Adapting Operating Systems to Embedded Manycores: Scheduling and Inter-Process Communication

Vishnuvardhan Avula
Abstract

Adapting Operating Systems to Embedded Manycores: Scheduling and Inter-Process Communication

Vishnuvardhan Avula

The aim of this thesis is to adapt a bare minimum version of Enea’s OSE, a real-time operating system to Tilepro64, a 32-bit many-core processor. The tasks include mapping memory regions, context handling and interrupt management and building drivers for programmable interrupt controller and timer. Time-based scheduling on the Tilera port of OSE has been achieved. Furthermore, design proposals for improving inter-core data throughput in LINX for Linux have been suggested. The suggestions include using smart pointers and shared heaps as part of the user space thereby achieving a zero-copy mechanism. With reference to TILPro64, consideration for leveraging the sharing of smart pointers using User Dynamic Network – a user accessible NoC – has been suggested in the adaptation design. Thus an inter-core process communication design involving minimal kernel involvement and reduced usage system calls was proposed.

This thesis is part of Portable and Predictable Performance (PAPP) on Heterogeneous Embedded Manycores - an active research initiative undertaken by Advanced Research and Technology for Embedded Intelligent Systems (ARTEMIS). It is also partly associated with the Many-core programming and resource management for high-performance Embedded Systems (MANY) project and funded by Information Technology for European Advancement (ITEA2). The thesis work is carried out at Enea Software AB and XDIN AB.
Acknowledgements

My deepest thanks to Patrik Strömblad and Sixten Thames Sjöström, for their technical guidance during my thesis. Discussions with Patrik were crucial and his directions gave me clarity in formulating objectives and proposing the design for inter-core communication. Thanks to Sixten’s knowledge transfer that I was able to jump-start with my work. His patience and willingness to take time off to help me with troubleshooting deserves a special mention. I am also grateful to Lennart Johansson, Iqram Ali and many others at Enea Software AB for showing interest in my work. I sought their support in a sizeable way while understanding an OSE system.

I am much indebted to Detlef Scholle and Barbro Claesson, my supervisors from XDIN AB for giving me this excellent thesis opportunity. Meetings with Detlef were highly motivational and he was instrumental in shaping up my thesis, presentation and the report.

My earnest thanks to Xiayoue Pan, my supervisor from University for her readiness to guide me during the study phase and with the report and presentations. She was welcoming for technical discussions that helped me comprehend some of the design fundamentals.

Further, I would like to thank my reviewer Dr. Philipp Rümmer, Programme Director for Masters Programme in Embedded System, Uppsala University for undertaking my Masters thesis and ensuring quality in my report.

Finally, cheers to friends and family for having been encouraging and supportive throughout my thesis.
# Contents

1 Introduction 1
  1.1 Background ................................. 1
  1.2 Problem Statement ......................... 3
    1.2.1 Porting RTOS to Many-Core ............ 3
    1.2.2 Achieving High Throughput Inter-Core Process Communication in LINX ............ 4
  1.3 Method ........................................ 4
  1.4 Limitations ................................. 4
  1.5 Outline of Results ......................... 5
  1.6 Thesis Organization ....................... 5

2 Background Study 6
  2.1 Classification Based on Parallelism .......... 6
    2.1.1 Instruction Level Parallelism .......... 6
    2.1.2 Data Level Parallelism ................. 6
    2.1.3 Thread Level Parallelism ............... 7
  2.2 Classification Based on Core ............... 7
    2.2.1 Homogeneous Multi-Core ............... 7
    2.2.2 Heterogeneous Multi-Core ............. 8
  2.3 Classification Based on Memory Architecture .. 8
    2.3.1 Shared memory .......................... 8
    2.3.2 Distributed Memory ...................... 9
  2.4 Classification Based on Software Model ...... 10
    2.4.1 Asymmetric Multi-Processing .......... 10
    2.4.2 Symmetric Multi-Processing ............ 10

3 TILEPro64 12
  3.1 Architecture Overview ..................... 12
3.2 Tiles ........................................ 12
  3.2.1 Processor Engine ....................... 12
  3.2.2 Cache .................................. 14
  3.2.3 Switch Engine .......................... 15
3.3 Register Usage .............................. 15
  3.3.1 Stack .................................. 17
3.4 Bare Metal Environment .................. 17
3.5 Protection Levels .......................... 17
3.6 Interrupts .................................. 18
3.7 Tile Timer .................................. 19

4 Enea OSE .................................... 21
  4.1 Architecture Overview .................... 21
    4.1.1 Processes ................................ 22
    4.1.2 Blocks .................................. 26
    4.1.3 Pools .................................. 26
    4.1.4 Signals ................................ 27
  4.2 Core Components ........................... 27
    4.2.1 Board Support Package ................... 27
  4.3 Interrupt Handling ........................ 28

5 Multi-Core Communication .................... 31
  5.1 Factors influencing ICPC ................. 32
  5.2 LINX for Linux ............................. 33
    5.2.1 LINX Architecture ....................... 33
    5.2.2 LINX Usage .............................. 35
    5.2.3 LINX for High Throughput Intra-Core Communication 37
    5.2.4 ICPC Design Specification ............... 37
  5.3 Design Case Studies ........................ 38
    5.3.1 Design I: Lock-free Channels ............ 38
    5.3.2 Design II: Zero Copy Inter-Core Message Passing 42
    5.3.3 Design III: Zero Copy Shared Heap .......... 48
    5.3.4 Summary of the Case Study Designs ........ 56

6 Implementation Details ....................... 59
  6.1 Adapting Enea OSE to TILEPro64 .......... 59
    6.1.1 Development Environment ................. 59
    6.1.2 Milestones Reached ..................... 60
6.2 Design Proposal for a High-Throughput ICPC in LINX  

7 Conclusions and Future Work  
7.1 Conclusions  
7.2 Future Work  

A Demonstration  
A.1 Application Code  
A.2 Application Output  

B Test Cases for LINX ICPC Design  
B.1 Test Case 1  
B.1.1 Use Case  
B.1.2 Desired Outcome  
B.2 Test Case 2  
B.2.1 Use Case  
B.2.2 Desired Outcome  
B.3 Test Case 3  
B.3.1 Use Case  
B.3.2 Desired Outcome  
B.4 Test Case 4  
B.4.1 Use Case  
B.4.2 Desired Outcome
## List of Figures

3.1 Internal layout of TILEPro64 .......................... 13  
3.2 Internal view showing processor, cache and switch engines  14  
4.1 Multi-layered architecture of Enea’s OSE ...................... 22  
4.2 The OSE Kernel along with other core components ............. 28  
5.1 LINX Architecture ........................................ 34  
5.2 Lock-free channels for sending/receiving small/large messages 39  
5.3 Zero-copy Inter-core Message Passing ........................ 43  
5.4 Micro-benchmark comparing throughput between various IPC mechanisms between a sender and a single receiver ........... 46  
5.5 Micro-benchmark comparing throughput between various IPC mechanisms between a sender and 23 receivers .................. 47  
5.6 UML diagram of HeapItem .............................. 49  
5.7 Layout of shared heaps ................................ 50  
5.8 UML diagram of BPtr ...................................... 51  
5.9 Shared heap based zero-copy operations ...................... 52  
5.10 The runtime of SimpleSim with 8 logical processes and variable messages sizes ................................................. 54  
5.11 The runtime of SimpleSim with a message size of 7000 bytes, varying the number of logical processes ......................... 55  
6.1 Interrupts vectors relocated via statically mapped TLB entries  61  
6.2 Software workaround for implementing interrupt status register 61  
6.3 Demonstration of the implementation .......................... 63  
6.4 A design proposal to adapt a shared heap design to LINX for high-performance ICPC ........................................... 64
List of Tables

3.1 Software accessible registers on a tile . . . . . . . . . . . . 16
5.1 Summary of the case study designs . . . . . . . . . . . . . . 57
Chapter 1

Introduction

1.1 Background

For many decades, Moore’s law had successfully overseen the trend that led to the development of modern computers. Rapid improvements in transistor technology were powered by the industry’s desire to build faster processors capable of running at increasingly high clock rates. Improvements in computational throughput came at the cost of increased design complexity and power consumption. The cost-performance curve began to flatten consequently. The industry thereby realized that merely improving clock rates and engineering complex circuitry does not scale up with the ever increasing demand for faster applications within a reduced power budget. At this stage, studies showed exploiting parallelism at task level reaped more benefits in terms of cost-performance. It required real multi-threading and hence the need for multiple CPUs\(^1\). It was thus time for replicating CPU cores on a single die which thereby saw the advent of multi-cores around the mid of the last decade. However, such parallel tasks were executed on cores optimized for serial performance. Building energy efficient processors on the other hand, requires simpler cores. Replicating many such low-complexity cores to the order of 10 to 100-fold times yields many-cores. Many-cores are both energy efficient and computationally effective. TILEPro64 is one such platform targeting applications demanding high computation.

However, single-threaded sequential programs do not execute any faster on

\(^1\)Central Processing Unit
multi-cores as they do not exploit the concurrency features of the underlying hardware. Applications need to be parallelized so as to utilize the available cores concurrently. Developing multi-threaded programs for many-cores and multi-cores is quite challenging. The programming and debugging techniques for concurrent algorithms are different from their single-threaded counterparts. A parallel programmer should consider synchronization, inter-core communication and data consistency aspects while developing a parallel algorithm. Programmers are required to be wary of the bugs related to data races and deadlocks that they might otherwise unintentionally introduce into the parallel program. In addition to a scalable design, one has to also consider the load distribution of the algorithm on the cores and the communication patterns between cores as they greatly influence the performance and power requirements of multi-core system. Thus the challenges faced while developing a parallel algorithm are many times more complicated than a sequential algorithm. In other words, it is a daunting task to parallelize legacy applications. Thus mainstream adoption of parallel programming style by costumers requires great support for concurrent programming solutions from hardware and software vendors alike, such as in the form of libraries, APIs \textsuperscript{2}, concurrent languages, distributed operating systems, performance and analysis tools, etc.

The project MANY, is commissioned with the objective to help the industry transition to many-core platform for developing embedded solutions. The company Enea AB, which has developed Enea OSE\textsuperscript{3} - a proprietary RTOS\textsuperscript{4} targeting embedded systems - is part of this research consortium, working towards developing tools facilitating rapid development of scalable software on multi-core platform. XDIN AB is the company which is involved in the directive to adapt Enea OSE to TILEPro64 platform. Thus the main objective of this thesis will be to adapt a minimal OSE image to TILEPro64 and set the system up and running. Eventually, this thesis contributes to extending the features of TILEPro port of OSE thereby providing a platform for further development work in the direction of achieving full-fledged real time support.

\textsuperscript{2}Application Programming Interfaces
\textsuperscript{3}Enea OSE: Multi-core Real-Time Operating System
\textsuperscript{4}Real Time Operating System
1.2 Problem Statement

1.2.1 Porting RTOS to Many-Core

A prior thesis work [25] has laid the foundation for adapting Enea OSE 5.5 to TILEPro64. This thesis aims at implementing the required functionalities in the BSP\textsuperscript{5} and HAL\textsuperscript{6} and thereby achieve a runnable many-core system capable of evaluating scheduling algorithms. Implementations will mostly be done in the layer below the OS and hence involves developing assembly codes specific to TILEPro64 processor. MicroBlaze is chosen as the reference architecture.

Support for Time Driven Scheduling

At present Enea OSE 5.5 only supports event driven scheduling of tasks on TILEPro64. However, in order to support time driven scheduling it is necessary to handle interrupts generated by the hardware timer. Moreover, a timer driver has to be implemented for providing the system tick needed for the scheduler and other time based functionalities.

Implementing UART for Communication via Console

At present the Tilera port of OSE only supports writing to Ramlog. A functional UART\textsuperscript{7} driver will be required in order to establish a two-way communication between the OSE shell and the external world.

