Power Management for a Many-core Platform

Master of Science Thesis

SEBASTIAN ULLSTRÖM

Examiner: Ingo Sander, KTH
Supervisors: Detlef Scholle, XDIN AB
Barbro Claesson, XDIN AB

TRITA-ICT-EX-2012:140
Abstract

The MANY (Many-core programming and resource management for high-performance embedded systems) project aims at providing the industry with tools for developing software on multi- and many-core.

This thesis makes its first contribution to the project through an investigation of power management techniques for processors with more than one core. The TILEPro64 processor from Tilera is used as a premise of discussion, and its network-on-chip architecture is also considered. A prototype implementation of a task mapping algorithm shows promising results for threads communicating between cores over the TILEPro64 on-chip networks. Traditional processor scheduling is introduced followed by an analysis of different power-aware multi-core scheduling algorithms.

In a previous thesis project, a port of the real-time operating system Enea OSE to the TILEPro64 was started. A second part of this thesis continues this work, with development on the hardware abstraction layer and board support package.
Contents

List of Figures

List of Abbreviations

1 Introduction .................................................. 1
  1.1 Background ................................................. 1
  1.2 Problem statement .......................................... 2
    1.2.1 Power management in many-core ....................... 2
    1.2.2 Porting OSE to the TILEPro64 ......................... 2
  1.3 Method ...................................................... 3
  1.4 Delimitations .............................................. 3

2 Power in multi-core ........................................... 5
  2.1 Processor power model ...................................... 5
    2.1.1 Dynamic power dissipation ............................ 5
    2.1.2 Static power dissipation ................................ 6
    2.1.3 The single-core power issue ......................... 7
  2.2 Managing processor power in hardware ....................... 8
    2.2.1 Gating .................................................. 8
  2.3 Multi-core scalability concerns ........................... 9
    2.3.1 Augmenting Amdahl’s law ............................. 9
    2.3.2 The KILL rule ......................................... 10
  2.4 Conclusions ................................................ 11

3 Network-on-chip .............................................. 13
  3.1 Network-on-chip architecture ............................. 13
  3.2 Network-on-chip power management ......................... 14
    3.2.1 Power model ........................................... 15
  3.3 Network-On-Chip task mapping ............................. 15
    3.3.1 Task model ............................................. 15
    3.3.2 Communication energy mapping ....................... 16
6 Power-aware multi-core scheduling 39
6.1 Dynamic voltage and frequency scaling 39
   6.1.1 The GRUB-PA algorithm 40
   6.1.2 The GSSR algorithm 40
6.2 Leakage-aware scheduling 41
   6.2.1 Shut-down overhead 42
   6.2.2 Procrastination scheduling 42
   6.2.3 Task migrating algorithms 43
6.3 Conclusions 44

7 Implementation: Task mapping on the TILEPro64 47
   7.0.1 Example tasks 47
   7.0.2 The algorithm 47
   7.0.3 Results 48
7.1 Conclusions 49

8 Enea OSE 51
8.1 Architecture overview 51
8.2 Processes and load modules 51
8.3 Priority and scheduling 52
   8.3.1 Shared resources 52
8.4 Signals 53
8.5 Semaphores and mutexes 54
   8.5.1 Semaphores 54
   8.5.2 Fast semaphores 54
   8.5.3 Mutexes 54
8.6 Power management 54
8.7 Hardware abstraction layer 55
8.8 OSE for multi-core 55
   8.8.1 Migration and load balancing 56
8.9 Conclusions 56

9 Porting OSE to TILEPro64 59
9.1 Previous state of the port 59
   9.1.1 Target interface 59
   9.1.2 Board support package 59
9.2 Porting tasks 60
9.3 Implemented tasks 60
9.4 Conclusions 61

10 Conclusions and future work 63
10.1 Conclusions ................................................. 63
10.1.1 Q1: What power management techniques should be con-
sidered by OSE on a many-core system and TILEPro64
respectively? ................................................. 63
10.2 Future work .................................................... 64
10.2.1 Power management ....................................... 64
10.2.2 Porting OSE to the TILEPro64 ......................... 65

Bibliography 67

List of Figures

2.1 Leakage currents in a CMOS transistor ......................... 6
2.2 Performance relative to power with leakage variation .......... 8
2.3 Processor frequency and transistor count contribution to the total
power consumption. ......................................... 8
2.4 Performance relative to power, k = 0.3 ....................... 10
3.1 Torus network .............................................. 13
3.2 Mesh and Torus performance comparison ...................... 14
3.3 FF, NN and CE mapping on a 4x4 2D-mesh NoC ............... 16
3.4 Result of the task mapping strategies in section 3.3.2 ........... 17
3.5 Multi-application mapping of three applications and their respec-
tive tasks .................................................. 18
4.1 TILEPro64 architecture overview ............................ 22
4.2 Cache latencies for the TILEPro64 .......................... 23
4.3 Overview of a tile .......................................... 24
4.4 Tile-to-tile communication over the UDN Network ............ 26
5.1 Rate monotonic scheduling .................................. 33
5.2 EDF scheduling .......................................... 34
5.3 Dhall’s effect in global EDF scheduling ...................... 35
5.4 Avoiding Dhall’s effect by considering utilization ............ 36
5.5 Semi-partitioned scheduling with the EDF-WM algorithm .... 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Global scheduling with Greedy Slack Reclamation</td>
<td>41</td>
</tr>
<tr>
<td>6.2</td>
<td>The GSSR (Global scheduling with Shared Slack Reclamation) algorithm</td>
<td>41</td>
</tr>
<tr>
<td>6.3</td>
<td>Slack reclaiming EDF</td>
<td>43</td>
</tr>
<tr>
<td>6.4</td>
<td>Slack reclaiming EDF with procrastination scheduling</td>
<td>43</td>
</tr>
<tr>
<td>6.5</td>
<td>Global scheduling with DVFS and leakage awareness. System design proposed in [16].</td>
<td>44</td>
</tr>
<tr>
<td>7.1</td>
<td>Task communication graph</td>
<td>48</td>
</tr>
<tr>
<td>7.2</td>
<td>Task mappings for the tasks in figure 7.1</td>
<td>49</td>
</tr>
<tr>
<td>7.3</td>
<td>Result of task mappings from figure 7.2</td>
<td>49</td>
</tr>
<tr>
<td>8.1</td>
<td>OSE kernel components</td>
<td>52</td>
</tr>
<tr>
<td>8.2</td>
<td>OSE priority bands</td>
<td>53</td>
</tr>
<tr>
<td>8.3</td>
<td>OSE architecture</td>
<td>55</td>
</tr>
<tr>
<td>9.1</td>
<td>OSE driver implementation structure</td>
<td>61</td>
</tr>
</tbody>
</table>
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMOS</td>
<td>Complementary metal–oxide–semiconductor</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Voltage and Frequency Scaling</td>
</tr>
<tr>
<td>EDF</td>
<td>Earliest deadline first</td>
</tr>
<tr>
<td>NoC</td>
<td>Network-on-Chip</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>RM</td>
<td>Rate Monotonic</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-time Operating System</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-case Execution Time</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

