by the Tilera Corporation. Therefore, it was the goal of this thesis project to study memory management techniques of manycore platforms in general together with OSE and TILEPro64 to propose a suitable memory management technique for implementation in the ongoing implementation of OSE on the TILEPro64 chip. Because of the fact that OSE is an RTOS, it was investigated how RTOS could be impacted by a manycore platform as well as how a key feature of OSE (message-passing) will cope with manycore platforms.

\[^1\text{http://www.xdin.com}\]
\[^2\text{http://www.itea2.org/project/index/view/?project=10090}\]
\[^3\text{Information Technology for European Advancement, http://www.itea2.org}\]


1.3 Thesis goals

1.3.1 Memory management on manycore platforms

A theoretical study of memory management techniques in general and multicore/many-core platforms was performed. Existing and ongoing research was studied in order to answer the following questions:

- What appropriate memory management techniques exist for manycore platforms and how can existing memory management techniques be adapted for manycore-/multicore systems?

- How do memory management-techniques for manycore systems cope with the demands of a Real-Time Operating System (RTOS)?

Time was also be spent studying OSE and to investigate how OSE’s message-passing technologies will cope with possible problems and pitfalls that the manycore platform may bring and to answer the question

- How will OSE’s message-passing technologies cope with a manycore platform?

And lastly, this thesis project will suggest one memory technique that is suitable for implementation in OSE on the TILEPro64 chip.

1.3.2 Porting of OSE

This thesis project is, as mentioned, part a MANY project with the particular goal of porting Enea OSE to the TILEPro64 chip. To continue the existing porting, a deep understanding of ENEA OSE, Tilera’s TILEPro64 and previous work up to this point was required. Once this was achieved, the contribution to the MANY project was to work on completing hardware-dependant code of OSE as far as possible.

1.4 Method

1.4.1 Phase one, pre-study

During the first phase (covering the first 10 weeks) a literature study of memory management techniques for manycore platforms was performed to answer some of the questions mentioned in the previous section. A study of ENEA OSE and Tilera documentation was also performed in order to continue with the practical implementation of the second phase.
1.5 Delimitations

1.4.2 Phase two, implementation

In the second phase (the final 10 weeks) work on continuing the portation of OSE took place. This means continuing development of the Kernel BSP and other hardware-dependent code of OSE. The step-off point for this work is based on [9].

1.5 Delimitations

The time limit for this thesis project was 20 weeks, in which all the literature studies, implementations, report and presentation was completed. The implementation was done using a Tilera board simulator and the version of OSE to be ported is OSE MCE.
Introduction
Chapter 2

Manycore Background

The clock-frequency race, driven by Moore’s law which stipulates the number of transistors in processors, ground to a halt around the year 2004 \cite{1}. At this point, the focus was shifted to achieving more computational speed by increasing the amount of work being performed in parallel by increasing the number of cores in the processor. The reasons that spurred the change of focus from single core to multicore were many. But the main reason was the fact that power consumption and dissipation became insurmountable obstacles. Technically as well as economically. So the focus had to be shifted from making each core as fast as possible by gorging evermore transistors onto each core and increasing clock frequencies to just adding additional cores next to existing ones.

What separates a multicore processor from a manycore processor is still generally undefined, although some processors with more than ten cores often seem to be referred \cite{1} to as being manycore processors.

2.1 Multicore architecture

When designing and implementing a multicore processor, there are many architectural challenges to take into consideration. Challenges such as designing a heterogeneous or homogeneous processor, how the individual cores should communicate with each other and the outside world, and how the memory should be handled. These issues will be addressed in this section.

2.1.1 Heterogeneous and Homogeneous

There are two different ways of implementing a multicore processor: homogeneous cores and heterogeneous cores. Homogeneous cores mean identical CPUs on all the cores. All cores are equal in architecture and work identically. Heterogeneous would then be a collection of different types of cores designed to perform specific tasks (such as digital signal processing, graphics engines etc.). A heterogeneous multicore processor would theoretically be more efficient and deliver higher performance. But this only holds as long as the task the processor performs is highly specialized. If not, the cost of manufacturing highly specialized processors is not economically justified \cite{1}. A homogeneous processor, most of which implement a shared global address space with full cache coherency, is undoubtedly easier to program for parallelism.
2.1.2 Interconnection Networks

Equipping a processor with several cores requires an inter-core communication mechanism. Historically [1] this has been done by the use of a common bus. This simple, easily implemented and cheap solution does unfortunately not scale very well. Latencies and bandwidth per core quickly becomes a major issue. Newer and emerging mechanisms such as multiple ring buses and switched on-chip networks are emerging and are becoming more and more common, due to lower power consumption, higher bandwidth or both. When the number of cores on a chip continues to increase, these communication networks will face an ever increasing scalability problem and power-consumption constraints. Figure 2.1 illustrates the most commonly used inter-core communication mechanisms of today.

![Interconnection Networks](image)

Figure 2.1: Different types of interconnection networks [1]

2.1.3 Memory Architecture

Memory Controllers

A critical component in any high-performing processor is the memory interface. Even more so in a multicore environment where the memory interface is likely to be a shared resource. Historically the memory interface has been tightly coupled together with the remaining Input/Output devices (I/O). But tendencies over the past decade (Intel and AMD included) have been to separate these in order to increase the memory bandwidth as well as to enabling the memory controller to be used simultaneously with I/O controllers. The physical location of the Dynamic Random Access Memory (DRAM) controllers in relation to processor cores is also of importance in increasing the performance. This is discussed further in section 5.2.1. The logic used in memory controllers...
2.1 Multicore architecture

for multicore processors needs to be more advanced than a regular FIFO-based (First In, First Out) scheduling. Instead, methods of combining access of same memory pages if possible has been proposed [10]. Further techniques for improving memory access will be discussed in section 5.1 as well as possible modifications to hardware in section 5.2. Improving the memory access logic is not a "silver bullet"-solution. When executing parallelized programs, threads executing on different cores tend to use same instructions and work on the same data-sets.

Shared Memory

Shared memory in the context of multicore processors means the way in which processor cores communicate and interact with each other. This is done by storing and reading data in the same (shared) memory location. The main problem with shared memory is that it really does not scale very well as the number of cores (or nodes) increases. While many multicore systems today use one or two levels of own cache memory per core, the problem arises when all cores are working (reading/writing) on the same shared memory location. Making sure that every core constantly has the exact same view (or local copy) of a shared memory location is crucial in making sure that programs will execute properly. Therefore many multicore systems today support a shared memory space between cores as well as a cache-coherent memory system. The definition of this being, that coherency is achieved if and only if, all cores have the same view at any point in time of what is the last globally written value to each location.

Cache architectures on multicore varies greatly. There are different models [1] in which each core has private or shared caches connected in various ways to the main memory. Figure 2.2 illustrates some example layouts (where P denotes processor core). The problem of coherency still remains to be solved in all of these.

Memory coherency

Obtaining full memory coherency is one of the most fundamental problems in designing a high-performing multicore system. This is best illustrated in the situation where multiple copies of the same physical memory location exist at various levels of caches but also within processor cores. This requires a consistent and easy-to-understand model of how parallel reads and writes are coordinated in order to maintain a consistent and coherent view of the memory.

A commonly used concept in this context is store atomicity. If there are multiple copies of the same memory location on several cores, an atomic store would be a store operation that needs to be propagated to all cores instantly. This is naturally impossible to implement practically, but the appearance of an instantly propagated store (atomic store) can at least be achieved if a global order of store operations to the same memory location is enforced. Any order used for this purpose needs to conform
to a consistency model. While there are many such models with variously hard/soft consistency requirements, one of the strictest models available today is called **sequential consistency**. It is formally defined as follows by [1]:

*A multi-processor is sequentially consistent if the result of any execution is the same as if the memory operations of all threads were executed in some sequential order and the operations of each individual thread appear in thread order.*

This implies that the memory (read/write) behaviour of a sequential consistent multicore processor is that of a singlecore system. For a programmer this is very practical, but it is not easily implemented in hardware.
2.2 Programming for multicore

This section will go through three common programming models for multi-processor systems and some thoughts on the concurrency problem for each.

The concurrency problem is one of the most fundamental issues when designing a multicore system. When programming for a single core system, one does not have to take into account the usage of shared resources more than a few threads that can be synchronized using mutexes, semaphores or similar parallelization tools. When involving multicore or even manycores into the picture, some designers rely on OS-supported locking mechanisms (such as mutexes or semaphores) to access shared resources [11].