Interrupts and Context Handling

Furthermore, the functions for entering into interrupt handlers and for ensuring safe context handling during interrupts are necessary to be implemented. It also requires mapping the virtual address space to the physical interrupt vectors in the TLB\textsuperscript{8}.

\begin{footnotesize}
\textsuperscript{5}Board Support Package
\textsuperscript{6}Hardware Abstraction Layer
\textsuperscript{7}Universal Asynchronous Receiver/Transmitter
\textsuperscript{8}Translation Look-aside Buffer
\end{footnotesize}
1.2.2 Achieving High Throughput Inter-Core Process Communication in LINX

The final part of the thesis involves designing an efficient inter-core process communication mechanism. Case studies on some existing designs shall be performed and the one that best fits the design criteria will be chosen for adaptation into high throughput inter-core communication using LINX on Linux.

1.3 Method

This thesis is divided into three phases. Phase 1 will last for 3 weeks during which a background study into the architectures of Enea OSE and TILEPro64 will be made for a thorough understanding.

Phase 2 is the implementation phase that will run over fifteen weeks. This phase covers the task of porting OSE to many-core platform. It also involves active discussions with the employees at Enea AB in order to accomplish the task.

Phase 3 marks the mid-thesis and involves an academic study on LINX for Linux and comparing the performances of some interesting inter-core communication protocols. An appropriate design satisfying the criteria laid for efficient inter-core process communication will be achieved for adaptation into LINX for Linux.

1.4 Limitations

The scope of this thesis includes literature study, design, implementation, report and presentations. The implementation will be done using Tilera board simulator and the version of OSE to be ported is OSE 5.5. The inter-core communication protocols studied will be limited to architectures based on shared memory model.
1.5 Outline of Results

As an outcome of this thesis, a working prototype of TILEPro port of OSE was delivered to Enea AB. The developed system is capable of handling timer interrupts and schedule user processes. Further, a design for high-throughput ICPC in LINX for Linux in many-cores was proposed after studying and comparing some suitable IPC designs.

1.6 Thesis Organization

Chapter 2 is a primer explaining some frequently used computer architecture terminologies, which are of relevance to this thesis. Chapter 3 briefly describes TILEPro64 architecture. Chapter 4 is centred around Enea OSE and mainly presents the software components used in the thesis. The background of LINX and the case studies for interprocess communication designs are presented in chapter 5. The implementation details of adapting TILEPro port of OSE is described in chapter 6. This chapter also includes a design proposal for improved inter-core process communication throughput in LINX for Linux. Chapter 7 concludes this report with the thesis conclusions and scope for further improvement. Appendix A includes the application code and output achieved during the scope of this thesis. The test cases used to validate the proposal design for LINX are provided as Appendix B.
Chapter 2

Background Study

This chapter serves a precursor to the specifics of Enea OSE and TILEPro64.

2.1 Classification Based on Parallelism

2.1.1 Instruction Level Parallelism

Instruction level parallelism is the simultaneous execution of two or more independent instructions by a processor. Achieving instruction level parallelism requires the support of either the processor or the compiler. Some examples are superscalars, VLIW\(^1\), out-of-order execution and speculative execution [19]. Although instruction level parallelism based processors can speed-up the execution significantly, their performance is limited by the level of parallelism that the hardware or compiler can discover in the window of instructions to be executed. Since it is not always possible to find independent instructions within a limited window, some of the additional execution units spend the cycles idling. As a result, the significance of incremental performance gained by adding 4 or more execution units is not as noticeable as the performance gain achieved by increasing the execution units to 2.

2.1.2 Data Level Parallelism

Data level parallelism is achieved when multiple units of independent data are simultaneously operated upon by the set of instructions. Multi-processors

\(^1\)Very Large Instruction Word
that execute SIMD\textsuperscript{2} [19] instructions exploit this property in order to achieve a high throughput. It is also an execution mechanism found in vector architectures. One instance of data parallelism as exploited by the compilers is the loop unfolding technique. User programs, for instance addition of a large matrix, can have access to data level parallelism by means of threads, where each thread operates simultaneously upon a subset of a larger set of data. Data level parallelism is commonly used in multimedia applications such as for changing the brightness of a picture, where a common value is simultaneously added or removed from the R-G-B\textsuperscript{3} data streams of the pixels. However, data parallelism is not applicable to algorithms which are not vectorized.

\subsection*{2.1.3 Thread Level Parallelism}

Multi-cores are said to have exploited thread level parallelism when each core operates on its own instruction and data. MIMD\textsuperscript{4} [19] systems are based on this type of parallelism. Multiple threads can be executed simultaneously. Thus a program benefits if its parallel parts are identified and assigned to threads executing on multiple cores. They are widely used in CMPs\textsuperscript{5}. The cores on a CMP are tightly coupled allowing better inter-core communication than in SMPs. They work at a lower operating frequency and consume lesser power than SMPs. Access to resources shared among threads must be serialized. Thread level parallelism is used in present day computers and increasingly in smart phones.

\section*{2.2 Classification Based on Core}

\subsection*{2.2.1 Homogeneous Multi-Core}

Homogeneous multi-core is a type of multi-core processor that has multiple replicas of a single core [6] over its die area. They operate at the same frequency and possess identical cache sizes and functionality. Since the cores are comprised of identical hardware and they operate on the same instruction

\footnotesize\textsuperscript{2}Single Instruction Multiple Data

\footnotesize\textsuperscript{3}Red, Green and Blue comprise the primary colours

\footnotesize\textsuperscript{4}Multiple Instruction Multiple Data

\footnotesize\textsuperscript{5}Chip Multiprocessor
set, homogeneous multi-cores are best known for ease of programmability. They are most suitable for general purpose applications and thus hog a large market share in the desktop computers segment.

2.2.2 Heterogeneous Multi-Core

Heterogeneous multi-core processor houses a number of cores with varying dimensions on the same die. Besides, their operating frequencies may differ from each other and each of them has its own power needs. Each core’s functionalities are optimized for a specific task and it operates on its own instruction set. Developing applications on such multi-cores is not as easy as could be done on their homogeneous counterparts. However, their power-to-computational efficiency far outweighs their complexity. Such multi-cores find their typical application in multimedia and networking applications where a general purpose processor provides takes care of handling user interaction and overall system services while a special signal processing core handles the more computationally intensive signal processing algorithms.

2.3 Classification Based on Memory Architecture

2.3.1 Shared memory

A shared memory system comprises multiple cores sharing a global physical memory space. Inter-core communication via shared memory is typically fast. However, they do not scale well in a distributed system environment as memory-access bottlenecks are introduced. They can further be classified into UMA\textsuperscript{6} and NUMA\textsuperscript{7} \cite{19} depending upon how the memory is arranged in the multi-core setting. When the physical memory is uniformly arranged amongst the cores it is said to be a UMA architecture. All cores have uniform access to the memory regardless of their physical location. On the other hand, a NUMA architecture comprises cores coupled with local memory. Although shared, the memory access to a core’s local memory is much faster than accesses to other core’s memory. Both UMA and NUMA architectures can

\textsuperscript{6}Uniform Memory Access
\textsuperscript{7}Non-Uniform Memory Access
involve caches, which complicate the memory architecture by introducing cache coherency protocols.

**Cache Coherency Protocols**

When caches are local to individual cores, updates made by one core are not apparent to another. This leads to cache incoherency if the variable, which resides in two or more caches, is not updated. A situation where a variable is resident in multiple caches leads to cache incoherency. Cache coherency protocols have been designed to deal with inconsistency in the value of variables. The two types of cache coherency protocols are directory-based and snooping protocols. These protocols primarily keep track of the meta-information such as the sharing status of a cached memory block.

*Directory Based Protocols*

The sharing status is stored in a centralized location called the directory and all cache operations occur via tandem references to this directory.

*Snooping Protocols*

The protocol is snooping based if the sharing information is copied along with a memory copy of a block from the physical memory into the cache. In such a protocol, all cache controllers constantly monitor the bus for any cache activity and execute actions on their caches accordingly.

### 2.3.2 Distributed Memory

Distributed memory is an architecture where memory is distributed among cores. Each core has its own private memory which is not visible in the global memory space. All local accesses to memory avoid access to the chip-interconnect bus and thus bandwidth is conserved. They scale well with the count of processors and are therefore suitable for use in parallel computers and distributed systems. Inter-processor communication is typically achieved by means of message passing. As a result of extensive send-receive style of communication taking place between processors requesting a data and the remote processor owning the data, remote memory accesses incur a high bandwidth overhead.
2.4 Classification Based on Software Model

2.4.1 Asymmetric Multi-Processing

When every core in the multi-core platform runs its own operating system, complete to scheduler, it is said to be following an AMP\(^{8}\) model. The heterogeneous multi-core architecture makes the most out of this model. AMP is suitable for use in a distributed system. A programmer can define execution boundaries and assign specific cores for those defined tasks. However, these systems do not feature a load balancing support. Consequently, load balancing needs to be taken care of by the programmer.

2.4.2 Symmetric Multi-Processing

When a single OS image is shared amongst multiple cores, the model is called SMP\(^{9}\). Such a system shares a number of resources between cores. Hence, synchronization, which becomes vital, is achieved by locking mechanisms. The OS provides a nice abstraction of the hardware making application programming easier. Load balancing is usually performed by the OS. However, the abstraction introduces significant overhead, rendering the OS less suitable for real-time applications involving intensive I/O\(^{10}\) operations.

---

\(^{8}\) Asymmetric Multi-Processing
\(^{9}\) Symmetric Multi-Processing
\(^{10}\) Input/Output
This chapter introduces the reader to TILEPro64, the multi-core processor to which Enea’s OSE will be ported. References here are mostly derived from Tilera manuals whose access is restricted outside Enea [8] [11].

3.1 Architecture Overview

TILEPro64 is a 64-core processor inspired by MIT’s RAW architecture. Tiles, as Tilera refers to the cores, are designed homogeneous and are placed in an 8x8 array fashion on the chip. Each of these tiles is a full-fledged processor capable of running an entire OS independent of the others. Each independent tile comprises a 32-bit integer processing engine, memory management unit, program counter, L1/L2 cache, and many more. The interconnection among tiles and with I/O and memory is achieved through Tilera’s iMesh technology.

3.2 Tiles

Each tile, which is a full-fledged processor, is comprised of the following components: Processor engine, Cache and Switch engine.

3.2.1 Processor Engine

As mentioned earlier, the processor engines are 32-bit processors. The instructions are VLIW and they encode 2 or 3 instructions. The instruction
set also contains instructions that are often used in DSP\textsuperscript{1}, network packet processing, etc. Processor engines comprise 64 registers that are 32 bits wide. Three execution pipelines (P0, P1 and P2) of five stages each have been implemented on these processor engines. P0 is used for performing arithmetic and logical operations, bit and byte manipulations, and all multiply and fused multiply instructions \cite{11}. P1 performs SPR\textsuperscript{2} reads and writes and conditional transfers in addition to arithmetic and logical operations. P2 processes the memory operations viz., load, store and test-and-set instructions.

\textsuperscript{1}Digital Signal Processors
\textsuperscript{2}Special Purpose Register
3.2.2 Cache

They comprise TLBs, caches and cache sequencers. The TLBs are separate for instruction and data. The cache houses a 16KB L1 Instruction cache, an 8KB L1 Data cache and a 64KB combined L2 cache. In addition, the TILE architecture combines all the L2 caches with the help of iMesh network to form an L3 cache. Such a cache is distributed among the tiles and is called DDC$^3$.

$^3$Dynamic Distributed Cache
### 3.2.3 Switch Engine

The six independent networks that make up the iMesh are implemented in the switch engine. These networks comprise the STN\(^4\) and five dynamic networks viz., UDN\(^5\), TDN\(^6\), MDN\(^7\), CDN\(^8\) and IDN\(^9\). The engines have implemented buffering and flow control into each of these networks, thereby enable asynchronous processing of data by the tiles.