This thesis has been conducted at XDIN AB (former Enea Services AB) in collaboration with the Royal Institute of Technology, KTH. The thesis is part of the MANY \(^1\) project hosted by ITEA2 \(^2\). The mitigation from single-core processors to multi-core is a well rooted trend that can be seen even in hand-held consumer devices today. A look at the TOP500 \([4]\) list of the world’s most powerful supercomputers show large nodes of multi-core systems with peak power of up to 10 megawatts. These systems are today still dominated by relatively standard processors from Intel and AMD operating at around 2-3 GHz per core. Multi-processor techniques have helped such systems to achieve huge computation power. However computation demand continues to rise, together with the increasing cost of powering large data centres around the world.

Many-core can be considered the natural next step in this development, increasing the number of cores further, while being able to lower the operation frequency of each core. Designing for low dynamic power has become of increasing importance. At the same time new challenges arise in terms of hardware and software design.

\(^1\)Many-core Programming and Resource Management for High-Performance Embedded Systems
\(^2\)Information Technology for European Advancement
CHAPTER 1. INTRODUCTION

1.2 Problem statement

In a previous thesis project [39], work started on porting Enea OSE \(^3\) to the Tilera \(^4\) TILEPro64 platform, but further work is required for a complete port. This operating system is the desired target for this project, and contributing to the porting process has been one of the tasks of this thesis. Once a software platform is achieved on the target hardware, it is desired to develop tools for performance and low power. Another aim of the thesis is to lay a foundation for implementations of multi-core power management software, through a theoretical study.

1.2.1 Power management in many-core

Given a hardware platform, power efficiency on the OS level can be improved by taking into account underlying hardware. A study is made on the possibilities of power savings in many-core systems and TILEPro64 specifically.

On a many-core platform such as the TILEPro64, controlling core load becomes interesting. A traditional approach from an operating system would be to spread computational load among available cores, to avoid a bottle-neck from a specific core. However at low loads this may prove counter-productive, resulting in high re-scheduling overhead of tasks, or keeping unnecessary many cores active. The TILEPro64 platform implements a "nap" instruction usable by software for putting a processor core in low power sleep mode.

- \textit{Q1: What power management techniques could be considered by an RTOS on a many-core system and the TILEPro64 respectively?}

1.2.2 Porting OSE to the TILEPro64

Porting OSE to a new hardware platform consists of developing code for the hardware dependent parts of the operating system. To achieve this, a good understanding of both OSE and the target platform was required, as well as insight in the already existing code and reference ports.

---

\(^3\)Enea Operating System Embedded  
\(^4\)Tilera Corporation
1.3 Method

This thesis can be divided into two phases of 10 weeks each. During the first 10 weeks a theoretical study was made. These weeks also included a study of OSE and TILEPro64 documentation, and getting familiar with the developing environment for OSE. A variety of power management techniques for many-core systems was studied and analysed for implementation on TILEPro64 and OSE.

Another 10 weeks has been dedicated to working on the port of OSE to TILEPro64. While this thesis is individual work, it should be mentioned that another thesis project [34] has been working on separate parts of the port in parallel. To execute OSE on the target platform should be seen as a prerequisite for implementing power management techniques as well as a platform for future thesis projects. It was however not expected that a complete multi-core version of OSE should be achieved at the end of this thesis project. A Linux release offered by Tilera was used to demonstrate a concept concluded by the theoretical study.

1.4 Delimitations

The time limit for this study was 20 weeks in which the literature study, implementation, report and presentation was completed.
Chapter 2

Power in multi-core

Fred Pollack, a lead engineer at Intel found that over a number of Intel architectures, starting in 1986, the performance of every subsequent architecture increased only with the square root of the power or silicon area. This relationship is commonly referred to as Pollack’s Rule [11].

Moore’s law [33], formulated some 40 years ago, states that the number of transistors that can be placed in an integrated circuit will double every two years. The law is commonly interpreted as the performance of a chip will double every 18 months. Co-existing with Pollack’s rule, processors have indeed been able to steadily increase their performance. While Moore’s law still holds, to keep doubling the performance, the current interpretation of the law is rather that the number of cores will double every 18 month.

2.1 Processor power model

Power consumption in the processor can roughly be divided into two components; static power dissipation and dynamic power dissipation.

CMOS technology mainly consumes power as a transistor switches its state, and is therefore relatively power efficient when idle. But as technology has been able to pack billions of gates in a single chip, both power components have become a major design parameter.

2.1.1 Dynamic power dissipation

CMOS circuits dissipate power when switching states as a capacitive load, $C_{\text{load}}$, needs to be charged. In a chip, the total number of gates switching at a certain clock cycle varies, giving the activity factor, $\alpha$. The dynamic portion of power dissipation is:
CHAPTER 2. POWER IN MULTI-CORE

\[ P_{\text{dynamic}} = \alpha \cdot C_{\text{load}} \cdot V_{\text{supply}}^2 \cdot f_{\text{processor}} \]  

(2.1)

### 2.1.2 Static power dissipation

Even when a CMOS transistor is not switching its state, it is still connected to the supply voltage, \( V_{\text{supply}} \) and related with a certain power dissipation due to a undesired current, \( I_{\text{leakage}} \) flowing through the transistor. The static power dissipation is:

\[ P_{\text{static}} = V_{\text{supply}} \cdot I_{\text{leakage}} \]  

(2.2)

\( I_{\text{leakage}} \) is the result of several physical aspects of the CMOS technology (Figure 2.1), where two of them are more often referred to on the topic of processor power scalability. These are sub-threshold leakage and gate oxide tunnelling.