2.2.1 Symmetric Multi-Processing

An SMP (Symmetric Multi-Processing) system is a system in which the operating system treats all available processor cores as equal resources. The ability to effectively load balance, that is, to equally divide available tasks between cores, is an important design feature. SMP also has the capability to tie a thread to a certain core. This is known as thread affinity. Most commonly, there is only one image of the operating system which is shared between the processor cores. Communication and synchronization take place through the usage of shared memory. This model is easy to manage from a software perspective since it provides a good abstraction layer in which the operating system supplies best-effort CPU load balancing. This high degree of abstraction is in many cases an advantage, but there is a substantial disadvantage in the way in which I/O operations are handled. That is to say, the overhead introduced when performing I/O operations. [12] argues that this introduces a bottleneck to SMP-systems that hinders the scalability of SMP-systems beyond four cores. SMP architectures are widely used today on homogeneous architectures where programmability and shared memory semantics are the main concerns. Most of the major operating systems of today (Windows, Linux, variants of Unix etc.) offer an SMP model.

Concurrency in SMP

Since all resources are managed by the OS in SMP, using a shared memory model for programming on SMP systems is problematic. As mentioned earlier, when a couple of threads try to share the same resources, this can to some extent be solved by locking mechanisms. As the number of threads goes up, more and more locking mechanisms are needed to coordinate resources. So much so, that the OS must spend so much time sorting out locks and coordination, that any potential performance gained by multicore will be reduced [13].
2.2.2 Bound Multi-Processing

The BMP (Bound Multi-Processing) model is a specialized version of SMP. Again, there is only one running OS image, but on a BMP system tasks and processes can be locked (bound) to specific cores, essentially removing the load-balancing feature of SMP and placing it in the hands of the programmer. There are some key advantages to this scheme over SMP, such as improved cache performance when location of tasks is taken into account over cache and a framework for optimizing applications with poor multi-processing behaviour.

Concurrency in BMP

Being a modified variant of SMP, BMP offers a similar setup. However, the key functionality of locking threads or applications to certain processors can mitigate the concurrency problem. Specific concurrency-heavy programs/threads may be locked on a particular core, eliminating some of the problem. But this comes at the cost of serious considerations when designing the system, such as keeping CPU utilization balanced and maximizing system performance.

2.2.3 Asymmetric Multi-Processing

In an AMP (Asymmetric Multi-Processing) system the emphasis and focus are on the (physical) difference of the processor cores. The aim is simply to take advantage of the differences and various benefits of the cores in the system. In an AMP system each core needs to run its own separate instance of the operating system. The advantage would be that high performance is achieved locally on each core and that it scales very well onto several cores. One disadvantage is the lack of support to load balance applications or OS resources in between cores. AMP-based systems are usually used on heterogeneous hardware architectures or in cases when isolation brings additional benefits, such as real-time systems.

Concurrency in AMP

In the AMP model, the division of shared resources is left up to the application designer. Since each core runs a complete and isolated OS, some method of coordinating resources is required. This would inevitably have to be a complex – and potentially, for each platform, specific – solution.

Another concurrency issue in AMP systems is the fact that a process will always be locked to execute upon the same core, regardless of the local core or any other core’s utilization.
2.3 Conclusions

This chapter covered some fundamental principles of manycore processors such as heterogeneous and homogeneous cores, interconnection networks, memory architectures and programming models for manycore processors. It is demonstrated throughout this chapter that the problems when moving from single to manycore processors are many, as are the possible solutions. The solutions chosen for new platforms really depend on the intended purpose of the system. A system intended for a specific task could potentially gain performance from being heterogeneously built and using an AMP programming model whereas a more generic platform, perhaps intended for a desktop computer, could make better use of a homogeneous processor core using BMP or SMP models.
Chapter 3

Tilera TILEPro64

This chapter aims to provide an overview of Tilera’s TILEPro64 chip. This is because it will be used as the target platform for the portation of OSE which is covered later in this thesis project and the platform for which a suitable memory management technique is chosen. This chapter is based upon Tilera’s own documentation [3,14,2], a paper on iMesh [15] by key members of the Tilera development team and one technical manual that is unavailable to the general public.

3.1 Architecture Outlines

The Tilera TILEPro64 is a manycore processor chip with 64 separate cores. These cores, or rather tiles as Tilera refers to them, are complete fully functional processors in that they are capable of running its own operating system, such as Linux. The tiles are arranged in a two-dimensional array of eight by eight tiles and are connected using a mesh-type interconnection network (see figure 3.1), consisting of five separate networks, developed by Tilera and called iMesh [15]. Set on the border on this array (on the same chip), and of course connected to the network, are memory controllers as well as remaining I/O interfaces.
3.2 Tiles

Each tile is, as mentioned, a complete processor in itself. In the configuration used by Tilera, however, further devices are required to communicate with the outside world and the other tiles. Therefore each tile contains three components: processor, cache and switch engines (see figure 3.2).

![Figure 3.2: Tile in the TILEPro64 processor [2]](image)

**Processor Engine**

Each processor engine is a 32-bit Very Long Instruction Word (VLIW) processor with a Reduced Instruction Set Computer (RISC) architect that has been extended to instructions commonly used in Digital Signal Processing (DSP). They have their own Program Counter (PC) as well as a Direct Memory Access (DMA) subsystem. The processor engine runs at 700 or 866 MHz frequencies.
### 3.3 iMesh

**Switch Engine**

The switch engine manages and routes packages through the six separate networks that make up the iMesh. The connections are logically independent, full duplex and flow-controlled. The switch engine also implements buffering and flow control capable of asynchronous communication. This ensures that the user does not have to worry about asynchronous/synchronous communication. The switch engine also enjoys tight access to the registers in the processor. This further reduces latency. For more information about iMesh see section 3.3.

**Cache Engine**

The cache engine contains the tile’s Translation Look-aside Buffers (TLBs), cache memories and cache sequencers. Each tile’s cache engine has 16KB L1 instruction cache, 8KB L1 data cache and a 64KB combined L2 cache. This provides the TILEPro64 with a total of 5.5MB on-chip cache memory.

Each cache engine includes a Direct Memory Access (DMA) engine. This can be used by an application programmer to perform several memory-related tasks, such as moving data to and from main memory and the L2 cache, and between tiles.

### 3.3 iMesh

All inter-tile and I/O communication is done on the Tilera chip over the iMesh network. This network consists of six separate networks, of which one is static (Static Network, STN) and the remaining are dynamic. The static network is not packeted like the others to allow streaming data between tiles at a lower latency. The routing of these streams is established at stream set-up and works in much the same way as a circuit-switched network. The remaining dynamic networks are packet routed, meaning that senders release a packet onto the network and then forget it. The routing is taken care of by the rest of the network in a dimension-ordered fashion, first in the X direction and then in the Y direction. The dynamic networks by design provide a deadlock-free transmission in one direction between nodes. But software can of course create circular dependencies at protocol level, creating deadlocks.

Other characteristics of the iMesh network on the TILEPro64 chip include: 32-bit, full-duplex network connections, single-cycle latency from the processor engine to the switch engine, single-cycle latency between adjacent tiles in the network, zero-cycle latency from the switch engine back to the processor engine and packet lengths from 1 to 128 words (32-bit word length), plus a header word.
3.3.1 iMesh Networks

The six networks that make up the iMesh are:

**UDN** The User Dynamic Network. Accessible by the user through a Tilera-developed API (Application Programming Interface). Hardware-wise this is deadlock free, but can still be deadlocked through careless programming.

**IDN** The I/O Dynamic Network. Mainly used for tiles to communicate with I/O devices as well as for I/O devices to transfer data to memory controllers.

**MDN** The Memory Dynamic Network. The MDN is used for memory related data transfers such as loads, stores, prefetches, cache misses or DMA, either between tiles or between tiles and the memory controllers. The cache engine on each tile has a direct hardware connection to this network.

**CDN** The Coherence Dynamic Network. The CDN is only used for cache coherency-related messaging such as invalidations.

**TDN** The Tile Dynamic Network. The TDN works together with the MDN to manage memory transfers. If a tile requests a cache line located on another tile, the request message is sent via the TDN, and the reply via the MDN. This prevents deadlocks in the memory system.

**STN** The Static Network. The STN is, as mentioned previously, unlike the other networks in that it is not a dynamic network. It is used for streaming data and is accessible from the user.

3.3.2 Hardwall protection

The two user-accessible networks, STN and UDN, are capable of isolating certain tiles from each other. This prevents application running on separate sets of cores from interacting with each other (see figure 3.3). This mechanism consists of a bit in a Special Purpose Register (SPR) of the UDN or STN switches that triggers an interrupt routine in the processor and takes any appropriate action such as rerouting the packet in another direction.