### 3.3 Register Usage

Each tile consists of 64 32 bit wide software-accessible registers. Table 3.1 lists the available software accessible registers along with their calling convention and usage. Based on the calling conventions that are used when a program executes function calls on the Tile platform, these registers are classified as caller-saved, callee-saved, dedicated and network. Caller-saved registers can be used in many ways such as for scratch-pad operations and parameter passing. Since their values may change after returning from a function, the caller should save these registers if their values need to be preserved. On the other hand, callee-saved registers are required to have the same value when control exit from a function as when it entered. Hence, such registers must be saved by the callee functions.

Dedicated registers are associated for specific data entities. ABI\(^{10}\) compliant software should use these registers as specified. Any deviation in their usage pattern could lead to inconsistent states in the system \([9]\). Network registers correspond to specific NoCs\(^{11}\) and are available for ingress and egress operations on the network data.

---

\(^4\)Static Network  
\(^5\)User Dynamic Network  
\(^6\)Tile Dynamic Network  
\(^7\)Memory Dynamic Network  
\(^8\)Coherence Dynamic Network  
\(^9\)I/O Dynamic Network  
\(^{10}\)Application Binary Interface  
\(^{11}\)Network on Chip
<table>
<thead>
<tr>
<th>Register</th>
<th>Type</th>
<th>Assembler Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 9</td>
<td>Caller-saved</td>
<td>r0 - r9</td>
<td>Parameter passing / return values</td>
</tr>
<tr>
<td>10 - 29</td>
<td>Caller-saved</td>
<td>r10 - r29</td>
<td></td>
</tr>
<tr>
<td>30 - 51</td>
<td>Callee-saved</td>
<td>r30 - r51</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>Callee-saved</td>
<td>r52</td>
<td>optional frame pointer</td>
</tr>
<tr>
<td>53</td>
<td>Dedicated</td>
<td>tp</td>
<td>Thread-local data</td>
</tr>
<tr>
<td>54</td>
<td>Dedicated</td>
<td>sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>55</td>
<td>Caller-saved</td>
<td>lr</td>
<td>Return address</td>
</tr>
<tr>
<td>56</td>
<td>Network</td>
<td>sn</td>
<td>Static network</td>
</tr>
<tr>
<td>57</td>
<td>Network</td>
<td>idn0</td>
<td>IO dynamic network 0</td>
</tr>
<tr>
<td>58</td>
<td>Network</td>
<td>idn1</td>
<td>IO dynamic network 1</td>
</tr>
<tr>
<td>59</td>
<td>Network</td>
<td>udn0</td>
<td>User dynamic network 0</td>
</tr>
<tr>
<td>60</td>
<td>Network</td>
<td>udn1</td>
<td>User dynamic network 1</td>
</tr>
<tr>
<td>61</td>
<td>Network</td>
<td>udn2</td>
<td>User dynamic network 2</td>
</tr>
<tr>
<td>62</td>
<td>Network</td>
<td>udn3</td>
<td>User dynamic network 3</td>
</tr>
<tr>
<td>63</td>
<td></td>
<td>zero</td>
<td>Always zero</td>
</tr>
</tbody>
</table>
3.3.1 Stack

The Tile processor does not contain a dedicated instruction to carry out stack operations. Instead, stack is managed entirely by software. The software makes use of the hardware-based stack pointer \( sp \), to operate the stack. The ABI requires that the stack starts from a high address and extends in the downward direction towards a low address. The stack pointer should always be aligned to a 64-bit boundary.

3.4 Bare Metal Environment

BME\textsuperscript{12} is a runtime environment that permits applications to gain direct access to hardware resources. It also provides runtime support for communication between tiles as well as with I/O devices. This environment avoids intervention by the supervisors or hypervisor.

3.5 Protection Levels

Tilera offers a simple yet fully-protected environment that allows for hierarchical privilege levels within an OS and coexistence of multiple OSes and user processes. This is achieved by implementing protection levels which are numbered 0 through 3 in the increasing order of privileges. The protection mechanism makes use of MPL\textsuperscript{13} and CPL\textsuperscript{14} registers. Each individual tile maintains a CPL register that holds the current privilege level of the tile. Additionally, the portions of the tile that require protection such as the timer, DMA, TLB, etc are associated with their own MPL register. The MPL dictates the minimum level of protection required to access a hardware resource without generating a GPV\textsuperscript{15} interrupt. When a protected action is initiated, the CPL is compared with the associated MPL. If the CPL is greater than or atleast equal to the MPL, the protected action executes successfully. However, if the CPL is found to be less than the MPL, a GPV is triggered.

A process executing with CPL at or above the MPL held by a particular

\textsuperscript{12}Bare Metal Environment
\textsuperscript{13}Minimum Protection Level
\textsuperscript{14}Current Protection Level
\textsuperscript{15}General Protection Violation
hardware’s MPL register can change the MPL value to an arbitrary lower or higher value up to the executing process’s CPL. However, if the process attempts to change the MPL to a value higher than its CPL will trigger GPV.

3.6 Interrupts

Tilera architecture calls all occurrences of exceptions as interrupts. There are 46 different types of interrupts that can occur on this system. They can further be grouped as maskable and non-maskable interrupts. Interrupts are serviced locally by the tile on which they are triggered. Should the interrupt be serviced on a remote tile, it becomes the responsibility of the OS to forward the interrupt to the concerned tile over the NoC. The interrupt system is governed by the same protection levels as mentioned in section 3.5. Thus, the MPL has a dual usage. The interrupts are vectored and each vectored location has the capacity to accommodate 256 bytes of interrupt code. There are 4 sets of vectored interrupts distributed across the available protection levels.

In the event of an interrupt, the MPL of the interrupt indicates the protection level at which that interrupt needs to be serviced. Certain other interrupts such as TLB miss and illegal instruction occur regardless of the protection level [10]. For such an interrupt, if the CPL is greater than or equal to the MPL of the interrupt, the interrupt is executed at the current protection level of the process. However, if the process’s CPL is lesser than the interrupt’s MPL, the interrupt is executed at the interrupt’s MPL.

The interrupt vector is calculated from the protection level, interrupt number and interrupt base address (0xFc000000) by

\[
\text{InterruptBaseAddress} + \text{ProtectionLevel} \times 16\text{MB} + \text{InterruptNumber} \times 256
\]

The PC\(^{16}\) and the protection level of the interrupted process along with the status of the INTERRUPT_CRITICAL_SECTION register are saved in a pair of SPRs accessible at the protection level of the interrupting process.

All maskable interrupts are associated with INTERRUPT_MASK_X[0, 1]

\(^{16}\text{Program Counter}\)
registers, where X represents the MPL. They are a pair of 32-bit SPRs where each bit independently masks or unmasks a particular interrupt. Clearing a bit corresponds to enabling the interrupt whereas setting it to 1 disabled the interrupt. Further, critically maskable interrupts can be masked when their MPL is less than or equal to the current protection level [10] and the INTERRUPT_CRITICAL_SECTION is set. Any queued interrupts that are previously masked are taken up for servicing as soon as their mask bit is cleared. An interrupt-specific action needs to be executed in order to clear queued interrupts.

### 3.7 Tile Timer

The tile timer is an in-tile system device. It triggers periodic interrupts that are used for generating system ticks and to perform other time related tasks. Ranked in the decreasing order of priority, the timer interrupt is positioned at 25 in the list of 48 interrupts. It is a critically maskable interrupt and can be queued.

The timer is, in principle, a perpetual down counter and is situated in the 31-bit COUNT field of the TILE_TIMER_CONTROL SPR. It triggers an interrupt with every underflow. The periodicity of interrupts can be varied by setting a suitable COUNT value. The timer notifies an underflow — wrap from 0 to \(2^{31}-1\), by flagging the UNDERFLOW bit of the TILE_TIMER_CONTROL SPR to high. The routine servicing the timer interrupt has to clear the UNDERFLOW bit manually in order to notify that the interrupt is serviced.

MPL_TILE_TIMER SPR indicates the MPL required to configure the tile timer and service its interrupt thereof. For this project, the MPL chosen for timer is 2 corresponding to BME. The MPL_TILE_TIMER_SET_2 SPR is set to 1 to accomplish this.
Chapter 4

Enea OSE

OSE, a product of Enea, is a distributed operating system built for supporting hard and soft real-time requirements. It is based on a microkernel architecture and asynchronous direct message passing is chosen as the primary means of inter-process communication. OSE is designed to support a wide range of applications involving time-driven and event driven codes, POSIX\textsuperscript{1} codes, clustered and single node codes, etc. By adopting a message based programming model, an application gains access to OS and application services alike, in a location-transparent manner in a distributed setting. In what would otherwise be omitted by a microkernel OS, Enea’s OSE offers feature-rich functionalities by structuring them as layered services and extensions. This chapter introduces the reader to some of the OSE components that are relevant for this thesis. References to this chapter are mostly derived from Enea’s white paper [23] and some limited-access Enea documents [3] [4] [7].

4.1 Architecture Overview

The OSE multi-core version is a hybrid AMP/SMP model. As mentioned earlier, OSE has adopted a layered architecture where every layer provides services to upper layers and uses the services of the lower layers. The kernel benefits from the simplicity and flexibility of an SMP system while retaining the scalability and performance of an AMP system. The services provided provided by the operating system

\textsuperscript{1}Portable Operating System Interface
4.1.1 Processes

Processes are the fundamental building structures in OSE. They can be seen as a means by which the system allocates CPU resources. OSE contains different types of processes and they all complement each other in order to achieve the overall system goals. They aid in prioritizing the jobs to meet real time requirements. These processes are conceptually equivalent to tasks and threads available in other OSes. Processes can be grouped and made to share memory pools and heaps.

Process States

Since the scope of this thesis is limited to bringing up OSE on a single core, this section describes only those states of an OSE process that are relevant to single core scheduling:

**Running**

There can at most be one process executing on a core at any instance. A process under execution is termed to be in the running state. The scheduler ensures that no ready process of a priority higher than the currently running
one waits. In other words, the scheduler swaps in a high priority process as soon as it has entered the ready state. When a running process has executed a blocking call, the scheduler pushes it into the waiting state.

Ready
A process that is queued in the ready state indicates it is ready for execution and is merely waiting for the high priority process to finish the run.

Waiting
When a process has executed a delay, a break point or a blocking call wherein it awaits additional resources such as semaphores, the process enters the waiting state. It does not compete for the CPU.

Scheduling Processes

The following scheduling phrases are applicable to OSE:

Fixed Priority
A priority based scheduler ensures that the highest priority process keeps running unless it is in the waiting state. The priority of every process remains fixed unless explicitly changed using a system call. Fixed priority is the predefined scheduling scheme available in OSE.

Preemptive
A preemptive scheduler can preempt a running process. All standard processes in OSE are preemptible by a higher priority process.

Periodic
A periodic scheduler schedules a process to run at a specific time instance in a cyclic manner.

Round Robin
A process in this scheduling scheme runs for a stipulated time slot before yielding control over the processor to the next waiting process. When all processes have finished a cycle of execution, the cycle starts over by scheduling the first process to run.
Process Types

OSE offers the following process types:

*Interrupt Process*
Interrupt processes are executed in response to hardware and software triggered interrupts. They should run to completion without being blocked. Hence, the usage of blocking system calls is not permitted in these processes. The priorities of these processes are configured during their creation. Only a higher priority interrupt process can interrupt an interrupt process that is already being executed. Interrupts that fall in the same or lower priority level are serviced only after the completion of the interrupt process that is being executed. Interrupt processes have a priority level that is higher than background and prioritized processes.

An interrupt process is associated with an interrupt vector [3] when it is created for a hardware interrupt. OSE uses the hardware specific vector handler and mask handler to service or mask the interrupt.

Additionally, OSE provides ISRs\(^2\) which are a light-weight and a faster solution for servicing interrupts. Unlike interrupt process, ISRs keep the interrupts disabled until their completion. This feature ensures that they do not have to switch process contexts or mask interrupt levels [3]. However, as a side effect, nested interrupts cannot be serviced by ISRs.