![Figure 2.1. Leakage currents in a CMOS transistor](image)

**Sub-threshold leakage**

A model for the sub-threshold current [12] can be simplified to show the parameters critical to the leakage current.

\[ I_{\text{leakage}} = A_1 W e^{-V_{\text{threshold}}/nV_0} \left( 1 - e^{-V_{\text{supply}}/V_0} \right) \]  

(2.3)

\( A_1 \) and \( n \) are experimentally derived and \( W \) is the transistor gate width. \( V_0 \) is the thermal voltage that will increase linearly as temperature rises.

The threshold leakage current can be reduced by increasing \( V_{\text{threshold}} \); however the threshold voltage is the potential barrier the gate must reach to switch its state. Increasing it has a negative effect on the speed at which the circuit can operate. The same arguments holds for lowering the supply voltage.
2.1. PROCESSOR POWER MODEL

Gate oxide tunnelling

As the CMOS process technology shrinks, the transistor gate oxide thickness must shrink proportionally, and is around 1 nm for 90 nm technology. When the oxide layer becomes that thin, a larger amount of current will start tunnelling through it. A simplified relation [12] for leakage current through gate tunnelling shows a strong relation to oxide thickness, $T_{oxide}$.

$$I_{oxide} = A_2 W \left(\frac{V_{supply}}{T_{oxide}}\right)^2 e^{\left(-mT_{oxide}/V_{supply}\right)}$$ (2.4)

$A_2$ and $m$ are experimentally derived. Silicon dioxide has been a popular insulator, but it cannot keep up with the small sizes of today’s CMOS, why industry has started looking at new materials to keep gate tunnelling leakage under control [1].

Considering leakage current in parallel computing

Assuming an application that can be parallelized to all available cores for 70% of its execution time, that is $P = 0.7$ in Eq. 2.6. The core idle power consumption, $k$ is the power dissipated through leakage current. While a larger amount of cores is likely to operate at lower supply voltages, and therefore drawing less leakage current, it is worth noting that un-utilized core capacity due to lack of parallelization induces a larger waste of leakage current totalling from all cores. This is illustrated by varying $k$ in Eq. 2.6 (Figure 2.2). The values for $k$ are optimistic. In fact, static power dissipation of processors today is close to 50% of the total power??.

This fact also concludes the static power portion of the so called dark silicon problem??, where transistors will become so many on a single chip that it is no longer feasible to switch, or even power all of them at the same time. Clock and power gating techniques attempting to cope with this problem is described in Section 2.2.

2.1.3 The single-core power issue

One method to increase the performance of the processor is a more complex or larger processor architecture. This increases the gate count, but is still a possibility as Moore’s law still holds. The other method is to increase the operating frequency, which at the same requires a higher supply voltage. Due to excessive heat dissipation, there is a practical limit in processor frequency where it is not feasible to increase it further. This is the main reason we saw a halt to increasing processor frequencies around 2004[41]. Figure 2.3 illustrates the two methods and their impact on power.
CHAPTER 2. POWER IN MULTI-CORE

2.2 Managing processor power in hardware

2.2.1 Gating

When a hardware component (e.g., processor, core or any architectural feature) is not needed, two major techniques can be applied to save power.
2.3. MULTI-CORE SCALABILITY CONCERNS

Clock gating

Clock gating aims to reduce the switching, or dynamic power dissipation by disconnecting parts of the chip logic not currently in use, from the clock source. Most modern processors implement clock gating to some extent, to cope with the power issues discussed in this chapter. Through clock gating, the activity factor $\alpha$ can be decreased (Eq. 2.1). The overhead of waking up the component is generally lower than with power gating, making it a useful technique in practice.

Power gating

Power gating disconnects the supply voltage of a component, completely eliminating current draw, therefore also the static power dissipation. This offers greater savings compared to clock gating. However, the overhead for waking up from a turned-off state is more costly, increasing the threshold idle time required for deciding to apply power gating.

2.3 Multi-core scalability concerns

In section 2.1 we saw the power problem for a single core processor. The multi-core design philosophy has been able to reduce the largest contributor to power consumption, which is the processor frequency (Figure 2.3). There are however considerations that has to be made for multi-core processors as well.

2.3.1 Augmenting Amdahl’s law

Stated by Gene Amdahl in 1967, Amdahl’s law [6] defines the limit of the theoretically achievable speed-up by parallelization of an application as follows:

$$Speedup = \frac{1}{(1 - P)} + \frac{P}{N}$$  \hspace{1cm} (2.5)

where $N$ is the number of processors, and $P$ the fraction of computation that can be made parallel.

To model the power consumption for a multi-core system [45], $k$ is introduced to represent the fraction of power each core consumes in its idle state. The parallel and sequential part of computation will be $(1 - P)$ and $\frac{P}{N}$, respectively. A core executing the sequential part of computation is defined to consume a power of 1. Thus the total processor consumption at this time is
1 + (N - 1)k. During parallel computation the power consumed is N. Adding this model to Amdahl’s law can thus be simplified into the following relation:

\[
\frac{\text{Speedup}}{\text{Power}} = \frac{1}{1 + (N - 1)k(1 - P)}
\]

(2.6)

Using this relation, as illustrated in figure 2.4, by utilizing each core closer to its maximum performance power scalability issues in multi-core can be set back to some extent.

![Figure 2.4. Performance relative to power, k = 0.3](image)

### 2.3.2 The KILL rule

The KILL (Kill If Less than Linear) rule [5] was introduced as a rule of thumb for multi-core hardware design by the MIT\(^1\) research group that was involved in designing the concept behind the RAW micro-processor [40], later to be commercialized by Tilera and used in their TILE architecture, which is described in chapter 4. The KILL rule states:

\'A resource in a core must be increased in area only if the core’s performance improvement is at least proportional to the core’s area increase. Put another way, increase resource size only if for every\'

\(^1\)Massachusetts Institute of Technology
2.4. CONCLUSIONS

1% increase in core area there is at least a 1% increase in core performance.

In other words, to make most efficient use of chip area, any architectural feature should scale linearly with the area it requires. Should it not, using that area for another core may in fact be a better choice.

2.4 Conclusions

The purpose of this chapter is bringing some insight into the multi-core trend, and the importance of power consumption both in single- and multi-core. It is definitely common knowledge that multi-core computers are needed to continue to raise the computation performance, as well as coping with energy constraints. It is however important to note that this performance does not come for free, but is highly dependent on how well applications can be parallelized. Augmenting Amdahl’s law to a power consumption perspective shows that power dissipated by cores in their idle state must be considered as core count increases.