3.3.3 Packets and Routing

The data being transported across the networks (excluding the STN, of course) is sent in packets. Each packet contains a number of header bits (containing destination, size of packet etc) and a payload of databits. These packets are routed using a dimension-ordered routing policy, where a packet always travels in X direction first and then in Y
direction. The TILEPro64 chip can, at boot time, be configured to use XY routing or YX routing. By contrast, the STN network uses a per-tile pre-configured routing table.

3.4 Memory Architecture

The TILEPro64 chip memory addressing is based on a 36-bit physical address space, viewable from software as 32-bit virtual addresses. By default, on-chip hardware provides a memory coherent (see section 2.1.3) view of data memory to applications. Instruction memory, however, that is written by an application itself (self-modifying code) is not kept coherent by hardware. Instead, a special instruction (ICOH) must be used to enforce coherency between data and instruction memories. Furthermore, the hardware of the TILEPro64 also maintains cache coherency for I/O accesses.

3.4.1 Cache Architecture

Much like many commonly sold processors of today (such as AMD or Intel), the TILEPro64 has more than one layer of cache memory available. This strategy is commonly used to combat the significant speed differences between processor and main memory. One of the most important reasons is to prevent execution of stalling (stopping) due to long latencies from main memory. On the TILEPro64, however, the processor engine does not stall on load or store cache misses. Instead, the execution of consecutive instructions continues until the data requested by the cache miss is actually needed by another instruction. For more details on the cache memories of the TILEPro64 chip, see table 3.1. The cache architecture also implements a software-programmable hardware DMA engine and supports using portions of the L2 cache as a scratch-pad memory.
### Cache properties of the TILEPro64

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 instruction (L1I) cache</td>
<td>16 KB, direct-mapped</td>
</tr>
<tr>
<td>L1 instruction TLB</td>
<td>16 entries, fully associative</td>
</tr>
<tr>
<td>L1 data (L1D) cache</td>
<td>8 KB, two-way associative</td>
</tr>
<tr>
<td>L1 data TLB</td>
<td>16 entries, fully associative</td>
</tr>
<tr>
<td>L2 unified cache</td>
<td>64 KB, four-way associative</td>
</tr>
<tr>
<td>Latency (load to use)</td>
<td>2 cycles L1D hit</td>
</tr>
<tr>
<td></td>
<td>8 cycles local L2 hit</td>
</tr>
<tr>
<td></td>
<td>30-60 cycles remote L2 hit</td>
</tr>
<tr>
<td></td>
<td>80 cycles L2 miss to memory</td>
</tr>
<tr>
<td>Architecture</td>
<td>Non-blocking</td>
</tr>
<tr>
<td></td>
<td>out-of-order</td>
</tr>
<tr>
<td></td>
<td>stall-on-use</td>
</tr>
<tr>
<td>Line Size</td>
<td>L1I: 64B</td>
</tr>
<tr>
<td></td>
<td>L1D: 16B</td>
</tr>
<tr>
<td></td>
<td>L2: 64B</td>
</tr>
<tr>
<td>Allocate Policy</td>
<td>L1I: Allocate on read miss</td>
</tr>
<tr>
<td></td>
<td>L1D: Allocate on load miss only</td>
</tr>
<tr>
<td></td>
<td>L2: Allocate on load or store miss</td>
</tr>
<tr>
<td>Write Policy</td>
<td>L1I: N/A</td>
</tr>
<tr>
<td></td>
<td>L1D: Write through,</td>
</tr>
<tr>
<td></td>
<td>Store update on hit</td>
</tr>
<tr>
<td></td>
<td>L2: Writeback</td>
</tr>
<tr>
<td>Error Protection</td>
<td>L1I: 64-bit parity</td>
</tr>
<tr>
<td></td>
<td>L1D: 8-bit parity</td>
</tr>
<tr>
<td></td>
<td>L2: 8-bit parity</td>
</tr>
</tbody>
</table>

Table 3.1: A detailed table of the cache properties of TILEPro64 chip [3]

### 3.4.2 Dynamic Distributed Cache

The TILEPro64 chip implements a Dynamic Distributed Cache (DDC) system. This provides each tile with a virtual L3 cache that is distributed among all of the tiles rather than a physical L3 on each chip or in a central storage. This distributed cache is fully coherent across tiles and scales due to its decentralized structure. On top of the DDC, Tilera have implemented a shared memory mechanism, called Dynamic Distributed Cached Shared Memory. This provides a hardware-managed, cache-coherent approach to shared memory. The DDC allows a page of shared memory to be placed (homed) on a specific tile or distributed across many tiles, then cached remotely (used) by other tiles. This mechanism allows a tile to view all tile’s cache memories as a large shared, distributed coherent cache. This form of shared memory access can be useful when processes read and write shared data in an interleaved manner such as locking or synchronization objects.
3.4.3 Memory Balancing and Striped Memory

The TILEPro64 chip is equipped with four memory controllers (as shown in fig 3.1). To balance load between these controllers, the Tilera chip has the ability to stripe the memory. This means that the physical pages in the memory are split in to four. Or to illustrate it by a simple example: a page of 64MB would have its first quarter of the page accessible through memory controller 0, the second quarter though memory controller 1, the third quarter though memory controller 2 and the last quarter through memory controller 3. The load would thus be balanced between the memory controllers. In addition, the tiles by map to the nearest memory controller by default, effectively dividing the chip into four quadrants with each set of 16 tiles using its nearest memory controller. This memory striping feature may, however, be deactivated, leaving the task of load balancing the memory controllers up to the user (or OS).

3.5 Conclusions

The TILEPro64 chip introduces a novel solution the problem introduced at the beginning of chapter 2, namely to increase the number of processor cores that works in unison rather than increasing the frequencies at which each core work. Tilera has solved this by equipping the TILEPro64 chip with 64 cores, each operating at 700 or 866 MHz. These cores are in themselves capable of running a complete OS but Tilera has also equipped the chip with not one, but six individual networks in which the cores can communicate and interact. Memory-wise, these 64 cores have to share access to the main memory over four individual memory controllers, or have 16 cores each sharing one controller each. Every core on the chip is equipped with a private Level 1 cache and a shared Level 2 cache. The sharing-aspect introduces the possibility of having a core using another core’s Level 2 cache as a virtual Level 3 cache. Further conclusions on the memory management of the TILEPro64 chip in the context of OSE will be drawn in chapter 6.
Chapter 4

ENEA OSE

ENEA OSE (Operating System Embedded) is a distributed real-time operating system developed for embedded systems by ENEA. It is based on a microkernel architecture and synchronous direct message passing as the primary inter-process communication (IPC) mechanism. OSE has been developed to support hard real-time as well as soft real-time code and POSIX code, clustered, time- and event-driven code. The microkernel allows OSE to execute on heterogeneous distributed systems and clusters and supports pre-emptive, priority-based scheduler, prioritized interrupts and processes. The microkernel leaves out much functionality that would normally be found in a kernel, but ENEA have solved this by breaking down OSE into separate layers of services and extensions. Figure 4.1 illustrates these layers. This chapter is based on Enea whitepapers [16, 13, 12] and several technical manuals and documentation that are unavailable to the general public and will go through some fundamental and – for this thesis project – relevant components of OSE, such as memory management and multicore/mannycore support.

Figure 4.1: The layers of OSE
4.1 OSE Architecture overview

4.1.1 Processes, blocks and pools

The most basic building block of OSE is the process. This corresponds to threads, tasks and light-weight processes in other operating systems. OSE uses a classification of processes to dictate priorities. Above processes, OSE uses the abstraction of blocks. A block in OSE is several processes grouped together in order to form a subsystem. Externally, a block behaves much like a process. It can be started, stopped, killed and share the same memory pool. Above that, blocks can be divided into domains together with pools and heaps as illustrated in figure 4.2.

The memory structure in OSE is referred to as pools. A pool is an area in memory in which signal buffers and process stacks are located. There is always at least one global pool from which processes and blocks may allocate memory. This is not the only way in which a process or block may allocate memory in OSE, but it is the fastest. It is deterministic in time and fast enough to be used for interrupt service routines. These allocations have been designed to result in as low memory fragmentation as possible, which is crucial when executing on embedded systems that may run for extended periods of time.