*Timer Interrupt Process*
These processes are periodically called into execution by the system timer process; system timer process by itself is an interrupt processes. The periodicities of these processes are configured during their creation. Their periods must be a multiple of system tick [3]. The kernel checks for the timer interrupt process that should run at every occurrence of a system tick. All timer interrupt processes inherit the priority of the system timer process and hence of the same priority. However, when several timer interrupt processes are in a ready state, they are executed in an undefined order.

*Prioritized Process*
This is the most frequently used process type on an OSE system. The pri-
orities of these processes are configured during their creation. They remain fixed unless explicitly re-prioritized by means of a system call.

The highest priority process amongst the prioritized processes is scheduled to run only when no other timer or interrupt process is available in the ready state. Prioritized processes are designed to run in infinite loops, with the provision to receive signals via corresponding blocking system calls that are placed in the beginning of the loop [3]. When several prioritized processes of same priority level compete to run, the process that was first to reach the ready state is scheduled to run and the order follows. Prioritized process can be configured to yield to another process with the same priority level, after a specified time duration.

*Background Process*
These processes are scheduled to run only when none of the processes described earlier are running. The duration of execution, called time-slice, is specified during their creation. The scheduler permits the background process to execute until its time-slice expires. These processes do not have any priority associated with them. They are scheduled in a Round Robin manner.

*Phantom Process*
Process of this type are not counted as real processes. They do not contain any program code and hence are never scheduled. They are used as proxies to real processes such as in a process hunt, as in the case of LINX.

**Process Priorities**

The valid priority levels in OSE are 0-31, where 0 is the highest priority and 31 is the lowest. The process priorities are grouped into three bands. Interrupt and timer interrupt processes are members of the top priority band. This is followed by prioritized processes which belong to the middle priority band. Background processes fall into the bottommost priority band and they hold equal priorities. The priorities differ in their purpose according to the process type.

*Interrupt Process Priorities*
The priorities of interrupt processes servicing the interrupts from hardware devices are mapped to hardware priority. Mismatched priorities lead to an
undefined system behaviour. Real-time scheduling methods can be used to set the priorities of interrupt processes.

**Timer Interrupt Process Priorities**
As mentioned earlier, all timer interrupt processes inherit the priority of system timer process and hence execute at the same priority level. They do not hold a per process priority level and hence are set to 0.

**Prioritized Process Priorities**
The priorities for these processes reflect the responsiveness. A higher prioritized priority is more responsive as it executes ahead of lower priority processes. Real-time scheduling methods can be used to set the priorities of interrupt processes.

**Background Process Priorities**
Priorities are not set for background processes. They are scheduled to run on time-slice basis, provided no other process types are scheduled for run.

### 4.1.2 Blocks
Similar processes can be grouped into a subsystem called block. A process has to belong to a block and as a design principle, a child process is usually assigned to the same block as its parent. Blocks can be handled similar to processes in the way they are started, stopped, killed and made to share the same memory pool.

### 4.1.3 Pools
Pools are used as a means to allocate dynamic memory and process stacks in a deterministic manner. Memory allocated this way is much quicker and efficient against fragmentation. Hence they are preferred way of memory allocation in interrupt processes. There always exists a global memory pool called the system pool in which the environment variables are stored.
4.1.4 Signals

There are a number of ways by which OSE processes communicate such as semaphores and mutexes. However, the preferred method of IPC\(^3\) is by means of signals. They follow a message passing method of communication. Every signal carries a message of a specific type and length. Only the pointer to a message is copied amongst the processors when the signal is sent locally. The remains the same even when the signal is sent across memory protection boundaries to remote locations.

4.2 Core Components

OSE is built as a scalable microkernel centred around Executive. It is bundled up with many optional and compulsory components. The kernel comprises Executive and Bios. Real time scheduling, efficient message passing, signals and error handling apart from interrupt management are some of the functionalities of Executive. Bios is an abstraction layer for the Executive and other components registered with it. It handles the system calls made by applications to the underlying Executive and other registered components in such a way that their locations are not exposed to the application.

The OSE’s MM\(^4\) handles all the memory management functions such as memory allocation for programs, cache management and memory protection. CPU Hardware Abstraction Layer abstracts variations in processors belonging to the same architecture. The services provided by this layer is used by the Executive and MM to perform CPU specific functions. Ramlog is a valuable component that is used for logging information. It stores the log in a circular buffer of a configured size, which is allocated on the RAM\(^5\). The log information is useful for later debugging and system analysis.

4.2.1 Board Support Package

This resides in the bottommost layer of the OSE architecture. It comprises device drivers and other low level codes for system start-up, which are pro-

\(^3\)Inter-Process Communication  
\(^4\)Memory Manager  
\(^5\)Random Access Memory
4.3 Interrupt Handling

During the initialization phase, OSE installs the interrupt vector handlers. As the device drivers get registered, their interrupt processes are mapped to unique OSE interrupt vector numbers. In a multi-core system, the interrupt vectors are bound to the core associated with the interrupt process.

When interrupts are enabled and an external interrupt gets triggered, the processor disables the interrupt and jumps to the interrupt handler. However, when the interrupts are disabled, the processor does not assert the interrupt request. When asserted, the vector handler returns the vector number that identifies the interrupt source. The kernel uses this number to scan through the vector number table to verify if a valid interrupt process or ISR is associated with it. If verified to be a valid interrupt number, the associated interrupt hooks are executed. The execution path taken thereafter depends on whether the interrupt is be serviced by an interrupt process or ISR.

*Servicing by an ISR*
In this case, one or more ISRs associated with the interrupt vector are executed with the interrupts still disabled. The kernel checks for the return values from each of the ISRs to determine if rescheduling is requested. When the ISRs are executed to completion, a new process is scheduled to run if rescheduling is necessary.

*Servicing by an Interrupt Process*

In this case, the kernel passes the priority associated with the interrupt process to a configured interrupt mask handler. The mask handler masks all interrupts with same or lower priority. Once the mask handler returns, the kernel re-enables the global interrupt. Thus, the system becomes available to process incoming interrupts of priority higher than the currently executing interrupt.

The control then enters the entry point of the interrupt process. On completion of the interrupt process, the mask handler re-enables the previously masked interrupts. Since the kernel passes the priority level as an argument, the mask handler does not have to keep track of unmasking previously masked interrupts.
Chapter 5

Multi-Core Communication

In the face of broadening the spectrum of applications supported by embedded systems and making them pervasive, the problems have become increasingly complex. The challenge lies in designing solutions that perform intensive computations with high data throughput within the power budget of the devices.

With traditional single-cores being less sustainable at coping with demands, the industry has turned towards multi-cores that redress the problem with chip level parallelism. To take advantage of their hardware threads, the algorithms developed thereof need to exploit true parallelism of the problem at hand. One approach considers reducing the problem to tasks and strategically spreading them out onto the cores to solve, thus achieving a higher degree of parallel execution. However, such tasks exhibit data dependence on each other. Moreover, due to the nature of dependency on shared system resources, tasks also require synchronization mechanisms to coordinate access. A fault-proof system has to maintain consistent states across the cores [16]. This requires transfer of data and metadata between cores and hence the need for inter-core communication. While the nodes in a distributed system also necessitate communication, the demand for efficient on-chip communication between cores is more pronounced in a multi-core platform as the volume of data transfer is orders of magnitude larger.
5.1 Factors influencing ICPC\textsuperscript{1}

A protocol developed for inter-process communication is ideally designed considering the reliability and integrity requirements apart from its ease of configuration and the type of underlying communication infrastructure. These in turn influence its overhead, speed and scalability [26].

Mechanisms such as MPI\textsuperscript{2} and RPC\textsuperscript{3} have evolved primarily targeting distributed systems. They employ reliable protocols such as TCP/IP in order to achieve robustness against the non-deterministic communication latencies and other network artefacts induced in loosely coupled systems. As a result, they possess a large memory footprint. Moreover, they make an extensive usage of kernel services which are only accessible via expensive system calls [20]. Even traditional mechanisms such as Unix domain sockets, FIFOs\textsuperscript{4}, message queues, mmap and shared memories — that have been optimized for inter-process communication within core — rely on kernel services and copy [16] data between kernel and user space multiple times. In addition to occupying system memory unnecessarily, redundant copying of large sized data consumes [20] valuable CPU time. In summary, all these mechanisms prove expensive for frequent communication.

A typical video decoding application such as H.264 running on an SoC\textsuperscript{5}, incurs a communication overhead accounting for around half [22] the execution time. It follows intuitively that a reduction in the ICPC cost significantly improves the performance of such applications. Unlike in distributed systems, the cores on a multi-core chip are networked closely as NoC by inherently reliable transport media and they share memory. Mechanisms meant for inter-core communication are therefore desired to be efficient to meet performance requirements [22]. A good design of ICPC considers mitigating communication latency by reducing system calls and achieve throughput by reducing unnecessary data copies. In addition, if the frequency of ICPC is known before hand [26] — as is the case in real time applications — polling strategies are preferred to interrupts in order to send/receive messages, con-

\textsuperscript{1}Inter-Core Process Communication
\textsuperscript{2}Message Passing Interface
\textsuperscript{3}Remote Procedure Call
\textsuperscript{4}First In First Out
\textsuperscript{5}System on Chip
sequently saving time over context switches. Pipeline based applications benefit furthermore out of an ICPC protocol that brings autonomy between sender and receiver cores.

The subsequent section introduces the reader to Enea’s LINX — a widely used IPC mechanism before presenting the problem statement highlighting the scope for improving intra-core communication in LINX; the motivation behind this literature study. All references to LINX are derived from [14] [15] [12]. Following this, case studies of some interesting designs have been made.

5.2 LINX for Linux

Enea’s LINX [12] is an IPC protocol that is designed to be platform and interconnect independent. LINX is supported by both OSE and Linux. However, the topic of interest in this thesis is improving performance of LINX for Linux. LINX abstracts the applications from the implementation details of the underlying hardware and network topology [12]. Applications can communicate transparently regardless of whether running on same CPU or on different nodes in a cluster. LINX is designed with the goal of achieving high performance for both inter-node and intra-node communication, regardless of whether the underlying interconnect is reliable or not. In addition, it possesses a small memory footprint making it suitable for a device of any size – large or small.

5.2.1 LINX Architecture

LINX is based on message passing mechanism. It comprises Linux kernel modules, a library called LINX lib that can be linked with applications and some command line tools for setting up inter-node links and to support statistics. One of the LINX kernel modules implements IPC mechanism and RLNH\textsuperscript{6} that establishes LINX connectivity, which spreads across multiple nodes using logical links. In order to support inter-node communication,

\textsuperscript{6}Rapid Link Network Handler
it uses the CM⁷ corresponding to the interconnect being used. LINX kernel module provides a standardized socket interface using dedicated protocol family [14] PF_LINX. LINX lib provides APIs⁸, which the application can use instead of directly accessing sockets.

LINX is a two-layered protocol stack: RLNH and CM. An illustration of the architecture is shown in figure 5.1.

---

⁷ Connection Manager
⁸ Application Programming Interface

Figure 5.1: *LINX Architecture [13]*
RLNH is equivalent to the session layer of the OSI model [12]. It carries out IPC tasks such as looking up endpoints using their associated names, link supervision and notification of termination of endpoint. It also provides functionality for creating, resolving and removing local representations of remote endpoints. It provides the service of translating the SPID from source to destination endpoint.

CM is equivalent to the transport layer of the OSI model [12]. It is responsible for an ordered transmission of messages over the interconnecting media. It provides flow control, retransmission of lost packets and peer supervision when the underlying interconnect is unreliable. On a reliable interconnecting infrastructure, CM has a simpler role. Thus, by providing connection and link supervision, LINX targets high-availability applications.