It is therefore reason to investigate power management methods concerning systems with more than one core, how well they scale as cores increase, and to what extent such methods are hardware dependent.
Chapter 3

Network-on-chip

With multiple cores on a single chip, an interconnection architecture is needed. Historically, a shared bus has commonly been used, and still is, also for multi-core processors. Buses favour from simple and inexpensive implementation. A single line of communication can however easily get congested as nodes (cores, memory, I/O devices etc.) connected to the bus increases. Lately, the NoC has proven to be a promising solution [2] [42] to solve the large amount of nodes that needs to communicate.

3.1 Network-on-chip architecture

A NoC is firstly defined by its topology. The topology determines to what nodes, each node is connected. In a 2-dimensional mesh network, each node is connected to its neighbour, except at the network boundaries. The torus network (Figure 3.1) also connects the boundary nodes to those of the opposite side, improving bandwidth, and lowers the hop count between some nodes.

Figure 3.1. Torus network
CHAPTER 3. NETWORK-ON-CHIP

The torus and mesh network are attractive partly because their symmetrical design fits well with chip packaging constraints. The Tilera company noted that even though a torus is fully implementable on a 2D chip, the cost of extra wire length and wire congestion increases by a factor of approximately two, compared to mesh [44].

The performance of a NoC is characterized by its bandwidth, latency and path diversity. Figure 3.2 shows two performance metrics of the mesh and torus topology [13].

<table>
<thead>
<tr>
<th>Topology</th>
<th>Bandwidth</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k \times n$ Mesh</td>
<td>$2k^{n-1}$</td>
<td>$\begin{cases} \frac{nk}{n} \div \frac{k}{3} - \frac{1}{3k} &amp; k \text{ even} \ \frac{nk}{n} \div \frac{k}{4} - \frac{1}{4k} &amp; k \text{ odd} \end{cases}$</td>
</tr>
<tr>
<td>$k \times n$ Torus</td>
<td>$4k^{n-1}$</td>
<td>$\begin{cases} \frac{nk}{n} \div \frac{k}{3} - \frac{1}{3k} &amp; k \text{ even} \ \frac{nk}{n} \div \frac{k}{4} - \frac{1}{4k} &amp; k \text{ odd} \end{cases}$</td>
</tr>
</tbody>
</table>

Figure 3.2. Mesh and Torus performance comparison

Path diversity the number of routes that exists between two network nodes. If more routes exists, which is true for the torus compared to the mesh, adaptive routing techniques can reduce network congestion. Adaptive routing may use several routes between the same two nodes, should a single path be too congested. Oblivious routing does not adapt to the current traffic, and typically uses a general algorithm for traversing the network for all nodes. Therefore, adaptive routing benefits greater from a network with high path diversity, but the complexity of the routing algorithm is higher.

3.2 Network-on-chip power management

Data traversing over any computer network is not free. With distance travelled through wires comes higher energy required due to wire resistance and capacitance. Although NoC’s benefits from short distances between two neighbouring nodes, networks are becoming larger, consuming more power. The 8x8 mesh network in the RAW processor in fact consumes 36% of the total chip power, with each router in the network nodes dissipating 40% of the total node power, which also houses the processor core and local caches [43]. This fact requires efficient hardware solutions. But designing software considering the network may also prove beneficial, as will be seen in this chapter.
3.3. NETWORK-ON-CHIP TASK MAPPING

3.2.1 Power model

A great amount of research in power management in NoC’s is based around a power model introduced by Ye et al [47]. In it, the energy required for transferring one bit of data through a network node (router) is broken down into three components, giving the following equation:

\[ E_{bit} = E_{S_{bit}} + E_{B_{bit}} + E_{W_{bit}} \]  \hspace{1cm} (3.1)

where \( E_{S_{bit}} \), \( E_{B_{bit}} \) are the energy consumed by the router and internal buffers, respectively. \( E_{W_{bit}} \) is the energy lost in the interconnection wires. Each component is hardware dependent and assumes a single router on the chip. The model can be extended to fit the mesh type NoC, which will be the topology focused on in this chapter.

\( E_{L_{bit}} \) is introduced to represent the link (wire) between two network nodes [18]. Internal wire lengths and buffering is found to be negligible compared to \( E_{L_{bit}} \) which is typically in the order of mm. Eq. 3.1 can therefore be reduced to:

\[ E_{bit} = E_{S_{bit}} + E_{L_{bit}} \]  \hspace{1cm} (3.2)

For any network with identical nodes, a general model for traffic traversing one hop in the network can thus be modelled as:

\[ E_{bit} = n_{hops}E_{S_{bit}} + (n_{hops} - 1)E_{L_{bit}} \]  \hspace{1cm} (3.3)

3.3 Network-on-Chip task mapping

Task mapping is the concept of mapping a set of tasks onto specific nodes in a network, where they will execute. Task mapping should not be confused with task assignment (Section 5.1), which is the scheduling problem where schedulability of all tasks across several resources is the main concern. Task mapping could however be performed dynamically [21] [19] and incorporated in a task scheduler. In these works, the mapping constraint is energy, based on the energy model in section 3.2.1. Due to the complexity of the mapping itself, a more common approach is performing a static mapping [17] [20] [46] on a longer time-scale, perhaps as long as the system is running.

3.3.1 Task model

All the research mentioned above in this section uses the same concept for task modelling. A task is defined as an element that must run on one spe-
cific core and not be divided further. Directed graphs are used to represent communication between two tasks, and the weight is the traffic load for that communication. $G = \langle T, C \rangle$ is the directed graph, where $T = \{t_1, t_2, ..., t_n\}$ is a set of tasks, and $C = \{(t_i, t_j, w_{ij})\}$ denotes communication between two tasks. $w_{ij}$ is the total traffic sent from $t_i$ to $t_j$. This task model is sufficient for the entirety of this chapter.

3.3.2 Communication energy mapping

Hu et al [21] proposes a dynamic task scheduling approach to reduce power consumption through task mapping. A similar power model to that in 3.2.1. is used, which is simplified to the argument that a lower communication distance leads to a lower power dissipation. Two energy oblivious mapping algorithms are compared with the proposed energy aware mapping algorithm.

First-fit mapping

First-fit mapping (FF) is the simplest mapping method. Whenever a task in the system needs to be mapped to a core, the algorithm looks for a free core in the order top left (0,0) to bottom right, traversing the X direction first (Figure 3.3a).