![Figure 4.2: Processes in Blocks with Pools in Domains](image)

4.1.2 Signals

OSE has a few inter-process communication mechanisms, for instance mutexes and semaphores. The most common way for processes on OSE systems to communicate is, however, by using signals. This technique is sometimes referred to as message passing, as the signals themselves are messages of varying size and content. They may contain data of many kinds (structures, records etc.) and the sending itself is very effective. When a message is sent, the only data that is actually being sent from one process to another is a pointer. The syntax and semantics used are the same regardless of memory...
4.1 OSE Architecture overview

4.1.3 Core components

As mentioned, OSE is a distributed operating system built on a microkernel bundled together with several optional as well as non-optional key components. Figure 4.3 illustrates non-optional components in black text and optional components in grey.

![Figure 4.3: The OSE Kernel and Core Components](image)

The OSE kernel itself consists of the Executive and the Bios. The executive component handles such key features as real-time scheduling, signalling, interrupt management and more. The Bios is used for applications that performs calls to the Executive without knowing its actual address. The memory manager (MM) is a component that handles all key memory management, such as allocation of memory for pools and programs, memory mapping, memory protection and cache management. The CPU hardware abstraction layer (CPU HAL) is an abstraction layer which eliminates variations in processors (e.g. from different vendors) that share the same basic architecture (PowerPC, MIPS, ARM). The Executive uses this to detect "CPU flavour" and perform certain CPU-specific operations. It also provides the MM with cache-related information such as sizes, TLB sizes and debugging support.

4.1.4 Portability and BSP

The bottom layer in the OSE architecture (see figure 4.1) is the Hardware Abstraction Layer (HAL). This layer is the target interface of the OSE kernel and provides device
drivers and all hardware functionality for OSE. This layer also includes a component of OSE called Board Support Package (BSP). This component contains system initialization code and device driver source code. The BSP is the component that needs most work in the upcoming processes of porting OSE to TILEPro64. Upon obtaining a copy of OSE, the BSP is not included by default. It is instead treated as a separate component that needs to be configured for each new platform used.

4.2 OSE Multicore Edition

When designing OSE MCE, Enea has chosen to use neither SMP nor AMP models (see sections 2.2.1 and 2.2.3) strictly. Instead Enea has tried to design a hybrid of both models where the goal has been to design a kernel that looks similar to SMP when it comes to simplicity, flexibility, application transparency, and error tracing, and similarities to AMP when it comes to scalability, determinism and performance [16].

As with the AMP-model, each core can in OSE MCE be loaded with a OS image running an own scheduler with associated data structures. The shared resources are controlled by a master core that executes nothing but kernel threads. By letting the other cores request shared resources via a specialized kernel IPC dubbed kernel event, OSE MCE becomes a highly distributed system with SMP-like behaviour in respect to shared resources.

4.3 Conclusions

OSE MCE is fundamentally a highly distributed operating system with very efficient IPC mechanisms as well as a high degree of fault tolerance [16]. The IPC has been optimized in OSE MCE on the sender and reciever ends, but this naturally puts certain demands on the core interconnection networks. They need to be as quick as possible and deadlock-free to remove any need for retransmission and similar mechanisms. So to answer the third and final question in section 1.3.1:

• How will OSE’s message-passing technologies cope with a manycore platform?

As long as the underlying hardware is fast and dependable enough (in terms of determinism and deadlock-free), OSE MCE’s message passing technologies should have no problem migrating to a manycore platform. In the context of this thesis project, with regard to the Tilera platform, the signalling used between cores needs careful attention, since it is possible to deadlock parts of the interconnection network (see section 3.3.3). Enea AB has no formal specifications or demands on the hardware in regards of the message passing technologies.
Chapter 5

Memory Management of Manycore Systems

Manycore and multicore processors is a field in which much research is still being done. Because of this, this chapter aims to enumerate both hardware- and software-based techniques for improving memory management in multicore systems.

5.1 Software-based techniques

This section will look at what the areas in which research is being done to improve memory-management for multicore and manycore systems, with focus on the software side as well as on cache-based techniques.

5.1.1 Managing Cache

The exact configuration of caches on multicore/manycore chips may vary greatly, but the configuration used by Tilera in TILEPRO64 (see chapter 3) is to equip each core with a private Level 1 (L1) cache and a shared Level 2 (L2) cache. The most common way of managing the L2 cache is to use private caching or shared caching [7]. In a private cache managed processor, each cache slice is associated with a specific core and replicates data freely as the processor accesses them. Shared caching acts and appears as a logical single cache. In [7] it is argued that these schemes do not scale very well, because private caching does not directly exploit the on-chip available cache storage, and a shared scheme could suffer access latencies in a chip with a large enough number of cores or more importantly a shared cache suffers from interference problems. A poorly written program can evict L2 cache content belonging to another co-scheduled program causing delays that could be serious in an RTOS-system. The usage of the cache system is of great importance in any system, but as the number of cores scales it becomes an ever greater factor in the performance of memory management.

Cache Bins

The authors of [7] strive to improve the usage of L2 in single-threaded programs by implementing a location aware data placement scheme for use on a shared cache. This is done by effectively dividing the L2-cache into bins (where a bin is a set of cache lines large enough to contain an entire memory page) and optimizing the usage of these. The authors propose two schemes, Dynamic and Static 2D Page Coloring, in which allocation of memory pages is decided either prior to run-time (this being the Static variant) or during run-time (the Dynamic variant) to belong to specific cache bins. The static scheme relies on accurate analysis of a memory trace of programs to determine
allocation of cache bins. The dynamic scheme allocates bins by keeping track of the cost of each bin. This is done by monitoring cache bin hotness (essentially measuring the number of hits and misses) and deriving a cost thereof. New memory pages are then simply allocated to the bin with the lowest cost. The authors simulate these schemes and compare them to the previously mentioned private and shared caching schemes, and show significant improvements when using Dynamic 2D Page Coloring.

Cache Equalizing

Cache management can be improved and implemented in many different ways (as shown throughout this section) in software. There are, however, some methods that do include hardware modifications. One of the more recent contributions [17] is an effective method of balancing, or equalizing, the load of L2 caches by implementing Cache Equalizer (CE). The problem that CE alleviates is that in a system where the cache memories are shared, the usage of the core’s L2 caches is not necessarily evenly distributed and could cause a set of fewer caches to suffer a high degree of misses (or pressure). The CE solution proposed in [17] does require some additions to hardware, so it is not immediately implementable on any multicore system based on private L1 and shared L2 caches. The extra hardware would consist of an array at the memory controller(s) that would record and keep track of the current pressure of all L2 caches on the chip. Upon fetching a block from the main memory, CE would then check the pressure array to identify the L2 cache(s) with the least amount of pressure and place the block at that/those cache(s). The pressure is measured by CE as the number of cache hits during a given time interval (referred to as epoch in [17]). When simulating this scheme on a 16-core mesh-connected multicore processor, the number of cache misses is reduced by an average of 13.6% and at most 46.7%.

Figure 5.1: Cooperative Caching [4]
Cooperative Caching

Using private L2 caches as a design point, the authors of [4] propose a scheme called *Cooperative Caching* (CC). The authors favour some of the benefits included in using private caching, including the latency benefits (execution-wise) and the reduction in bandwidth usage for the interconnection network. Therefore the authors aim to make private L2 caches cooperate and act like a collective shared cache via cooperative actions (see figure 5.1 where the shaded area represents the aggregate shared cache formed via cooperation). For example, cooperative caches will attempt to use remote L2 caches, if spare space is available, to store (and serve) data that would not fit in the local L2 cache. Beyond using available space elsewhere on the chip, CC will try to only evict lines from a local L2 cache (and "spilled" onto another) that are replicated elsewhere on the chip. If none are available, more commonly used eviction schemes are used (such as LRU). Lastly, CC will extend the LRU-eviction scheme to be used globally on the entire chip. Using these three methods, Cooperative caching can reduce the execution time of simulated workloads by 4-38%, and performs at worst 2.2% slower than the best of the private and shared caches in extreme cases.

OS-Level Page Allocation

In [18] the authors argue that a shared caching scheme can potentially achieve better overall usage of the available on-chip L2 cache than a private scheme because memory blocks and subsequently accesses are finely distributed over a large caching space. The authors base their work on two key observations; finely interleaving cache lines to physically distributed L2 cache slices is not good for program-data proximity, and OS-level page allocation can directly control where, among many L2 cache slices, cache lines are placed if data to L2 cache slice mapping is done at the memory page granularity. Using these observations, the authors of [18] mainly strive towards achieving better data proximity as well as relieving highly pressured (used) L2 caches by better allocation of pages. The decision to implement this at the OS level is to achieve a flexible solution that does not require complex hardware to run.