5.2.2 LINX Usage

LINX Endpoints

Communication using LINX requires creating endpoints. Every endpoint is assigned a name. However, names are not mandated to be unique. Associated with each endpoint is a binary identifier called SPID. A pair of communicating endpoints refer to each other by their corresponding SPIDs. The uniqueness of SPIDs is restricted to the node in which they are created. All endpoints are internally connected to their respective sockets. Any resource owned by an endpoint is freed only when the endpoint itself is terminated.

LINX Signals

Applications interact by sending and receiving messages between endpoints. Messages are referred to as signals in LINX terminology. Signals [14] are buffers comprising a 4-byte signal number. This is followed by optional data that need to be communicated. The signal number ranges between 0x10000000 and 0xFFFFFFFF and could be used by user space applications. Signals are declared by means of a structure. The various signals that an application will use are contained in a union. LINX APIs make use of signals by converting them to pointers to non-generic unions.

---

Signals are allocated and initialized before they can be dispatched. The signal buffer is owned by the endpoint that created it. This ownership is transferred to the receiving endpoint on reception. LINX signals are received in a queue maintained by the destination. The destination endpoint has the ability to decide when a LINX signal can be received and can also choose to receive signals selectively. The receiver can thus reorder the messages based on the signal number before consuming them.

The receiver can reply by reusing the sender buffer. It overwrites the signal number field with a new value which can either be a response data or an acknowledgement. LINX supports automatic endian conversion of signal numbers. However, this feature is not supported for the entire message.

**Hunting an Endpoint**

The SPID of the destination endpoint has to be obtained before any signal can be transmitted to it. Hunt is a mechanism by which this is determined. LINX sends a response signal to the hunting endpoint on behalf of the hunted endpoint when the peer is found. Querying the received hunt [15] response for the sender reveals the SPID of the hunted peer. LINX maintains the hunts internally in a pending state until they are resolved [14].

**Attaching to Supervise Endpoints**

In the event of a termination/unavailability of receiving endpoint, the signal is discarded. LINX provides attach [15] mechanism for endpoint supervision by notifying the endpoint when its connection with the remote endpoint ceases. Such a signal is sent by the LINX runtime although it appears to have originated from the terminated endpoint.

**Inter-Node Communication**

Connections between LINX endpoints can sprawl across nodes than being restricted to just one node. Moreover, they are not bound to communication within the same OS; a LINX endpoint in Enea OSE can communicate with another LINX endpoint residing in Linux [12]. It takes two communicating endpoints to establish a link. Links must be uniquely named within the node [14]. The entire path in terms of the link name is specified in order to reach the distant node.
A LINX cluster is a logical network established between a group of nodes interconnected by one of the underlying transport media supported by LINX.

**Virtual Endpoints**

Since the uniqueness of an SPID is limited to within a node, it is not practical to refer to remote endpoints by their original SPID. LINX communicates across nodes by creating local virtual endpoints — representatives of the remote counterparts. These virtual endpoints assume unique SPIIDs. LINX runtime ensures that the signals to virtual endpoint are forwarded to the destined remote endpoint. A virtual endpoint is created once the remote endpoint is determined. They follow the same address as that of the link addresses leading to the desired remote endpoint [14]. The virtual endpoint is valid as long as the remote endpoints that it represents, exists.

**5.2.3 LINX for High Throughput Intra-Core Communication**

LINX has achieved high-performance for off-chip data transfers i.e., inter-node communication. However, its implementation on Linux does not distinguish on-chip communication from off-chip communication. As mentioned earlier, LINX for Linux internally makes use of sockets for endpoint communications, both on- and off-chip. It copies data between user and kernel buffers many times and does multiple switches between user and kernel space. Although the resulting latency is not noticeable in inter-node communication, it becomes a performance bottleneck in multi-cores as applications have necessitated the need for frequent on-chip IPC. Therefore, designing LINX in order to use it as a high-throughput backplane for on-chip IPC is necessary.

**5.2.4 ICPC Design Specification**

The design for improved throughput in LINX for inter-core process communication is expected to be lightweight and satisfy the following criteria [23]:

**Req1** Be built in user space and reduces system call usage for on-chip IPC.

**Req2** Exhibits scalability across cores and message sizes.
**Req3** Supports mechanisms that are relatively free of locks for simultaneous communication between many pairs of processes.

**Req4** Introduces minimal overhead for communication. This is essential for achieving high data throughput and reduced latency.

## 5.3 Design Case Studies

This section concerns the study of some of the design cases focusing IPC performance improvement on a multi-core platform. The designs chosen are based on shared memory model and restricted to UMA, so that they remain within the interest of Tile architecture. Further, the designs selected here do not represent an exhaustive set of designs available, since only a smaller subset has been selected to serve the purpose of the study.

### 5.3.1 Design I: Lock-free Channels

The design as presented in [17] is a memory copy mechanism implemented in the user space, satisfying **Req1**. It claims to achieve better scalability and high performance for intra-node communication in MPI [18]. It allows lock-free access to communication infrastructure thereby achieving low latency for ICPC. Moreover, it chooses an efficient memory copy mechanism according to the message size, thereby reducing memory usage and improving scalability.

**Architecture**

The design in [17] assumes $P$ processes running on different cores of multicore processor. Each individual process maintains a *Send Buffer Pool (SBP)*. In addition, every process dedicates one *Send Queue (SQ)* and one *receive buffer (RB)* to each of the other $P-1$ remaining processes. The explanation follows the notations $x$ and $y$ to represent the processes. $SQ_{xy}$ denotes the Send Queue maintained by process $y$ and is meant to hold the data for process $x$ to consume. Receiver Buffer $RB_{xy}$ is a shared buffer which the process $y$ maintains to store any data originating from process $x$. Similarly, the Send Buffer Pool owned by process $x$ is denoted by $SBPx$. The sending process transmits the data by utilizing the free buffers from its SBP. The buffers in SBP are structures and called *cells*. Furthermore, every process manages an array of pointers, where each pointer references the head of Send Queues.
Figure 5.2: *Lock-free channels for sending/receiving small/large messages*

Figure 5.2 is an illustration of the design using four processes identified by their PIDs\(^\text{10}\) 0 through 3.

**Mechanism**

As mentioned earlier, this design classifies messages based on their size. The smaller messages, identified to occur more frequently, are directly copied over to the receiver buffer. By doing this, it avoids the overhead involved for sending large messages. On the other hand, copying larger messages into the receive buffers is neither efficient in performance nor scalable. Hence, it

\(^{10}\text{Process ID}\)
adopts a mechanism involving lock-free channels to transfer larger chunks of data.

The design functionality is explained with the help of the use-cases below:

**Transferring Messages of Small Size**
This use-case explains the work-flow involved when transferring small message between process 2 and process 3. Figure 5.2 illustrates this use-case, where process 2 and process 3 are the sender and receiver respectively.

1. Since the sender has dedicated access to the receiver’s receive buffer, it copies the small message from the source buffer directly into the receive buffer.

2. The data in this receive buffer is then copied back by receiver into the required destination buffer.

The setup thereby avoids the overhead involved in communicating via lock-free channels — meant for large messages.

**Transferring Messages of Large Size**
This use-case shows how a large message is transferred between two processes using the mechanism meant for large-sized messages. The mechanism supports packetizing very large messages. Figure 5.2 illustrates this use-case where process 0 and process 2 are the sender and receiver respectively. The sender operations are:

1. Sender copies the message from the source buffer into a free cell allocated from the send buffer pool and marks the cell as busy.

2. The newly loaded cell is enqueued [17] to the appropriate sending queue.

3. A control message containing the address of the loaded cell is written into the receiver’s receive buffer by the sender.

The receiver operations are:

4. Receiver polls the receive buffer for the control message. Subsequently, the address of the loaded cell containing the actual data is read.
5. From this information, the receiver accesses the cell, which is enqueued in the send queue corresponding to it in the sender’s process.

6. Receiver copies the data in the referenced cell to the destination buffer and marks the cell as free.

Step 5 is useful for packetized transmission of message. A large chunk of message is broken down into multiple packets of the size of a cell to be loaded. Since the packets are queued, it suffices to transmit the control message with the address of starting packet. The addresses of the remaining packets need not be explicitly transmitted as they can be accessed by iterating through the queue. Thus, the receiving process can start copying the packets to its destination buffer before the entire message is sent out by the sender.

The cells are marked as busy and free by the sending and receiving processes respectively. Marking a cell as busy indicates that the data within the cell has not yet been consumed by the receiving process. The receiving process marks the cell as free in order to indicate to the sender process that it has consumed the data and that the cell is available for new data to be loaded. Thus, the design ensures that the cells are reclaimed into the sender’s buffer pool without overwriting the unread data.

**Design Analysis**

The design [17] ensures that each resource is only shared between one sending and one receiving process. The resources are designed to form a lock-free inter-core communication channel, satisfying Req3. As a result, the contention-time for locked resources is nil. Since, the sender and receiver can overlap their sending and receiving operations, latency is reduced further.

Since the sender process reclaims the previously freed cells, it is likely that these cells already exist in the cache. Thus, the design exploits temporal locality of the cache [19]. As a result, the sending process does not have to fetch the cell from memory, thereby saving time.

**Reflections on the Design**

By setting up an independent communication channel (Send Queue + Receive Buffer) between a pair of communicating processes, this design achieves
a lock-free ICPC mechanism. However, the design violates Req2 as it is not scalable from memory point of view. This is because it requires $p^*(p-1)$ communication channels for $p$ inter-core communicating processes.

The Send Queue is an implementation of a shared linked list where the queued cells point to each other. These pointers are later sent to the receiver process via control messages. However, regular pointers contain invalid references once outside the process in which they are created; regular pointers cannot make cross-process dereferences, as will be discussed in section 5.3.3. As an alternative, modifications to the design should replace regular pointers with offset pointers — references to objects within the shared send queue become relative to the base address where the shared segment is attached [24]. However, this modification will restrict the capacity of Send Queue to a single shared segment [24]. As a result, the modification will require flow-control mechanisms to prevent overflow of the queue, which becomes an overhead thereby violating Req4. Although a lightweight solution, this design deviates from the ICPC design criteria for LINX by a large margin.

5.3.2 Design II: Zero Copy Inter-Core Message Passing

Zero-copy Inter-core Message Passing or ZIMP [16] is an inter-core communication mechanism that improves data transfer throughput by reducing the memory copy operations. The sending process does not buffer a message locally before transmitting it. Instead the sender creates the message directly in a communication channel. Thus, copying the message to be sent is avoided. This protocol satisfies Req1 as it is built into the user space. Thereby, it does not make use of expensive system calls. The protocol is furthermore designed for broadcast mechanisms whereby the sender does not have to transmit the same message multiple times to the many receiving processes. This is an efficient mechanism in one-to-many transmissions where no sender is allowed to hog the communication channel.

Architecture

This protocol [16] makes use of communication channel — a circularly linked buffer shared between a group of inter-communicating processes. The size and the maximal amount of messages that can be stored are determined
during the initialization phase of the channel and remain fixed throughout the channel lifetime. A group of receivers subscribe to the channel. This enables them to receive all messages sent across that channel.

![Diagram of Zero-copy Inter-core Message Passing](image)

*Figure 5.3: Zero-copy Inter-core Message Passing*

Each channel maintains a variable called `next_send_entry`. `next_send_entry` is read by the sender in order to locate the next free buffer entry in the channel to which it can write the message. Since multiple senders can write to the channel concurrently, access to `next_send_entry` is synchronized by means of a lock. Figure 5.3 illustrates this design. Associated with each buffer entry in the channel, there exists the `bitmap` variable [16], of bit width equal to the number of receivers that have subscribed to the channel. The sending process sets all the bits in the bitmap variable to 1, indicating to the receivers that a
new message has been written to the buffer location. Subsequently, the \textit{i}^{th} bit of the bitmap is reset to 0 once the message is copied by the \textit{i}^{th} receiver into its local buffer. The setting and resetting of bits is done in an atomic manner.