Nearest neighbour mapping

Nearest neighbour (NN) maps the first task to core (0,0), as FF would. After this, each task is mapped to achieve the shortest path to the previously mapped task (Figure 3.3a).
3.3. NETWORK-ON-CHIP TASK MAPPING

Communication energy mapping

Communication energy (CE) is a best-effort to mapping tasks with high communication as close as possible. The strategy is to map tasks in decreasing order of their communication with other tasks. The details of the algorithm is as follows:

1. Mark all tasks in $G = \langle T, C \rangle$ as white.

2. Traverse $G$ to find the task $t_i$ with the highest communication energy $w_{ij}$ with another task. Map $t_i$ to core (0,0) and mark it black.

3. Find a white task $W$, with the highest communication energy with a black task, $B$.

4. Find the core with the minimum path distance to $B$, and map $W$ to it.

5. Repeat step 3 and 4 until all tasks are mapped.

Four example tasks, $C = \{(A_1, A_4, 100), (A_2, A_3, 100), (A_1, A_3, 50), (A_2, A_4, 50)\}$ are used to illustrate the result of the different mappings (Figure 3.3).

CE mapping is dependent on knowing the communication energy $w_{ij}$ beforehand. As this may not be the case in a real system, for the algorithm to work dynamically, communication data must be retrieved at run-time of each task. The strategy is to first conduct NN mapping, and after a desired amount of communication data has been collected, the mapping is changed to CE.

Results

The relative performance between the different mappings is calculated as the sum of the communication energy times the hop count (Figure 3.4 for all communicating tasks).

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Calculation</th>
<th>Communication Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>$100x3+100x1+50x2+50x2$</td>
<td>600</td>
</tr>
<tr>
<td>NN</td>
<td>$100x2+100x2+50x1+50x1$</td>
<td>500</td>
</tr>
<tr>
<td>CE</td>
<td>$100x1+100x1+50x1+50x1$</td>
<td>300</td>
</tr>
</tbody>
</table>

Figure 3.4. Result of the task mapping strategies in section 3.3.2
CHAPTER 3. NETWORK-ON-CHIP

3.3.3 Multi-application mapping

Similarly to the communication energy mapping in section 3.3.2, Yang et al [46] maps tasks based in order of communication volume. But instead of mapping all system tasks individually, tasks are modelled to belong to an application. Applications and tasks are mapped separately in two stages. First applications are mapped to regions of the NoC, varying in size depending on the size of the application. After all applications are mapped, the tasks of an application is mapped within the region. The purpose of the two-level mapping approach is to achieve an optimized task mapping for all applications. Communication volumes between tasks in an applications is assumed to be known a priori.

![Diagram of multi-application mapping](image)

Figure 3.5. Multi-application mapping of three applications and their respective tasks

Application mapping

Figure 3.5b illustrates the proposed strategy for mapping applications. In 3.5a, tasks of applications have been assigned cores randomly by an operating system oblivious to network locality and which tasks belongs to which application. In 3.5b, each application is encapsulated by its minimum rectangle and tasks mapped to the cores within the region. Unused cores are divided into non overlapping rectangles, that will be divided further by other applications.

Task mapping

The task mapping within an application is similar to that of CE mapping in 3.3.2. Each application task-set is mapped independently in turns. The highest communicating task is placed on the center core of the application.
3.4. CONCLUSIONS

region, after which tasks are mapped in decreasing order of communication

volume.

3.4 Conclusions

3.4.1 Network-on-Chip

As the number of cores on a chip will continue to increase, the on-chip inter-
connect is quickly becoming the bottleneck in terms of latency, throughput and
power consumption. For power consumption, data showed that the network
already consumes a large amount of power. This motivates new techniques
not only in hardware, but also software.

3.4.2 Task mapping

To cope with the latency and power scalability issues in NoC’s, task mapping
seems to be a good software solution. The authors of [46] claims that compared
to a random task placement, their solution achieves almost a 60% improvement
both in terms of network latency and power consumption. In [21], simulations
of the proposed solution achieved around 20% improvement in performance
and power consumption, compared to oblivious task mappings. It can be
concluded that performance and power benefits almost equally.

None of the research studied during this thesis considered real-time con-
straints or co-existing with the OS scheduler. It seems to remain an open
issue how well task mapping performs as an online solution for a general sys-
tem. A static mapping of applications and processes at boot time of a system
could be a good start for implementation.
Chapter 4

Tilera TILEPro64

The TILE architecture has its origins in the RAW research processor developed at MIT\(^1\) and later commercialized by Tilera, founded by the original research group. Tilera’s many-core processor TILEPro64 is the target platform for the porting process of this thesis (Chapter 9), why it’s dedicated this separate chapter. Also, a demo of task mapping on the platform is described in chapter 7. The information is this chapter is derived from Tilera’s documentation [3].

4.1 Architecture overview

The TILEPro64 is a 64-core processor with a 8x8 2D-mesh NoC architecture (Figure 4.1), where the network nodes are named tiles. Each tile consists of a general-purpose 32-bit VLIW (Very Long Instruction Word) processor, a cache engine, and a non-blocking switch engine interconnecting the tiles to the on-chip network. Each VLIW bundle of instructions is capable of encoding two or three instructions. Three execution pipelines are designed asymmetrically, each able to execute a different subset of the instruction set. Maximum performance is specified as 443 BOPS (Billion Operations Per Second). The cache engine for each tile has 8 KB data and 16 KB instruction L1 cache, and a unified 64 KB L2 cache. Main memory is organized as four 64-bit DDR2 controllers, each accessible from any tile.

4.2 Memory architecture

The four memory controllers on the TILEPro64 are symmetrically located on the edges of the interconnection network, all being accessible from any tile.

---

\(^1\)Massachusetts Institute of Technology
4.2.1 Striped memory

The striped memory mode can be enabled at boot time of the TILEPro64, and overrides the default mapping of physical memory pages onto the four memory controllers. In striped memory mode, a physical page is "striped" evenly across the four memory controllers at a 8 KB granularity. This means that access to a physical memory page from a tile is carried out pseudo-simultaneously to all four memory controllers, thus balancing the load of all controllers, as well as the memory traffic in the interconnection network.

4.3 Cache architecture

By default, the hardware provides full cache-coherency, and any read of an address in the main memory will return the most recent write to that address. Modes for disabling cache coherency are available, as well as turning data caching of completely. The L2 cache of a tile can be accessed from any other tile, making up a distributed L3 cache. The access times of the caches in
4.4. INTERCONNECTION NETWORK - IMESH

TILEPro64 can be seen in figure 4.2. The execution pipeline implements out-of-order execution and does not stall on cache misses until the data is needed by an instruction.