User Level Cache Control

The authors of [5] take a very different approach to the task of optimizing cache performance. In [5] a special library (User Level Cache Control or ULCC) is proposed to give the programmers control over cache allocations (see figure 5.2). The library interface would require some input from the user regarding the memory environment of the target system and two underlying components that manage and allocate the various requests made by the programmer. Lastly, a memory management unit would run in a separate thread to handle allocations, garbage collect the physical memory pages. When used by a skilled programmer, the authors of [5] do show good benchmarks of

\[^1\]Least Recently Used
ULCC. The drawback of course being that programs would need considerable amounts of code re-writing to fully attain the possible performance gain made possible by ULCC.

**Scalable Directory Architecture**

When designing a shared-memory multicore or manycore processor, a major obstacle to overcome is that of maintaining full cache coherency, in order to keep multi-threaded applications from interfering or destroying each other’s data. One common way of implementing cache coherency is through a directory-based model. This is when the sharing status of a block of physical memory is kept in just one location, referred to as the directory [19]. The authors of [20] focus on the scalability of a mesh-interconnected multicore processor and propose a new directory-based protocol in which each core is equipped with an L1 cache, an L2 cache and a small directory. This directory would contain a number of entries of memory lines and the sharing cores. If an update is triggered to a memory line that is in the directory, invalidations will be sent to all sharing cores. Data lines of the L2 cache would, however, not be written back to memory until it is replaced. The increased number of invalidation messages sent over the interconnection network is substantial, but the authors contend that it is a viable tradeoff to reduce off-chip traffic. Though the scheme proposed in [20] requires additional hardware modifications, the authors do simulate a variant in which the directory is located within the L2 cache, thus relinquishing any hardware demands. The authors move on to simulate their proposal with a number of different entries in the directories as well as the unified L2-directory variant. The simulation results were positive for many different program characteristics as well as scaling. Although the unified L2-directory variant performed worse, this was due to the inescapable reduction in the capacity of the L2 cache.
5.1 Software-based techniques

5.1.2 Malloc

Some researchers believe that memory management can be improved upon for multicore/manycore platforms even without serious rewriting of code [21]. This is achieved by modifying the way a system allocates and frees up sections of memory, most often by analysing and modifying malloc, the library functions within the C-programming language.

Both [22] and [21] test the ideas of adapting malloc for a multicore or manycore environment. [22], however, do so in the context of Graphical Processing Units (GPUs) and SIMD-based CPUs. The idea proposed in [22] is a malloc designed for heavily multi-threaded applications and exploiting SIMD-capabilities by aggregating allocation requests. By implementing this, the authors do reach an implementation that scales very well as the number of threads increases. Whilst an interesting solution, it does introduce more requirements on the processor that may not be possible or feasible in all situations.

In [21] the authors experiment with simply wrapping the existing malloc and free in a new library, and running this in a separate thread. They try lock and lock-free based versions of this, and without modifying benchmark source code, a performance gain of 2-3% was achieved. Although this might seem small and insignificant, it is an interesting accomplishment without serious code-rewriting.

5.1.3 Transactional Memory

As mentioned in recent literature [1, 23], an emerging concept for multicore/manycore systems is that of Transactional Memory (TM), where a memory transaction is a sequence of memory operations that execute atomically and in isolation. Atomically in this context means that all of the operations or none are executed, and in isolation means that the effects are not visible or modifiable by any other transaction until it has been committed. In short, TM is a mechanism to enable concurrent updating of shared memory locations: a thread is allowed to progress with its own updates until it completes its transactions. At that point, a check is carried out to find out whether any other concurrent accesses took place on the same memory locations; if not, all the changes are atomically committed and made visible to other cores/threads; if the memory areas modified by the transaction have been modified elsewhere, the complete transaction is rolled back and re-attempted [1].

There are basically three ways to implement TM: hardware only (called Hardware Transactional Memory or HTM), software only (Software Transactional Memory or STM) or a hybrid solution, combining both hardware and software solutions. An STM

\(^{2}\text{Single instruction, multiple data (SIMD), describes CPUs with instructions that can perform operations on multiple elements simultaneously.}\)
solution would mostly consist of heavy compiler modifications to set up the various barriers needed for TM to work. Although STM does not put any constraints on underlying hardware to work, the overhead remains relatively high \cite{23}. A HTM system outperforms both STM and hybrid solutions but introduces a serious shortcoming: the read/write buffers required for transactions have a finite capacity and may overflow on long transactions. Introducing a length restriction on transactions is unacceptable from a programmer’s perspective, so further firmware-based mechanisms are necessary \cite{23}. Furthermore, HTM would rely on cache coherency protocols for conflict detection and hence reducing the overhead introduced by STM.

Newer papers \cite{24} do, however, contend that TM has not reached a major breakthrough because of significant hardware demands for high performance, and that STM is not deterministic. This is a key feature that is desirable when designing an RTOS. This claim is backed up somewhat in \cite{25} who analyse STM in a Haskell programming environment. In this paper, the authors analyse the performance of STM on Haskell applications, in a Haskell Runtime System (RTS) and on a garbage collector. The results indicate unnecessarily high allocation rates and memory usage. The authors of \cite{25} do, however, move on to propose some areas of future work for optimization and research and the authors propose using a novel solution to locks or TM. This work will be discussed further in section 5.1.4.

5.1.4 Shared memory without locks and Transactional Memory

The author of \cite{24} contends that, due to their two-pieced nature (requires two different pieces of program code), locks can lead to dead- and livelocks (for more on dead- and livelocks, please see \cite{11}) and that TM (see section 5.1.3) scales poorly, and lastly that both TM and locks are neither predictable nor deterministic. The author instead proposes a system in which access to a shared memory location is always done by one and the same processing core which will perform the requested operation (according to a configurable policy) on behalf of threads requesting access to that very same memory location. This would be implemented as User Processing Entity (UPE) and Resource Guardian (RG) where a UPE would be a core executing program and an RG would be a separate core guarding each section of shared memory. A core acting as an RG may still execute other programs and access shared data, but must naturally contact the designated RG for the portion of shared memory used, whether or not RG is the own core or not. The author of \cite{24} unfortunately does not simulate or analyse this implementation and has only provided some basic empirical results. These results where however promising. When comparing locking and initiation of transactions to STM, better or at least equal performance were achieved. When it comes to memory usage efficiency and cache performance, the proposed method shows an improvement compared to traditional systems. The gain varied according to the size of the shared data area, the size of the cache and the size of critical sections, but it was, in an experimental setup, at least 50% and sometimes up to 200% (in the case of small
5.1 Software-based techniques

Caches and data areas larger than the cache size). It must however also be noted that this implementation is not yet finished and further work is needed.

5.1.5 Superpackets

The authors of [26] do base their work on multicore GPUs, but the idea proposed is nonetheless relevant in the context of this thesis. The authors propose that memory requests be throttled at source by congestion-aware memory scheduling as well as grouping together several memory requests into single packets, or Superpackets, rather than sending single-request packets, thereby effectively reducing the load of the interconnection network. Preliminary results of combining throttling at source and superpackets are promising, but it should be noted that this was simulated on a GPU rather than a heterogenous multicore processor.

5.1.6 Routing of interconnection networks

In [6], the physical placement of memory controllers and ports is discussed (more on this in section 5.2.1), as well as the impact of routing algorithms used to transport data packets throughout the interconnection network. The authors propose a Class-based Deterministic Routing (CDR) algorithm to be used in mesh networks, as opposed to the more commonly used XY (where packets are first routed in the X dimension followed by the Y dimension) or YX (where packets are first routed in the Y dimension followed by the X dimension) algorithms. The O1Turn algorithm [27] is also discussed and compared in [6], where each packet, at source, is randomly selected to use either XY or YX routing. The CDR algorithm proposed in [6] tries to take advantage of both XY and YX routing, but the message is taken into consideration. That is, a requesting message packet uses XY routing, and the replying message uses YX routing. These algorithms are put to the test in the paper through simulation, and the conclusion is argued that the O1Turn algorithm [27] is unable to load balance on a two-dimensional meshnet and that the CDR algorithm will improve uniform random traffic by 56% on a Tilera-style chip.

However, as discussed in [23], when designing and selecting a routing algorithm, its effect on delay, energy-usage, throughput and reliability must be taken into account. If an excessively complex, and thus time-consuming, algorithm is chosen, the effects on memory performance as well as the processor as a whole can be reduced.
5.2 Hardware-based techniques

 Whilst being outside the scope of the upcoming portation work of this thesis project, it is worth mentioning some of the work that is being done on the hardware side of manycore processors and how it might increase performance in the memory systems.