Additionally, each channel makes use of an array variable, \textit{next\_read}, in order to store the index of the buffer entry from which the next messages will be read. Since each entry in the array corresponds to a specific receiver, access and updates to the index values do not require synchronization mechanisms.

Optimizations to the design include padding the ZIMP structures in order to align them with the cache line. This improves cache usage as the structures are not invalidated because of false sharing. Additionally, the bitmaps are allotted an entire cache line in order to speed up the the access to it during polling. The bitmap is fetched from the memory only the first time when it is polled and it remains in the cache until a subsequent update has invalidated this entry in the cache line.

**Mechanism**

Figure 5.3 illustrates the operation of ZIMP. The sender’s operations are:

1. It gets the address of the next available buffer in the channel by reading \textit{next\_send\_entry}. It subsequently updates this variable to point to the next buffer entry, allowing other senders to write to the channel concurrently. The get and update \textit{next\_send\_entry} is an atomic operation.

2. The sender then waits until all the receivers have read the old message that is stored in this buffer. It does this by polling to check if all the bits in the bitmap associated with the buffer are cleared. Once satisfied, it writes the new message directly to the buffer.

3. It then updates the bitmap by setting all bits to 1.

The receiver’s operations are:

1. It obtains the location of the buffer to read from by reading its corresponding entry from the \textit{next\_read}[r] array variable.

2. If its associated bit in the bitmap corresponding to this buffer entry indicates an unread message, it updates the \textit{next\_read}[r] value. Then it copies the the new message into its local buffer.
3. It then updates its associated bit in the bitmap by resetting it to 0.

4. However if no new message is found at the buffer entry, it keeps polling the bitmap until a new message is detected.

Design Analysis

The authors of [16] evaluated ZIMP for throughput performance by testing it on a 24-cored HP Proliant DL165 G7 [16] machine. The machine consists of two AMD Opteron 6164HE processors running at 1.7GHz and with a memory of 48GB. Each processor consists of 12 cores evenly split into 2 sets sharing an L3 cache of 6MB [16]. The cores individually comprise an L1 cache of 64kB and an L2 cache of 512kB.

The micro-benchmarking was carried out by transmitting messages of sizes varying between 1B to 1MB for two scenarios 1 receiver and 23 receivers. Figure 5.4 presents the micro-benchmark results for communication between a transmitter and a single receiver. It can be observed that the throughputs of Barreelfish MP and ZIMP are much higher compared to the other IPC mechanisms. Interestingly, Barreelfish MP performs better than ZIMP for messages of size less than 1kB. This is attributed to the fact that ZIMP induces a slight overhead caused by waiting time required to access and update the ZIMP structures. However, for large sized messages (> 1kB), it outweighs the performance of Barreelfish MP and other IPC mechanisms. The throughput gains are even more significant in the case of data transfer involving a single transmitter and multiple receivers as shown in figure 5.5. The superior performance of ZIMP is achieved because it does not require the sender to retransmit the same data to multiple receivers. Thus, ZIMP is a better inter-core communication protocol for broadcasting large sized chunks of data to multiple receivers. Although, the information about size of the communication channel is not provided in [16], it can be pointed out that the larger the buffer count of the communication channel, the more accommodative this design is towards slower receivers. This is because a longer channel yields a slower buffer reuse rate to the sender. The time gained from the slower reuse rate of the buffers compensates the time taken by slower receivers to read the message off the channel buffers. In other words, the sender does not have to wait for a longer duration before the buffers become write-ready. Thus, one
can infer that, a longer communication channel influences the performance positively by preventing the sender from getting hogged due to slow receivers.

**Reflections on the Design**

Although the sender writes the messages directly to the communication channel, the receivers cannot avoid copying these message into their local buffers. Hence, this is not a true zero copy mechanism. Moreover, it violates Req3 as sending processes wait on the lock protecting next_send_entry. This is a clear bottleneck in the design particularly for intensely communicating processes. Moreover, the senders have to wait until all bits on the
Figure 5.5: Micro-benchmark comparing throughput between various IPC mechanisms between a sender and 23 receivers [16]

bitmap are cleared. Thus, the overall speed of the mechanism is slowed down by the slowest receiver.

On the other hand, the design is built for broadcasting messages. A specific modification that will be of interest is bitmap manipulation. Receiving processes, at the time of setting up the channel, can register the type of messages that they prefer to receive. The senders can then set the bits on the bitmap accordingly such that only the intended receivers read the message. However, additional logic will be required to ensure that all receivers update their next_read value in events of such directed broadcasts. As a result of this modification, the need for separate communication channels for each
message type can be avoided. To summarize, this design is a solution for applications involving broadcast messages such as periodic status requests, synchronizations, etc. However, it adds an overhead for unicast messages, which is a violation of Req4. This design is therefore a partially suitable solution for efficient ICPC in LINX, provided LINX developers desire to extend the feature set to include directed broadcasts.

5.3.3 Design III: Zero Copy Shared Heap

Most IPC mechanisms using shared memory apply double-copying techniques to transfer the message; the first copy occurs between the sending process and the shared memory and the second copy is between the shared memory and the receiving process. Furthermore, such shared memory methods of communication do not scale across the boundaries of a process in a virtual environment for messages containing pointers. This stems from the fact that the shared memory segments attached to one process are not necessarily attached to the same virtual addresses in the other process. Hence, the pointers should be dereferenced so that the actual data is copied to the shared memory. These redundant copying of messages add up to the communication cost as mentioned in 5.1.

Alternatively, Shared Heap Based Zero-Copy Message Passing [24], offers a scalable IPC mechanism, in which the sender process need not buffer a message locally before sending it to the receiver. The runtime allows the sender to instead create the message directly in the shared memory region, accessible by all other processes. The sender then merely shares a special pointer to this message with the receiver via a shared memory. The receiver dereferences this *smart pointer* to use the data directly off the shared memory region instead of writing it back to its local buffer. As a result, this design achieves zero-copy message passing by making use of shared memory and smart pointers.

**Architecture**

This design primarily makes use of two primitives:

*Shared Heap*

Shared Heap is a shared memory area created by stacking a group of sequen-
tial heap items [24]. The size and number of heap items to be created are configured during compile time. These heap items are permitted to be of mixed sizes varying from 50 bytes for smaller blocks to 5000 bytes and more for the larger ones. Such preallocations of memory blocks facilitate the usage of efficient memory allocation and deallocation procedures. However, all heap items belonging to a particular shared heap are required to be allocated from the same memory segment. This ensures that the relative positions of the heap items within the shared heap remain unchanged, regardless of the virtual address of the logical processes to which the shared heaps are mapped. As a result, any logical process can access the heap items of a shared heap by offsetting values relative to the virtual address of a processes to which the base of the shared heap is mapped. The UML$^{11}$ class diagram of the content stored in a heap item is shown in figure 5.6.

![UML Diagram of HeapItem](image)

Figure 5.6: *UML diagram of HeapItem [24]*

It comprises the logical process $lp$ to identify the heap number of the shared heap to which the heap item belongs. The heap item in the shared heap

---

$^{11}$Unified Modeling Language
is identified by the *offset* field. The actual data is stored in the *data* field, which is an array variable. *RefCount* indicates the number of active references made to the heap item. Every logical process owns a shared heap and initializes it during the initialization phase. The heaps are created in the shared memory location and the permissions are set so that every process has access to its own shared heap and the shared heaps created by other processes. The layout of this setup consisting of shared heaps created by two processes is shown in figure 5.7. Every process maintains *HeapCluster*, a global array which holds the virtual memory pointers to these shared heaps.

![Figure 5.7: Layout of shared heaps [24]](image)

*Smart Pointers*

Although the processes have pointers pointing to the same shared heaps, the virtual addresses where these heaps are attached differ between processes. Since the virtual addresses referenced by regular pointers are invalid beyond the address space of a process, this design instead passes meta-information...
of the actual shared data by making use of smart pointers. A smart pointer is a structure comprising information that can uniquely identify the memory location across processes. The UML block diagram of a smart pointer is shown in figure 5.8.

![UML diagram of BPtr](image)

**Figure 5.8: UML diagram of BPtr [24]**

The meta-information includes heap number of the shared heap that stores the actual message, an offset which points to the heap item within the shared heap. It also contains a heapItem pointer that points to the heap item referenced by the heap number of shared heap containing the referenced data. The heapItem pointer is automatically updated when the smart pointer is passed between processes so that it always points to the right heap item.

**Mechanism**

The sender creates new messages by calling `Create` method. The runtime allocates the message in a free heap item, chosen on best-fit basis. It updates the associated smart pointer `BPtr` with the lp number of the process that
created the message and the offset number pointing to the heap item in which the message has been allocated. It increments the refcount field, indicating that a process made an active reference to the heap item. The BPtr is then returned to the process which created the new message. Figure 5.9 illustrates the work-flow behind zero-copy technique to transfer messages between processes using shared heap. If necessary, requests for additional shared heaps are made to the kernel. The kernel then creates new heap structures and gives them unique heap numbers [24]. All processes later connect to the newly created shared heaps and update their respective heap clusters.

Figure 5.9: Shared heap based zero-copy operations. (a) A message is created in the shared heap of LP A and is accessed by LP A using the smart pointer (b) LP A copies the smart pointer into a shared memory region which is then copied by LP B into its memory space (c) LP B directly reads the message off the shared heap of LP A using the smart pointer [24]
To send the message, the sender copies the message’s BPtr into a shared memory region agreed upon by the communicating processes. This operation merely copies two meta data viz., the heap number and the offset of the heap item. The receiver reads the BPtr from this shared piece of memory by using GetPointer function, which then returns the correct virtual address to the heap item storing the message. The operations involved in this procedure are enlisted below:

1. GetPointer uses lp to index the heap cluster and selects the virtual pointer that references the right shared heap.

2. It adds the offset meta-datum to the value obtained in step 1 to compute the virtual address of the heap item.

3. The virtual address is updated into the HeapItem field of the copied BPtr. It also increments the refcount of the associated heap item in order to reflect the count of active references made to it.

The receiving process then dereferences the HeapItem pointer to read the message directly off the shared memory without copying it. The refCount of the associated heap item is automatically decremented by one when the Bptr goes out of scope in a process. The heapItem is deallocated once the refCount reaches zero and thus becomes available for a future allocation.

**Design Analysis**

The authors of [24] tested the protocol for efficiency by running SimpleSim - a distributed discrete event simulator [24]. The first experiment consisted of 8 logical processes, which were used for inter-communication purpose. Messages of varying sizes ranging from 500 to 50000 bytes were passed between these processes for a period of 25000 simulator seconds [24]. The trials were conducted for 10 iterations. The averaged results were plotted as runtime vs size of message curve and is depicted in the figure 5.10. The results were compared with other message passing protocols viz., Boost and MPI full copy.

Boost [2] is a zero-copy message passing technique that makes use of offset pointers to reference memory locations within a shared memory segment. An offset pointer holds the difference between the offset pointer’s address
Figure 5.10: The runtime of SimpleSim with 8 logical processes and variable messages sizes [24]

and its referenced memory location [24]. Since the difference in the base addresses of two different shared memory segments mapped into the virtual space of one process is not guaranteed to be the same between processes [24], the reference made by an offset pointer to a data item residing in a different shared memory segment is invalid. As a result, Boost design often utilizes only 1 shared memory segment between all the logical processes [24] in order to maintain valid offset pointer references. Consequently dereferencing operations to the single shared segment from multiple logical processes are serialized. This bottleneck causes application using Boost to take more time to finish the benchmark as is evident from the plot. On the other hand, the performance of the two zero-copying techniques are consistent, suggesting they scale well with message size. More precisely, the shared heap based zero-copy outperforms Boost because the smart pointers allow multiple shared heaps to be dereferenced concurrently, if the messages are not stored in the same shared heap.