<table>
<thead>
<tr>
<th>Latency (load-to-use)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1D hit</td>
</tr>
<tr>
<td>L2 hit (local)</td>
</tr>
<tr>
<td>L2 hit (remote)</td>
</tr>
<tr>
<td>L2 miss to memory</td>
</tr>
</tbody>
</table>

Figure 4.2. Cache latencies for the TILEPro64

4.3.1 Cache-as-RAM

The Cache-as-RAM mode allows the application to use the address space of the local L2 cache as general memory storage, before the main memory controllers have been configured. Any data access within the L2 cache address space will use the cache. If main memory is available, it will be used for addresses not valid in the L2 cache.

4.3.2 Hash-for-home

By default, the MDE\(^2\) provided by Tilera uses hash-for-home for all except stack data. Software may define a set of tiles where the hashing function spreads data among the L2 cache of those tiles, so called distributed L3 caching. Performance can be improved for multi-threaded applications working on a shared, larger data set, not able to fit in a single local L2 cache.

Under this mode, a read operation from tile A first checks its local caches for a specific cache-line. On a miss, it is fetched from the tile B where the cache-line has been placed by the hashing function, called homed. Finally the local L1 and L2 caches of A is updated with the cache-line. In a write operation of A where the cache-line is homed at B, the data is directly written to B’s L2 cache, and B is responsible for invalidating any other tile with a copy of the cache-line, and finally sending a confirmation back to A.

4.4 Interconnection network - iMesh

The TILEPro64 uses a 2D-mesh topology for its interconnection network, categorizing the platform as a NoC. The switch engine controls six independent

\(^2\)Multi-core Development Environment
physical networks through crossbar switching. Any input port can arbitrate for any output port, excluding itself. Each switch is connected to its four neighbours except for nodes located on the mesh boundaries, and one connection to the local processor.

Another physical connection exists between the switch engine and the cache engine, relieving the processor from cache-coherency handling. Figure 4.3 illustrates the architecture of a tile in the TILEPro64.

The five dynamic networks are the user dynamic network (UDN), tile dynamic network (TDN), memory dynamic network (MDN), coherence dynamic network (CDN) and I/O dynamic network (IDN). The dynamic networks use a packet based interface, where a packet header contains the coordinates of the destination node.

Transferring data between two neighbouring tiles, or one network hop, has a delay of 1-2 cycles. The UDN, IDN and STN networks are tightly integrated with the processor pipeline, allowing any instruction to read or write to these networks.

![Figure 4.3. Overview of a tile](image)
4.4. INTERCONNECTION NETWORK - IMESH

4.4.1 The six networks

UDN The user dynamic network is the only user-visible of the dynamic networks. While all dynamic networks provides deadlock-free routing, it is the responsibility of the user to avoid deadlocks by circular dependencies in sending and receiving data between tiles.

IDN The I/O dynamic network is software visible and handles communication between tiles and I/O devices and between I/O devices and memory. A protection mechanism limits access to the IDN to OS-level code.

MDN The memory dynamic network is used for memory data transfer (resulting from loads, stores, prefetches, cache misses or DMA (Direct Memory Access)) between tiles themselves, and between tiles and external memory. Only the cache engine has a direct hardware connection to the MDN.

CDN The coherence dynamic network (CDN) is used to carry cache-coherence invalidate messages.

TDN The tile Dynamic Network (TDN) use is similar to the MDN and supports memory data transfer between tiles. Only the cache engine has a direct hardware connection to the TDN.

STN The static network does not use the same dynamic routing scheme as the dynamic networks. Instead a static path is set up between all tiles, providing efficient transport of data streams. A packet based communication is not used. The routing paths can be configured in special purpose registers.

4.4.2 Tile communication

The Tile processor supports two ways of communicating between tiles when writing parallel applications.

Shared memory

A traditional shared memory model can be considered the default way of communication. Each process may access the shared data in memory at any time, and the developer must ensure data synchronization through access ordering or mutual exclusion.
UDN network

The UDN network is dedicated to the user to utilize. The UDN network offers point-to-point packet based communication without having to go through the main memory. In fact the cache system may also be bypassed by sending data words directly between the local registers of two tiles, improving performance further. Packets is sent between tiles, and processes that wishes to communicate over the UDN must be bound to specific tiles. This method of communication is especially suitable for producer-consumer applications. Figure 4.4 shows an example of several applications with their communication dependencies, and a possible mapping of each process onto different tiles.

![Tile-to-tile communication over the UDN Network](image)

**Figure 4.4.** Tile-to-tile communication over the UDN Network

### 4.4.3 Network hardwall

A hardwall mechanism allows for a programmable protection bit for each output port of the UDN and IDN switch. No data can be sent from a protected port, and an interrupt is triggered if attempting to send a packet to the port. This can be used to prevent unwanted communication between user applications running on adjacent tiles. It may also not desirable that the user is able to communicate directly with I/O devices through the IDN network.

### 4.4.4 Network routing

When a packet is sent on one of the dynamic networks, its header contains the X and Y destination coordinates, that are compared with those of the sending node, before making a routing decision. A shortest-path dimensional ordered routing scheme is used, meaning a packet first traverses one direction, then the other to reach its destination. The default order in TILEPro64 is the X direction first, or X-Y routing, but the order may be swapped. The routing
4.5 HARDWARE SUPPORT FOR POWER MANAGEMENT

algorithm is therefore deterministic in that the same path will always be used between two specific tiles. It has the advantage of being easy to implement with a low latency, when the network is not congested.

4.5 Hardware support for power management

The TILEPro64 is designed heavily around low power consumption, which is mainly achieved by running the cores at the relatively low frequency 700 MHz with a 1 V supply power. Also, extensive clock gating is implemented in the processor. An assembly "nap" instruction usable by software, puts a tile into a low-power idle mode until a user-selectable external event, such as an interrupt or a packet arrives. The "nap" functionality does not completely turn off the core, why a small static power dissipation from the core will remain.

Measurements have showed that the power consumption was 14 W with 63 of the cores performing no work in the 'nap' state [38]. In the same work, maximum operating power under high load was measured to around 25 W. It can be assumed that the 11 W difference between the operating conditions mainly consists of the dynamic power dissipation of the cores. TILEPro64 implements no means of adjusting core frequency or voltage.