5.2.1 Placement of memory controller ports

In [6] it is discussed that the physical placement of the memory controller ports around processor cores impacts on the load and could even congest the interconnection network of the chip, thereby increasing latency in memory requests and possibly lowering the system’s performance. The exact number of memory controllers was not as important in this paper as the number of ports and their placement. The authors start out from a Tilera-based design with a square grid of cores, and propose, simulate and evaluate six different layouts, including such novel placements as diagonal, diamond and checkerboard. After using a "link contention simulator" the patterns remaining were row0_7 (based on the Tilera-chip), diagonal X and diamond (shown in figure 5.3).

By simulating various loads and examining the network usage, the authors conclude that the diamond configuration (figure 5.3c) reduces the maximum channel load of the mesh-network by 33% from the baseline (and Tilera-inspired) row0_7 configuration (figure 5.3a). The conclusion reached in [6] is a very interesting design aspect to include when designing manycore processor chips. Not to mention the relative low cost of simple electrical wiring on chips [13].

5.2.2 Additional Buffers

For multi-processors based upon shared memory (see section 2.1.3, the authors of [28] propose a particular hardware buffer to be placed beside the memory controller to
5.2 Hardware-based techniques

handle synchronization tasks. The focus is mainly on the most common busy-wait synchronization techniques (e.g. mutual exclusion or barrier synchronization) and events. This buffer, or Synchronization-operation Buffer (SB) as the authors call it, would be searched every time the memory controller has been issued with a locking or unlocking call, and the corresponding processor would be informed. This buffer was then simulated by the authors in a multicore processor of 4, 8 and 16 cores (in a mesh interconnection grid) and the results showed a 40% performance improvement and 30% energy reduction in comparison to a coherency protocol-based system. The authors do, with good simulation results, contend that this technology will scale well, but they lack simulations beyond 16 cores.

5.2.3 Memory Access Scheduling

Most modern DRAM systems rely on memory controllers that use out-of-order scheduling to maximize row access locality, or in other words, memory accesses will be scheduled on adjacent rows to achieve high DRAM bandwidth. A great deal of interesting work is being done on the subject of memory accessing and scheduling (\cite{29,10}), but to better suit the conditions of a multicore environment, the authors of \cite{30} suggest that changes in hardware be made for increased performance.

The authors of \cite{30} contend that as the number of cores increases in a processor using a shared memory system, multiple memory access streams (from each core) interleave and interact in such a way that they will frequently destroys the inherent row access locality present in each individual access stream, effectively reducing the DRAM efficiency. The authors of \cite{30} propose to solve this by modifying the interconnection network to include an arbiter scheme that would try to obtain full use of row access locality in the DRAM. The authors propose and analyse two different methods of network arbiters on ring, mesh and crossbar interconnection networks. The authors conclude from their work that it is possible to attain 90% throughput of that achievable with memory-limited applications (applications that work more with memory, than computations). It can be noted that much of the research in this article is done with multicore Graphic Processing Units (GPUs) in mind, rather than homogeneous multicore CPUs. The conclusions drawn in \cite{30} might therefore suit GPUs, whose tasks are similar, than CPUs, whose tasks may vary greatly.

5.2.4 Scratchpad Memory

A scratchpad memory (SPM) is an alternative to cache memories that is faster, uses lower on-chip area and is more energy efficient \cite{31}. The authors of \cite{32} experiment with the idea of replacing all cache memories in a multicore processor with SPM (in an embedded systems environment) and continue with optimizing task scheduling and mapping for this suggested platform, the motivation being speed and power consumption relative to cache-equivalent system. Scheduling and mapping are examined because
the burden of allocating data to an SPM lies with the compiler. The authors of [32] do reach some interesting conclusions and show that much performance can be gained through careful optimizing, but unfortunately lack any comparisons to a cache-based system. Therefore it is difficult to draw any conclusions of relevance to this thesis from this article alone.

5.3 Conclusions

This section will go through all the previously mentioned techniques, both software and hardware, and finally return to and answer the questions posed in section 1.3.1.

5.3.1 Software-Based techniques

Managing the cache effectively is crucial in any environment, and so far we have seen cache-management techniques that can be implemented at firmware-level on chips, and some that are implementable at OS level. When migrating from multicore to many-core processor design, a key feature to bear in mind is scalability. Or, more precisely, maintaining performance and keeping latencies low even as the number of cores grows. The concepts of CB, CE and CC all try to tackle the same problem: when using shared L2 caches, trying to reduce the load (usage) of high-pressured cores. CB manages this purely by software and it scales very well in the proposing paper [7] beyond 1024 cores. CC and CE use software-based and hardware-aided methods respectively. [17] shows that CE outperforms CE by 8%, and more interestingly, that CC spills blocks to neighbouring L2 banks without knowing if spilling helps or hinders performance. Thus CE would be preferable if the option of modifying hardware exists. Comparisons between CE, CC and CB are however lacking and would be useful knowledge if the target hardware is fixed. OS-Level Page Allocations could be a feasible alternative for fixed hardware targets. If the option exists to implement this in an OS, this solution would be very interesting, although it would be time-consuming to alter the memory management unit. ULCC is an interesting approach but does, as previously mentioned, require extensive code rewriting when porting existing programs to a multicore/many-core platform. When developing new programs from scratch, however, this could be a viable solution. Furthermore on cache, regardless of whether the option to modify hardware exits, the Scalable Directory Architecture proposed in [20] is a very scalable and interesting proposal. It does however come at the cost of higher usage of the interconnection network. So as long as the capacity of the network is good enough, it is a good method of increasing cache usage.

Moving on to other software-related ways of improving memory management, modifying malloc is an issue to be considered. In some cases, this can provide good scalability as well as performance boosts. But the problem is just that. It has so far only been tested in the context of GPUs and the types of computation that concern
graphics. Further studies are required before this becomes a realistic alternative.

Transactional Memory is discussed at length in a previous section and it may be concluded that the software version STM has some serious limitations that need to be dealt with, and that the hardware version HTM is promising, but needs heavily modified hardware changes to be implementable.

An alternative to TM is introduced by [24] in the form of shared memory without locks and Transactional Memory. This is very much work a in progress and far from finished, but it is a very interesting idea as the number of cores scales to even greater numbers and coordination becomes a greater issue.

Regarding the usage of the interconnection network, the concept of Superpackets is a small and good one. Although only tested so far on GPUs, any contribution to reducing traffic on the interconnection network should be considered. The routing can also provide much difference as shown by [6]. But as long as the underlying hardware does not support any complex algorithms or at least the possibility to change a general routing algorithm during run-time (the TILEPro64 does not, as discussed in section 3.3.3), not much can be done though software alone.

5.3.2 Hardware-Based techniques
As discussed and analysed at great length in [6] the placement of memory controller ports can seriously impact the load on the interconnection network. The conclusions reached in [6] are indeed interesting and should be considered when designing a multicore or manycore processor, but the cost of extra memory controllers and the pure feasibility (electrical-wise) of re-arranging the controller ports as proposed are not discussed in the paper.

The usage of additional buffers on the memory controllers does not require particularly invasive changes to be made to the hardware. The results are good, but simulations or any indications on the scalability of the solutions are lacking.

The authors of [30] suggest that the memory controllers re-order incoming requests to better maintain row access locality. Though the paper simulates this in a GPU environment, the problem is very much relevant even in a homogeneous multicore/manycore processor running multiple programs and threads at once. Further studies are required to examine whether this solution is feasible on a more common processor with more irregular memory loads, but the idea is nonetheless interesting.

Scratchpad Memories are a novel idea in themselves if the promises in terms of speed and power consumption can be delivered. This is very much an area of ongoing
work and cannot therefore be considered at this time when designing a multicore/many-core processor.

5.3.3 Answering Questions

The two first questions asked in section 1.3.1 were:

- What appropriate memory management techniques exist for manycore platforms and how can existing memory management techniques be adapted for manycore/multicore systems?

- How do memory management-techniques for manycore systems cope with the demands of a Real-Time Operating System (RTOS)?

Before the first question can be answered, some remarks need to be made: techniques that are still unfinished or untested on regular multicore/manycore environments (not GPU) will be excluded. These techniques are not in their current state appropriate per se, but were still relevant for the general discussion of this chapter. Hardware based techniques will not be excluded because, although irrelevant for the latter part of this thesis project, they are relevant in this context due to their potential impact on memory performance.