The emphasis on scalable performance in the case of shared heap mechanism
Figure 5.11: The runtime of SimpleSim with a message size of 7000 bytes, varying the number of logical processes [24]

is more pronounced in figure 5.11 whose plots depict the results obtained when the experiment was performed by passing messages of a constant size of 7000 bytes but for a varying amount of logical processes. The near linear performance in the case of shared heap mechanism can again be attributed to its ability to allow concurrent dereferences to the disjointed heaps.

To summarize, the shared heap design reaps the benefits of zero-copy technique while not falling into the serialization trap and hence the reason for its superior inter-process performance in terms of data transfer rate.

Reflections on the Design

As there is no physical replication of the message amongst any of the communicating processes, the design is a true zero-copy mechanism. The shared heaps are dynamically extensible. Hence there is no limitation on the message size. The near-constant runtime curve suggests that this design is
scalable across core counts and message sizes. This is because of the support for concurrent access to disjointed sections of the shared heap as already mentioned. An improvement to this design should consider permitting concurrent access to the same heap item, provided all participating processes access the data object in read mode. It can be achieved if additional information about the access mode, whether to read or modify the data object, is passed as a parameter while accessing the heap item. It improves the support for parallelism and makes it a strong contender for adaptation into LINX.

5.3.4 Summary of the Case Study Designs

Table 5.1 consolidates the details of the designs presented in the previous sections of this chapter. It is evident on comparison that Design III is the most favourable of the designs as it meets most of the design requirements set for ICPC using LINX. However, since all these designs are based on a shared memory model and do not feature a memory protection mechanism, the processes and messages are vulnerable to being unintentionally corrupted by other participant processes.
Table 5.1: Summary of the case study designs

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Lock-free Channels</th>
<th>Zero Copy Inter-Core Message Passing</th>
<th>Zero Copy Shared Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoids system call</td>
<td>Yes</td>
<td>Yes</td>
<td>Used during setup</td>
</tr>
<tr>
<td>Scalability over message size</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Scalability over core count</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Lock-free communication mechanism</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Communication overhead</td>
<td>High</td>
<td>Low for broadcast message</td>
<td>Negligible</td>
</tr>
</tbody>
</table>
Chapter 6

Implementation Details

This chapter details the implementation work done towards porting Enea OSE to TILEPro64. It also describes the design proposed to improve ICPC throughput for LINX on Linux.

6.1 Adapting Enea OSE to TILEPro64

The system as left by author of [25] was capable of handling event triggered task scheduling. Events were triggered by sending signals between a pair of processes. The system was capable of printing Ramlog messages.

However, neither time-based scheduling nor time-related system calls could be performed since the timers were non-functional. Besides, the system was not capable of handling interrupts and context switches. Hence, the outcome expected from this part of the implementation phase is a successful Tilera port of a minimal version of OSE. The system should be capable of handling interrupts and schedule timer processes.

6.1.1 Development Environment

The implementation of this thesis work was carried out using Tilera’s Multi-core Development Environment (MDE). It included ANSI standard C and C++ compilers, standard command-line tools like GDB and its own simulator apart from other software components. Tilera’s Simulator Development Platform used in this thesis was configured with a physical memory
of 128MB. The MDE ran on Enea’s development server operating on Linux.

6.1.2 Milestones Reached

This section comprehensively describes the implementation details of adapting OSE to a tile.

Handling Interrupts

As a first step, interrupt handling had to be accomplished. The base address for vectored interrupts in Bare Metal Environment was located at 0xFE000000. This was an address space that lay outside the configured physical memory. Moreover, the system ran with no MMU\(^1\) support and hence would not be able to make use of virtual addresses space. The approach sought to fix this problem was to manually map the vector locations into reachable addresses on the configured physical memory. The region of physical memory used for this purpose was marked off as *Exception region*. Thus a successful static mapping of entries in both ITLB\(^2\) and DTLB\(^3\) of the processor was accomplished. However, the vectored locations did not provide sufficient space to write an entire ISR. Hence, jump codes were installed to re-vector to the interrupt controller. Figure 6.1 illustrates how the interrupts were statically mapped to the reachable physical addresses.

Following this, the driver for PIC\(^4\) was developed. It included implementing interrupt vector handler and interrupt mask handler amongst other functionalities. Unlike many other microprocessors, Tilera does not feature an interrupt status register. With such a shortcoming, implementing the interrupt vector handler that selects the highest priority pending interrupt to be serviced, became particularly challenging. A workaround was achieved by implementing a software based interrupt status register. The design considered other Tilera specifics such as the program counter’s content can not be read unlike from a usual register and the fact that Tilepro64 provides 3 system save registers, which could be used for saving the processor’s states during context switches. The workaround is illustrated in Figure 6.2. It is

---

1 Memory Management Unit  
2 Instruction Translation Look-aside Buffer  
3 Data Translation Look-aside Buffer  
4 Programmable Interrupt Controller
Figure 6.1: *Interrupts vectors relocated via statically mapped TLB entries*

Figure 6.2: *Software workaround for implementing interrupt status register*

also important that the state of the registers be stored so that they can be restored back once the interrupt has been serviced. This mechanism is called interrupt context handling. The codes for setting up the interrupt stack and saving and restoring the context using the stack, were implemented in this thesis. The PowerPC’s port of OSE was used as a reference code. The codes
performed many stack manipulation operations and extensive debugging was required in order to isolate the problem areas in already written parts of the code. Problems related to corruption of interrupt stack frames were fixed during this thesis.

**Timer Driver**

The other important part implemented in this thesis was setting up the tick based scheduler. The scheduler was already available. However, the input to the scheduler - ticks\(^5\) - were missing. In order to achieve this, the timer driver was implemented. Codes were written in the scope of this thesis to initiate the timer control register of the tile processor. It was configured as a down counter with a calculated underflow reload value so that timer interrupts could be generated at a uniform interval. Every time the timer interrupt occurs, the tick gets updated and the scheduler takes over. Thus, the functionalities of timer driver were implemented.

**Demo Application**

A demo application was built in order to prove that the implementation goals were reached. The demo comprised 3 processes: a timer process named **Timer** and two priority processes named **Demo 1** and **Demo 2**. The **Timer** process was configured to send a wake-up signal, once every fifth system tick, to **Demo 1**. Soon after receiving the wake-up signal, **Demo 1** messaged a greeting to **Demo 2**. **Demo 2** waited for 2 ticks before greeting back **Demo 1**. On the reception of a greeting, **Demo 2** waited on a new wake-up signal from **Timer** processes. This marks the end of one demonstration cycle. The illustration of the demonstration is shown in Figure 6.3

The periodic scheduling of the **Timer** process shows that the system can now schedule time-based processes. In addition, the delay calls executed by **Demo 2** show that the timer functionality is usable for other time bound operations implemented in OSE. The application did not crash after a long duration proving the point that interrupt context handling is taken care of.

Please refer to appendix A.1 and A.2 for the application code and the application output respectively.

\(^5\)In this implementation: 1 tick = 1 millisecond
6.2 Design Proposal for a High-Throughput ICPC in LINX

The base design as presented in [23] forms a foundation for the design to be proposed in this section. In the base design, the sending process allocated a message buffer from a global shared buffer pool. It wrote a message directly into the allocated buffer. Next, a send operation was performed which took the pointer to the allocated buffer and destination SPID as parameters. The destination core was computed using the destination SPID. The send operation was completed by posting the message pointer and the destination address over the NoC\(^6\) to the destination core. The destination core could then process the received message pointer via an ISR or polling and queue it onto the intended receiving process’ receive queue. The receiver process used the message pointer to directly read off the shared buffer without requiring to copy the data.

As it is evident from Design III that it satisfies the design specifications presented in section 5.2.4, it has been chosen for adaptation into the base design with some modifications. Figure 6.4 illustrates the design proposal. The steps are enumerated below:

\(^6\)Network on Chip
Figure 6.4: A design proposal to adapt a shared heap design to LINX for high-performance ICPC

1. Upon `linx_open()`, in addition to the creation of an endpoint, the LINX runtime should check to see if a shared heap (SH) has been associated with the process owning the created endpoint. If not, a shared buffer pool is created in the shared buffer pool and attached to the created endpoint’s owner. A heap number `lp` is returned the first time an SH is created. This has to be a unique identifier of the SH.

2. The runtime should then register the `lp` in a global memory space, such that every new processes created reads the global space to attach the
SH into their memory. An alternate design should consider attaching to the shared heap of the hunted endpoint after a successful hunt.

3. The `send()` operation, should make use of smart pointer instead of the regular pointer.

However, similar to the designs studied earlier, this design is based on shared memory model and does not seek the memory protection services of the kernel layer beneath, making it vulnerable to unintentional memory corruption by participating processes. In particular, the receiver processes, not restricting to the specific heap items shared by the sender process, can illegally step through the entire shared heap space of the sender by merely incrementing the smart pointer. Thereby, this design points to the trade-off between a light-weight solution and memory protection.

Appendix B comprises the test cases that were used in validating the design against the requirements presented in section 5.2.4. In essence, these test cases validated the design for system call usage, scalability over message size and core count, lock-free communication mechanism and communication overhead. Further, these test cases should serve as reference for developing more exhaustive test cases that will be required in future during the implementation of the proposed design.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

As an outcome of this thesis, OSE with minimal functionality has been adapted to TILEPro64 successfully. The thesis implementation has contributed to extending the features of TILEPro port of OSE thereby providing a platform for further development work in the direction of achieving full-fledged real time support. The present system is capable of handling interrupts and uses system timer to schedule tasks. In addition, the system timer is made available for use by other time-related system calls. However, the UART driver could not be implemented owing to configurability issues. Tilera’s Simulator Development Platform did not support configuring the RShim to host a UART interface. Nonetheless, a demo application was built to verify the other implemented modules. The results have shown that the scheduler is functional and tasks can be deployed and scheduled according to the OSE scheduling principles on the TILEPro port of OSE. However, the tasks could not take advantage of virtual memory as memory management is not yet supported in the current implementation. The implementation work was predominantly carried out in the BSP and HAL layers suggesting that OSE is capable of supporting many-cores with minimal modifications to the core layers.

A design for high-throughput ICPC in LINX for Linux in many-cores was proposed after studying and comparing some suitable IPC designs. The proposed design builds on the concept of using shared heaps and smart pointers
to communicate, whereby the physical replication of the actual message can be avoided. In addition, the design avoids the costly system calls for communication. Moreover, it can be concluded that the design does not require any modifications to the RLNH and CM layers. The adaptation design points towards using the NoC of a many-core platform. In case of TILE platform, this NoC can be a User Dynamic Network which is available for communication to the user-level through vendor supplied library calls.

7.2 Future Work

In order to accomplish a full-fledged port of OSE for the TILE platform, routines for Inter-Processor Interrupts have to be developed. This is necessary for transferring the OSE image from tile to tile and also for inter-core communication purposes.