4.6 Conclusions

4.6.1 Power management

The "nap" instruction

While many modern processors offers hardware support for adjusting the voltage and frequency, this is not supported in the TILEPro64. One reason may be the chip area cost for such functionality. Another reason may be that the cores of the TILEPro64 already operates at a relatively low frequency (700 MHz), with a supply voltage of 1 V. In the work of [38], dynamic power used by all the 64 cores was measured to around 11 W. There may in fact not be much to earn from lowering the frequency and voltage further.

The 'nap' instruction was also evaluated in [38], by using a worker thread style application. Each core was assigned a worker thread, and when a thread had no work, it was deactivated to let the core be completely idle. This in itself led the greatest energy savings due to previous overhead from workers looking for work. Running the 'nap' instruction at this time led only to 4%
average further energy savings, as it can be seen as only a more aggressive idle state. Still, it should be further analysed how to utilize this instruction, and how to design software in general when the amount of needed cores varies.

Using the networks

Processes must be bound to a specific tile when using UDN for communication. Network delay between tiles is 1-2 cycles per hop in the network, and spatially short distance communication can improve performance. The largest improvement is made by only using the network, reducing accesses to the main memory. Power consumption is in fact reduced at the same time, as it is easily understood that power dissipation in wires are dependent on the wire resistance and capacitance, which increases with wire length. This was shown in section 3.2. It is therefore interesting exploring the task mapping techniques introduced in section 3.3 on the TILEPro64.
Chapter 5

Real-time multi-core scheduling

The process scheduler is a vital component of any computer system, and has allowed the single-core processor to execute multiple applications in pseudo-parallel, by switching between processes according to some principle of choice. Already in 1969, Liu [28] noted the increased difficulty of multi-core scheduling:

"The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors"

This chapter will introduce scheduling concepts when moving from uni-core to multi-core, in a real-time context. Priority driven scheduling is assumed throughout the chapter.

5.1 Scheduling taxonomy

Using the notation of [30], one of the problems that arise in multi-core scheduling is task assignment, or on which core each task should execute on. Once assigned, the following classifications is defined for what changes are allowed in the assignment.

1. *No migration*. Once a task has been assigned to a core, migration to other cores is not permitted.

2. *Task-level migration*. The jobs of a task may execute on different cores; however, each job can only execute on one core.

Further, scheduling algorithms where migration is permitted are referred to as *global*, and those where no migration is permitted, as *partitioned*. 

29
In the assumed priority driven scheduling of this chapter, priorities can be either *fixed* or *dynamic*. A fixed priority scheduling algorithm assigns the same priority to all jobs of a task, while a dynamic algorithm may assign the priorities of jobs of a task differently.

### 5.1.1 Preemptiveness

An operating system is said to be either *pre-emptive*, or *non-preemptive*. In preemptive scheduling, a task may be interrupted at any time, inducing a context switch, typically by a higher priority task. In a non-preemptive system, once a task has started its execution it will run to completion and cannot be preempted.

### 5.1.2 Online and offline scheduling

Intuitively, an *online* scheduling algorithm makes each scheduling decision as the system is running, without knowledge of the release of future tasks. Clearly, online scheduling is the only approach in any system where the workload is unpredictable, and may depend on external events. The biggest challenges exist in the research area of online algorithms, where the future uncertainty makes it difficult or impossible to achieve an optimal scheduling decision.

In the case of a real-time system where all execution patterns are known before starting the system, an *offline* scheduling algorithm can calculate a fixed schedule for a given task-set. Because the schedule is computed offline, the complexity of the scheduling algorithm is not important, while for online scheduling, the algorithm adds to the overhead of each invocation of the scheduler.

### 5.1.3 Task assignment

*Task assignment* is the problem that each task in the system must be assigned a core where it should execute. The task assignment algorithm must decide:

1. On which core the task should execute.
2. How many cores should be used for the assignment of all tasks.

1 and 2 can easily be translated into the NP-complete bin packing problem [30], why a heuristic is often used. The general bin packing problem is: A set
of objects, each with a certain volume must be packed into a finite number of bins of a volume $V$, such that the number of bins are minimized.

Task assignment may in addition need to handle data dependencies between tasks running on different cores, as well as precedence constraints for tasks.

### 5.1.4 Optimality

The definition of an *optimal* scheduling algorithm is best explained by introducing *feasibility*. A task-set is said be feasible on a system if there exist a scheduling algorithm that is able to schedule the task-set without missing any deadlines. A scheduling algorithm is said to be optimal if any task-set can be scheduled that is also feasible on the given system. For a single-core processor, optimal online scheduling algorithms exists, such as the commonly used EDF algorithm, which will be briefly explained in section 5.4.2. For multi-core scheduling however, it is known that no optimal online algorithm exists for a sporadic task-set [15]. Several global scheduling algorithms exists to produce an optimal schedule for periodic task-sets, such as the Pfair algorithm which will be described in 5.6.2.

### 5.2 Real-time systems

Systems are referred to as real-time when the correctness of the system depends not only on the logical result of an operation, but also on the time at which it is performed. A real-time operating system must therefore keep a correct notion of time, after which it can schedule tasks appropriately. A scheduling algorithm of a general real-time operating system must be preemptive, such that a task of low priority can not block a higher priority task from executing.

Deadlines of real-time systems can be categorized into soft and hard deadlines. A hard deadline must be met at all times and any timing violation can cause catastrophic consequences, imaginable by embedded systems in air planes or medical equipment. Soft deadlines should be met, but the consequence of not doing so is not catastrophic, and is usually related to the quality of service provided to the user.
5.3 Task model

Sometimes referred to as the Liu and Layland model, the periodic task model [29] characterizes a real-time task \( \tau_i \) by its:

- Relative deadline, \( D_i \)
- Worst-case execution time (WCET), \( C_i \)
- Period, \( T_i \)

The utilization \( u_i \) of a task is given by \( \frac{C_i}{T_i} \), a measure of processor capacity consumed by the task over the period. The model is interpretable also for aperiodic and sporadic task-sets.

5.3.1 Periodic, aperiodic and sporadic tasks

*Periodic* tasks are invoked once per its period, with a deadline typically within or at the end of the period. Periodic tasks could be core functionality of a real-time embedded system, with hard deadlines.

The arrival time of *aperiodic* tasks can not be known a priori. Aperiodic tasks are said to have soft deadlines or no deadlines, and will always be accepted by the scheduler and completed as soon as possible. The time from the release of an aperiodic task and when its allowed to execute is called the response time.