To answer the first question, appropriate and adaptable techniques are: Cache Bins because of the proven scalability and positive simulation results. Cache Equalizing because of its efficiency when coupled with the required hardware changes. OS-Level Page Allocation because of the flexibility provided at OS level, when hardware is insufficient. User Level Cache Control because however much code rewriting that it may imply, it could provide one more optimizing step if one is pressed on performance. Scalable Directory Architecture because although it produces extra load on the interconnection network, simulation results were positive and it scales impressively. Routing of interconnection networks because of the simplicity of the idea of reducing interconnection network load by different routing. Lastly, the only appropriate pure hardware-related techniques presented in this thesis project is placement of memory controller ports. Although some electrical considerations are needed, the results are nevertheless significant.

To answer the second question, any mechanism or technique that will aid a system in producing quicker and more correct computations in a deterministic and safe way should in theory be able to cope with an RTOS. However, it would be advisable to run further tests and simulations before launching such a sensitive system as an RTOS on a multicore/manycore platform.
Chapter 6

Memory Management on TILEPro64

This chapter returns to the memory technologies mentioned in chapter 5 with the goal of proposing one for implementation in OSE to run on the TILEPro64 chip.

6.1 TILEPro64

Looking back to the technical prerequisites of the TILEPro64 described in chapter 3 and the memory management techniques demonstrated in chapter 5, the sheer number of available techniques will have to be pruned.

The Cache Bins discussed in section 5.1.1 is simulated in environments with 64 (and more) cores and performs well under those circumstances. The OS-level Page Allocation and ULCC (as discussed in sections 5.1.1) are both very much implementable on the TILEPro64 chip. The Scalable Directory Architecture mentioned in section 5.1.1 is implementable in hardware or software alone. Interestingly enough, the TILEPro64 chip already implements a distributed directory in each tile. The extra network traffic this scheme would induce is alleviated by the fact that the TILEPro64 chip benefits from having dedicated networks to deal with cache coherency.

All of these techniques do require more or less invasive modifications of memory management unit in the OS running on a TILEPro64 chip. The hardware alone is capable of supporting them.

Albeit a highly interesting technique when designing manycore chips, the CDR mentioned in section 5.1.6 is not applicable on the TILEPro64 chip (the target system for the implementation part of this thesis project) since this uses strictly XY or YX routing (defined at boot-time) that cannot be altered once the system is running.

6.1.1 Discussion

The previous section concluded that three of the techniques shown in chapter 5 are deemed feasible to be implemented on the Tilera-platform. These are: Cache Bins, ULCC and Scalable Directory Architecture. Of these, the Tilera-team have already implemented a version of Scalable Directory Architecture in the TILEPro64 chip. ULCC is certainly interesting in itself, but places control in the hands of the programmer, rather than the operating system. The effective performance gained is directly correlated with the skill of the programmer. Therefore Cache Bins remains as the one memory management technique suitable for implementation on the TILEpro64 chip.
6.2 Cache Bins

From an architectural standpoint, Cache Bins is feasible on the Tilera platform simply because the authors of [7] base their experiments and simulation on a tiled, mesh-interconnected structure of cores equipped with a switch, L1 and L2 caches (see figure 6.1). This is remarkably similar to the architecture of the TILEPro64 chip which is discussed in chapter 3. The authors of [7] propose two variants of Cache Bins: a static variant and a dynamic variant. (These were covered previously in section 5.1.1.) The dynamic variant uses less complex algorithms to keep track of bin hotness (or effective cost) than the static variant and relies more on counters than previously collected statistics (for each program). The dynamic variant is therefore the most interesting one to simulate and analyse on the Tilera platform as it is more capable of running dynamic processes than a statically configured system.
6.3 Conclusions

The layout and usage of dynamic 2D page coloring (the dynamic variant of Cache Bins) is demonstrated in figure 6.2. The Global Bin Info is a table that contains as many lines as there are cores in the processor. These lines only contain the optimum Bin (in terms of cost) for each core and are kept updated at regular intervals when the cores send each other update messages. This table would however need to be implemented on a number of lines in the L2 cache, effectively reducing the capacity to some extent.

Implementing Cache Bins in the context of this thesis project is not a trivial task. But it is not impossible since OSE does not actually manage the L2 cache in any way, so implementing Cache Bins is indeed possible without rewriting or modifying OSE’s MMU to any extent. But before undertaking this task, a very deep understanding of the cache-functionality of the TILEPro64 chip as well as OSE’s MMU is needed.

The potential gains of implementing Cache Bins on the Tilera platform are promising considering the results demonstrated by the authors of [7]; an average of 32% improvement over Shared Caching (the scheme TILEPro64 uses by default). The result will vary depending on what applications are being run in terms of SMP/AMP systems.

6.3 Conclusions

This chapter returned to the memory management technologies discussed in chapter 5 and pruned the alternatives with regard to Enea OSE and Tilera’s TILEPro64. The one technique that adheres to the prerequisites of this thesis project was found to be Cache Bins. This is because the general idea of Cache Bins was constructed around an architecture very similar to that of TILEPro64, the fact that Enea’s OSE leaves room for components to manage the cache if necessary and the potential performance gains demonstrated by simulations in [7] are promising.
Chapter 7

Porting Enea OSE to TILEPro64

The main objective with in this chapter is not to reach a final working implementation of OSE on the TILEPro64 chip, but rather to work towards the point at which the proposed memory management technique could be implemented.

7.1 Prerequisites and previous work

Before any work on the port could begin, previous work and prerequisites needed to be studied. The work done by the author of [9] was used as a stepping-stone. The goals reached by [9] were:

- Setting up a build environment This meant setting up and building OSE core libraries as well as the Tilera simulator.

- Build a Coresys version of OSE, launch it in the simulator and write to the ramlog. Coresys version of OSE is a minimalistic build with basic features for testing. And ramlog is simply a log accessible through the RAM memory.

The point at which the author of [9] left off, is a system in which only one processor core is running, MMU disabled and no driver support. This means the system cannot communicate with the outside world in any other way than writing to the ramlog, and having no timer means that the system is unable to use any time-related system calls or even time-based scheduling. As long as event-driven scheduling is used, however, the scheduler will work.

The environment set up to work in is a development server at Enea running SuSE-Linux, ClearCase for version handling and for the purpose of this project, Tileras own MDE (Multicore Development Environment) which includes a simulator capable of simulating the hardware of the TILEPro64 chip.

7.2 Design Decisions and Goals

The goals and design decisions where made in cooperation with the author of [8]. The main goal was decided to be to work towards a fully functional OSE build as possible in a single-core mode, and from that point, expand and engage more cores. This simply to ensure quality over quantity. It was decided that the author of [8] would focus on device drivers (Timer and UART\(^1\)) and that the focus of this thesis project would be the

\(^1\)Universal Asynchronous Receiver/Transmitter. Used for serial communication.
HAL. If time had permitted, this thesis project would also have begun on implementing Cache Bins in the MMU. The reason for postponing the implementation of MMU to such a late stage is simply that OSE can be configured to run without one and without basic interrupt functionality or drivers, the OS cannot make much use of the MMU.

The goal of the implementation-part of this thesis project is thus continuing as far as possible with completing the Hardware Abstraction Layer. There was a focus from an early stage on ensuring that interrupts were working properly since this would affect [8] as well as this thesis project. The design decisions made together with the author of [8] are shown in table 7.1.

### 7.3 Implementation

#### 7.3.1 Implementing Cache Bins

Previous chapters (chapter 6) have covered the feasibility of implementing Cache Bins on the TILEPro64. In more practical terms, OSE does, as previously mentioned, not manage cache allocations to any extent. This leaves headroom for Cache Bins to be implemented. The TILEPro64 is equipped with 64 KB of L2 cache to be used for bins with their respective counters. The size of these bins depends on the size of memory pages used by OSE, which is 4 KB. The size of the counters needed at each bin is however not easily defined prior to testing. Without knowing the average lifetime or number of accesses a bin may need, this variable must be left some space for tweaking during testing and analysing. Beyond the bins, the L2 cache also must facilitate the global table used for storing information on the other tile’s current bin usage.

Before any significant design decisions can be made on implementing Cache Bins in OSE in the context of this thesis project, the system much reach a point at which it is even possible to enable and make use of a MMU. This step had not been reached at this point (as demonstrated in section 7.1), so work proceeded with the Hardware Abstraction Layer.

#### 7.3.2 Hardware Abstraction Layer

This layer was described in section 4.1.4 and is located in the bottom of the OSE stack of layers as shown in figure 7.1. This layer consists of highly hardware-specific
functionality such as being able to create, store and restore a process context, interrupt vectors and traps. Much of this layer is written in C and Assembler.