It would also be interesting to study the performances of various memory distribution strategies between the tiles. Memory placement plays an important role in load distribution for memory accesses by the cores and a strategic placement of memory will ensure optimal access times to the shared heaps of individual cores. Additionally, the LINX runtime has to be studied for further modifications in order to accommodate the proposed ICPC design without exposing the smart pointers to the users. This will ensure the re-usability of LINX APIs and portability with existing applications.
Appendix A

Demonstration

A.1 Application Code
#define DEMO_SIG (101) /* !-SIGNO(struct my_signal)=! */
#define WAKE_ME_SIG (102)

void ose_core_start_handler_0(void);

struct user_signal
{
    SIGSELECT sig_no;
    char *message;
};

union SIGNAL
{
    SIGSELECT sig_no;
    struct user_signal user_signal;
};

static const SIGSELECT demo_sig[] = {1, DEMO_SIG};
static const SIGSELECT wake_me_sig[] = {1, WAKE_ME_SIG};

static char *message1 = "Howdy ho!";
static char *message2 = "Tally ho!";

PROCESS pid1, pid2, timer_pid;

/* Our demo process. Add test code here or create more processes */

OS_PROCESS(timer)
{
    union SIGNAL *sig;
    sig = alloc(sizeof(struct user_signal), WAKE_ME_SIG);
    send(&sig, pid1);
}

OS_PROCESS(demo1)
{
    char temp_buff[50];
    union SIGNAL *sig1, *sig2;
    sig2 = alloc(sizeof(struct user_signal), DEMO_SIG);
    for(;;)
    {
        ramlog_printf("Demo1: Sleeping\n");
        sim_write("Demo1: Sleeping\n");
        sig1 = receive(wake_me_sig);
        ramlog_printf("Demo1: Woken up by timer interrupt process\n");
        sim_write("Demo1: Woken up by timer interrupt process\n");
        sig2->user_signal.message = message1;
        send(&sig2, pid2);
        sig2 = receive(demo_sig);
        ramlog_printf("Demo1: Demo2 greets %s\n", sig2->user_signal.message);
        sprintf(temp_buff, "Demo1: Demo2 greets %s\n", sig2->user_signal.message);
        sim_write(temp_buff);
        sig2->user_signal.message = message1;
    }
}

OS_PROCESS(demo2)
{
    char temp_buff[50];
    union SIGNAL *sig;
    for(;;)
    {
        sig = receive(demo_sig);
        ramlog_printf("Demo2: Demo1 greets %s\n", sig->user_signal.message);
        sprintf(temp_buff, "Demo2: Demo1 greets %s\n", sig->user_signal.message);
        sim_write(temp_buff);
        sig->user_signal.message = message2;
        delay(2);
        send(&sig, pid1);
    }
}
/ * Called during system startup when all processes are created (ose_devman in * particular). In the OSE reference system, it is called by the core * supervisor. */

void bspStartOseHook2(void)
{
    timer_pid = create_process(OS_TI_PROC,
        "timer",
        timer,
        1025,            /* Stack size */
        0,               /* Priority */
        (OSTIME) 5,      /* Timeslice */
        (PROCESS) 0,
        (struct OS_redir_entry*) NULL,
        (OSVECTOR) 0,
        (OSUSER) 0);
    start(timer_pid);
    pid1 = create_process(OS_PRI_PROC,
        "demo1",
        demo1,
        256,             /* Stack size */
        15,              /* Priority */
        (OSTIME) 0,      /* Timeslice */
        (PROCESS) 0,
        (struct OS_redir_entry*) NULL,
        (OSVECTOR) 0,
        (OSUSER) 0);
    pid2 = create_process(OS_PRI_PROC,
        "demo2",
        demo2,
        256,             /* Stack size */
        15,              /* Priority */
        (OSTIME) 0,      /* Timeslice */
        (PROCESS) 0,
        (struct OS_redir_entry*) NULL,
        (OSVECTOR) 0,
        (OSUSER) 0);
    sim_write("Demo processes created\n");
    start(pid1);
    start(pid2);
    ramlog_printf("PID of the demo1 process is: %u\n", pid1);
    ramlog_printf("PID of the demo2 process is: %u\n", pid2);
A.2 Application Output
[0] 0.000:ROFS: No embedded volume found. Use rofs_insert host tool to insert one.
[0] 0.000:Detected TILEpro64, PVR 0xffff, D-cache 8 KByte, I-cache 16 KByte
[0] 0.000:mm:
  mm_open_exception_area(cpu_descriptor=36a004, vector_base=0xfc000000, vector_size=256)
[0] 0.000:CPU_HAL_TILEPRO: init_cpu.
[0] 0.000:mm: mm_open_exception_area MMU=0
[0] 0.000:biosInstall Module: biosInstall
[0] 0.000:biosInstall Module: biosOpen
[0] 0.000:biosInstall Module: biosUninstall
[0] 0.000:biosInstall Module: biosRunlevel
[0] 0.000:mm: mm_install_exception_handlers: entry
[0] 0.000:mm: start parsing log_mem string: krn/log_mem/RAM
[0] 0.000:mm: max_domains:255 @ 1206a80
[0] 0.000:Boot heap automatically configured. [0x0036c000-0x011fffff]
[0] 0.000:init_boot_heap(0x0036c000, 0x011fffff)
[0] 0.000:phys_frag:
[0] 0.000:Data cache not enabled in configuration. For NOMMU only.
[0] 0.000:Instruction cache not enabled in configuration. For NOMMU only.
[0] 0.000:mm: map_regions()
[0] 0.000:mm-meta-data: [0x001e5000-0x0011ffff]
[0] 0.000:mm init completed
[0] 0.000:has_mmu= 0
[0] 0.000:biosInstall Module: MM
[0] 0.000:biosInstall Module: MMAN
[0] 0.000:mm: initialization completed.
[0] 0.000:biosInstall Module: ose
[0] 0.000:biosInstall Module: CACHE
[0] 0.000:Cache bios installed.
0.000: biosInstall Module: RAMLOG
0.000: biosInstall Module: SYSPARAM
0.000: biosOpen Module: ose
0.000: syscall ptr: 2f0fa0 bios ptr: 2f1080
0.000: biosOpen Module: MM
0.000: Starting syspool extender.
0.000: Starting mainpool extender.
0.000: biosInstall Module: OSE_PTHREAD
0.000: kernel is up.
0.000: Starting RTC.
0.000: biosInstall Module: RTC
0.000: Starting system HEAP.
0.000: biosInstall Module: HEAP
0.000: biosOpen Module: HEAP
0.000: Starting FSS.
0.000: biosOpen Module: RTC
0.000: Starting PM
0.000: Starting SHELLD.
0.000: OSE5 core basic services started.
0.000: ROFS: /romfs: No volume found. Has ose_rofs_start_handler0() been called?
0.000: Starting DDA device manager
0.000: Installing static device drivers.
0.000: biosOpen Module: HEAP
0.000: dda: ddamm_alloc_uncached from MM(16384) = 0x4b5000
0.000: devman: Started (log_mask=0x3)
0.000: Register driver pic_tilepro
0.000: Register driver timer_tilepro
0.000: Activating devices.
0.000: dda/pic: Attach to dda IRQ0 (32 lines)
0.000: dda/pic: successfully activated.
0.000: dda/timer: Attach IRQ25 to dda/pic (vector 1).
0.000: biosInstall Module: HWTIMER
0.000: Timer handler attached
0.000: dda/timer: Enable IRQ25 (vector 1)
0.000: Timer handler process created
0.000: System timer activated. dda/timer: successfully activated.
0.000: dda/uart: Activation failed, no suitable driver found.
0.000: Demo1: Sleeping
0.000: PID of the demo1 process is: 65561
0.000: PID of the demo2 process is: 65562
0.000: Starting SERDD.
0.000: Starting CONF.
0.000: Adding program type APP_RAM execution mode user
0.000: APP_RAM/text=phys_mem:RAM log_mem:RAM cache:0xc perm:0x707
0.000: APP_RAM/pool=phys_mem:RAM log_mem:RAM cache:0xc perm:0x303
0.000: APP_RAM/data=phys_mem:RAM log_mem:RAM cache:0xc perm:0x303
0.000: APP_RAM/heap=phys_mem:RAM log_mem:RAM cache:0xc perm:0x303
0.000: No conf for: pm/progtype/APP_RAM/heap, using default.
0.000: biosInstall Module: PM
0.000: biosOpen Module: PM
0.000: Adding program type SYS_RAM execution mode supervisor
0.000: SYS_RAM/text=phys_mem:RAM log_mem:RAM cache:0xc perm:0x707
0.000: SYS_RAM/pool=phys_mem:RAM log_mem:RAM_SASE cache:0xc perm:0x303
0.000: SYS_RAM/data=phys_mem:RAM log_mem:RAM cache:0xc perm:0x303
0.000: SYS_RAM/heap=phys_mem:RAM log_mem:RAM cache:0xc perm:0x303
0.000: No conf for: pm/progtype/SYS_RAM/heap, using default.
0.000: Starting RTL ELF.
OSE5 core platform started.

Timestamp format tick.usec: (1 tick = 1000 micro seconds)

Demo1: Woken up by timer interrupt process

Demo2: Demo1 greets "Howdy ho!"

Demo1: Demo2 greets "Tally ho!"

Demo1: Sleeping

Demo1: Woken up by timer interrupt process

Demo2: Demo1 greets "Howdy ho!"

Demo1: Demo2 greets "Tally ho!"

Demo1: Sleeping

Demo1: Woken up by timer interrupt process

Demo2: Demo1 greets "Howdy ho!"

Demo1: Demo2 greets "Tally ho!"

Demo1: Sleeping

Demo1: Woken up by timer interrupt process

Demo2: Demo1 greets "Howdy ho!"

Demo1: Demo2 greets "Tally ho!"

Demo1: Sleeping

Demo1: Woken up by timer interrupt process

Demo2: Demo1 greets "Howdy ho!"

Demo1: Demo2 greets "Tally ho!"

Demo1: Sleeping

Demo1: Woken up by timer interrupt process

Demo2: Demo1 greets "Howdy ho!"

Demo1: Demo2 greets "Tally ho!"

Demo1: Sleeping

Demo1: Woken up by timer interrupt process

Demo2: Demo1 greets "Howdy ho!"

Demo1: Demo2 greets "Tally ho!"

Demo1: Sleeping
Appendix B

Test Cases for LINX ICPC Design

B.1 Test Case 1

This test verifies Req1 i.e., the design is built in user space and makes reduced usage of system calls for on-chip IPC.

B.1.1 Use Case

1. Deploy two user level processes on two different cores.
2. Make one process the Sender and another one the Receiver.
3. Pass a smart pointer dereferencing the message to be communicated.
4. Trigger an `strace` from the Senders end after a successful send operation over a User Dynamic Network on Chip.
5. Retrieve the smart pointer at the Receivers end and trigger an `strace` after successfully read the message from the shared heap.

B.1.2 Desired Outcome

The `strace` generated log does not report any system calls that trace back to the send/receive ICPC operations.
B.2 Test Case 2

This test verifies Req2 partly i.e., the design exhibits scalability across varying message sizes.

B.2.1 Use Case

1. Deploy an arbitrary number of inter-core communicating processes on the different cores of a many-core chip.
2. Send and receive smart pointers that dereference messages of a particular size.
3. Run this setup until a certain number of messages have been communicated by each process.
4. Measure the time taken for the setup to complete execution.
5. Perform the test iteratively for variable message sizes.
6. Plot a curve tracing the measured execution times against message sizes.

B.2.2 Desired Outcome

The setup exhibits a uniform execution time for any size of the messages.

B.3 Test Case 3

This test verifies Req2 partly and Req3 i.e., the design exhibits scalability across cores and supports mechanisms that are relatively free of locks for simultaneous communication between many pairs of processes.

B.3.1 Use Case

1. Deploy an arbitrary number of inter-core communicating processes on the different cores of a many-core chip.
2. Send and receive smart pointers that dereference messages of a particular size.
3. Run this setup until a certain number of messages have been communicated by each process.

4. Measure the time taken for the setup to complete execution.

5. Perform the test iteratively by scaling-up the number of on-chip communicating cores until all the available cores have been engaged in inter-core process communication.

6. Plot a curve tracing the measured execution times against the number of cores.

B.3.2 Desired Outcome

The setup exhibits a uniform execution time for any number of communicating cores and processes.

B.4 Test Case 4

This test verifies Req4 i.e., the design introduces minimal overhead for communication.

B.4.1 Use Case

1. Deploy two user level processes on two different cores.

2. Make one process the Sender and another one the Receiver.

3. Pass smart pointers dereferencing messages of variable sizes to be communicated.

4. Plot a curve tracing the ratio of size of the smart pointer to size of the message.

B.4.2 Desired Outcome

The curve traces a negative slope indicating that the overhead becomes negligible as the size of the message to be communicated increases.
Bibliography


[14] ENEA AB. Linx for linux user’s guide.