![Figure 7.1: The HAL layer of OSE](image)

The author of [9] based much of the code in the HAL from a previous implementation of OSE on a Microblaze processor. The work done on the HAL layer has therefore mainly consisted of translating Microblaze Assembler to Tilera assembler and modifying existing code to adapt to the new hardware conditions in the Tilera chip. This is not a trivial task, considering there is little to no documentation on the routines and the fact that some Microblaze instructions were completely undocumented.

Several files in the HAL were worked on, but most of the interrupt-related code was located in a file called `ti_s.S`. This is an Assembler file that contains low-level functions such as context creation, handling, saving, restoring and, more importantly, external interrupt handling. This routine, called `zzexternal_interrupt`, will be covered more thoroughly in the next section section (7.3.3).

### 7.3.3 `zzexternal_interrupts`

This routine is an entry point for the kernel when dealing with an external interrupt. Translating this routine from Microblaze to Tilera Assembler was a considerable task that consumed much time during the implementation-phase. It is responsible for OSE:s handling of external interrupts and is shown in figure [7.2]. The functionality is as follows:

- Enter function `zzexternal_interrupts` (interrupts are disabled at this stage)
- Push previously used registers onto stack

---

2Microblaze is a CPU developed by Xilinx (www.xilinx.com).
Setup arguments to \texttt{zzinit\_int}
\begin{verbatim}
add r5, zero, r43  # pcb
add r6, zero, r31  # wakeup
add r7, zero, r44  # int\_level
add r8, zero, r35  # oldint\_level
\end{verbatim}
This is a call to \texttt{zzinit\_int}
\begin{verbatim}
moveli r45, lo16(zzinit\_int)
auli r45, r45, ha16(zzinit\_int)
jalr r45
bz r0, return\_from\_int  # If r3 = 0
addi r0, r0, -2
bz r0, do\_wakeup  # If r0 = 2
\end{verbatim}
\texttt{switch\_process}:

Table 7.2: Section of assembler code in \texttt{zzexternal\_interrupt} where \texttt{zzinit\_int} is called and returned

- Call a vector handler that returns the vector number of the interrupts which caused the interrupt to begin with.

- Call a mask handler that will mask any lower prioritized interrupts and enable all higher interrupts.

- Call the OSE function \texttt{zzinit\_int} with the vector number. This will locate and execute the code belonging to the current interrupt. This part of the code is shown in table 7.2

The function \texttt{zzinit\_int} will return with one of three values: \texttt{return\_from\_int}, \texttt{do\_wakeup} or \texttt{switch\_process}. This depends on events that occurred in the interrupt routine. Some events may lead to re-scheduling being needed or another interrupt needing to execute. If a \texttt{return\_from\_int} was returned:

- Call the mask handler which will adjust interrupts to the same state as before the interrupt occurred.

- Restore all registers from stack.

- Continue executing the process that was interrupted to begin with.

- Interrupt routine completed.

If, instead, a \texttt{do\_wakeup} was returned by \texttt{zzinit\_int}, a new interrupt routine needs to be called and the following will happen:

- Call mask handler and adjust interrupts according to the new levels needed by the new interrupt.

- Call \texttt{zzinit\_int} with variables pointing to the new interrupt that needs to execute.
Lastly, if `zzinit_int` returns with a `switch_process` value, an event transpired that causes a different process to be executed (with higher priority), rather than the one that was interrupted. This is achieved by carrying out the following steps:

- Call mask handler which will adjust interrupts to the new levels needed by the new process.
- Move the registers and context from the previously interrupted process to a user-accessible area so execution can continue at a later stage.
- Call the OSE-function `zzkm_quit_int` which will search for the new process to be executed.
- Execution is resumed on the new process.

This function has been finished, but it could not be properly tested without executing on a system capable of generating external interrupts for it to use. Therefore, it was decided at this phase together with [8] to work together towards achieving functioning interrupts.

Figure 7.2: Functionality of the HAL-function `zzexternal_interrupt`
7.3.4 Interrupts

Work on enabling interrupts on the system was initiated together with the author of [8] in a joint effort to proceed with both thesis projects. This task consisted of being able to successfully map 128MB of memory as well as mapping the interrupt addresses used by the Tilera chip to an address space reachable though the simulator. This task unfortunately proved to be significant and consumed much time and energy. Tilera manuals and specification were scrutinized for assistance, employees at Enea tried to provide as much assistance as possible but to no avail. In the end the interrupt addresses where not successfully re-mapped with TLB entries. Remaining addresses were however successfully mapped with TLB entries.

7.4 Conclusions

This part of this thesis project focused on porting as much as possible, within the time frame of the thesis project, of the Hardware Abstraction Layer from a previous Microblaze implementation to the TILEPro64 architecture. The HAL in its entirety was unfortunately not completed, but this thesis project made many contributions to low-level assembler functions dealing with OSE’s handling of external interrupts. Work on porting these functionalities consumed a considerable amount of time due to sparse documentation at best and the sheer complexity of the functions.

Beyond these contributions, much effort was put into trying to enable interrupts in the simulator used for the porting process. These efforts were, however, hampered by a lack of time and documentation.
Chapter 8
Conclusions and Future Work

8.1 Conclusions

The theoretical studies performed in this thesis project consisted of investigating what memory management techniques are available today and what progress is being achieved in the research fields. Chapter 5 goes through this at great length and demonstrates that it is indeed a field in which much research is being done as well as the fact that much work currently derives in one way or another from GPUs. This is a natural move, since multicore- and manycore- based GPUs are more common than single or few-core GPUs. Furthermore, it was concluded that memory management techniques for manycore systems copes with RTOS without any problems, as long as they produce quick and correct computations in a deterministic and safe way. Moving on to the question of how OSE’s message-passing features cope with a manycore platform, this proved not to be an issue as long as the hardware is dependable enough. Enea AB has, as previously mentioned, no formal specification or demands on the hardware in regard to the message-passing technologies. In chapter 5 Cache Bins was proposed for implementation in OSE to run on the TILEPro64 chip. Section 7.3.1 discussed how this could be implemented practically.

Furthermore, contributions were made to the ongoing project of porting OSE to the TILEPro64 chip. These contributions surrounded the HAL layer and some large and key functions for interrupts. Progress was, however, halted by the fact that interrupts were never successfully activated in the simulator.

8.2 Future work

The future suggestions laid out for the continuation of the porting project are many, but the first few steps are clear:

**Complete work on the HAL and BSP** Before any experimentation on multicore or even manycore versions of OSE, a foundation on which OSE can function properly on single-core is needed. This means completing all hardware-dependent code.

**Complete work on device drivers** This is necessary for OSE to be able to communicate with the outside world, as well as having a properly functioning scheduler.

**Complete work on C-runtime** Contains some architecture-dependent functionality such as system calls and memory manipulation.
Conclusions and Future Work

**Complete work on MMU**  Not a necessary step initially, since OSE can run without it. It is however not a trivial task. In this step, it is also possible to spend time on optimizing the MMU for the TILEPro64 chip.

**Experiment with multicore bootstrap**  This is the step in which experimenting on booting OSE to multiple cores can start.

**Inter Process Interrupts**  This is required to implement the kernel event backplane used in the multicore version of OSE.

**Cache Bins**  As discussed in chapter 6, Cache Bins is the one technique proposed for implementation in OSE on the TILEPro64 chip. It may need some tweaking and experimenting before any optimal configuration can be reached, but experimental results are as shown in [7] promising.

### 8.3 Discussion and Final Thoughts

The pre-study that was performed over the initial 10 weeks really opened my eyes to the future of processors and multicore software. The number of articles I came across were as staggering as the number of subjects and points possible to include in the thesis. I was forced to exclude much interesting information simply because it had not been possible to cover everything within the scope of this thesis.

Throughout the porting process of this thesis, much has been learned about the complexity of operating systems. The sheer mass of code and information that needs to be understood is not to be taken lightly. It has been obvious that much research needs to be done on every step of the way. Research though manuals, documentation and most importantly, by contacting key members of the OSE development team who undoubtedly have important experience and knowledge of the core components.

On the personal side, I have learned much about the complexity of operating systems in general, as well as practical skills with tools such as Clearcase, makefiles and Assembler programming.
Bibliography


[10] “Sonics inc., sonics memmax scheduler,”
    http://www.sonicsinc.com/uploads/pdfs/memmaxscheduler_DS_021610.pdf,
    Accessed: 03/02/2012.