Aperiodic tasks with hard deadlines are called *sporadic* tasks. Because of the hard real-time constraints, the scheduler can only accept the task if no other deadlines in the schedule are violated. Otherwise the sporadic task must be rejected. Periodicity of aperiodic and sporadic tasks is not considered, and instead replaced by a minimum inter-arrival time.

5.3.2 Example task-set

The task model above can be written as \( T_i = (P_i, C_i, D_i) \) for a task. The following task-set will be used in this chapter to illustrate the various single-core and multi-core scheduling concepts.

- \( T_1 = (6, 3, 6) \)
- \( T_2 = (8, 3, 8) \)
- \( T_3 = (10, 1, 10) \)
5.4 Classic scheduling algorithms

Two scheduling algorithms in particular are worth a mention, because they are widely referenced in research, or work as a basis for further development of scheduling algorithms, such as multi-core or power-aware algorithms.

5.4.1 Rate monotonic scheduling

A well known fixed priority algorithm is the RM (Rate Monotonic) algorithm [30]. The rate at of which a task is invoked is the inverse of its period, and RM assigns the highest priority to the shortest period, and so on. The inventors showed that RM is optimal in the sense that no other fixed priority scheduling algorithm can schedule a task-set that cannot be scheduled by RM. The least upper bound of total processor utilization under the RM algorithm is given by Eq. 5.1.

\[
\sum_{i=0}^{n} u_i \leq n(2^{1/n} - 1)
\] (5.1)

For large values of \( n \), the least upper bound converges to 0.69, which means that any task-set of less utilization is guaranteed to be schedulable by RM. Task-sets \( \sum_{i=0}^{n} u_i > 0.69 \) are not guaranteed a feasible schedule, but it might exist. Our example task-set fails under the RM algorithm (Figure 5.1).

![Figure 5.1. Rate monotonic scheduling](image)

5.4.2 Earliest deadline first

Of the dynamic scheduling algorithms, EDF (Earliest Deadline First) is perhaps the most well known [29]. It is dynamic in the sense that the priority of tasks is assigned in order of their deadlines. Tasks with earlier deadlines will be executed at higher priorities, why \( T_1 \) is not able to preempt \( T_3 \) at time
CHAPTER 5. REAL-TIME MULTI-CORE SCHEDULING

instance 6 (Figure 5.2). Since EDF does not make any scheduling decision based on periodicity, it is equally applicable for both periodic and aperiodic tasks. The least upper bound for EDF is given by equation 5.2.

\[ \sum_{i=1}^{n} u_i \leq 1 \]  

(5.2)

Figure 5.2. EDF scheduling

5.5 Partitioned scheduling

Partitioned scheduling is characterized by a separate run-queue for each core. A task is initially assigned to a core, after which no migration of that task is allowed. Once assigned, each core’s task-set may be scheduled according to a single-core algorithm such as EDF or RM, why the advantages include simplicity and scalability. A drawback to a fully partitioned approach is utilization fragmentation. Several cores may not be fully utilized, however, no core has enough remaining capacity to schedule further tasks. In fact, for partitioned EDF scheduling, to guarantee deadlines the total task utilization in a system with \( m \) cores may not exceed \( (\beta m + 1)/(\beta + 1) \), where \( \beta = \lfloor 1/u \rfloor \) is the maximum number of tasks of utilization \( u_i \) [31]. For \( u = 1 \) and \( m \to \infty \) the worst-case utilization bound is only 50%.

5.6 Global scheduling

To allow for task migration between cores, a global run-queue has to keep track of all the tasks in the system, which opens for several advantages compared to partitioned scheduling. By migration to balance tasks among cores, preemptions in the system could be reduced. Further, when a task finishes execution before its WCET, remaining capacity can be utilized by any task in
5.6. GLOBAL SCHEDULING

the system waiting to execute, not just those on the same core. For large sys-
tems the overhead of manipulating a global run-queue, may however become
excessive.

5.6.1 Dhall’s effect

A problem with global multi-core scheduling was introduced in the seminal
paper of Dhall and Liu [14], why it came to be known as Dhall’s effect. Dhall’s
effect occurs when a task-set of relatively low processor utilization can not be
scheduled due to several smaller tasks blocking a larger. This may happen
when applying single-core deadline based priority assignment directly to multi-
core. Assuming EDF, which is in fact an optimal algorithm for one core, the
following task-set will be used to illustrate Dhall’s effect on a processor with
three cores.

- \( T_1 = (10, 2, 10) \)
- \( T_2 = (10, 2, 10) \)
- \( T_3 = (12, 10, 12) \)

\( T_3 \) has the later deadline and will under EDF be assigned the lowest pri-

ority. The generated schedule is shown in figure 5.3. Although the total
processor utilization, \( U \) (Eq. 5.3), is far less than the maximum capacity of 3
(three cores), \( T_3 \) fails to meet its deadline in the first cycle.

\[
U = u_{T_1} + u_{T_2} + u_{T_2} = \frac{2}{10} + \frac{2}{10} + \frac{10}{12} \approx 1.23
\] (5.3)

\[
\text{Figure 5.3. Dhall’s effect in global EDF scheduling}
\]
Avoiding Dhall’s effect

Dhall’s effect can intuitively be avoided by assigning tasks of high utilization a higher priority. As an example, in [32] tasks are divided into light or heavy, depending on their utilization. All heavy tasks have a higher utilization than light tasks, and light tasks are internally scheduled according to EDF. In figure 5.4, \( T_3 \) has been assigned a higher priority due to its high utilization, and the task-set becomes schedulable.

![Figure 5.4. Avoiding Dhall’s effect by considering utilization](image)

5.6.2 The Pfair algorithm

The Pfair (Proportionate fair) algorithm [10] is applicable to a periodic task-set and is known to be optimal such that any set of tasks is schedulable to meet all deadlines on \( m \) processors as long as equation 5.4 holds.

\[
\sum_{i=1}^{n} u_i \leq m \tag{5.4}
\]

The main idea of the Pfair algorithm is that each task is scheduled, or makes progress in its execution at an explicit rate, proportionate to its utilization.

The time line is broken down into equal length quanta or time slots, and tasks divided to execute in a number of time slots, the last of which is its deadline. At each invocation \( t \) of the scheduler, a task may be either ahead (tnegru) or behind (urgent) of its execution. Urgent tasks will be scheduled at time \( t \), while tnegru tasks will not be.

A uniform sub-division of tasks reduces fragmentation and allows for a full utilization of cores. A drawback of the Pfair algorithm is the high overhead induced by invoking the scheduler at each quanta, as well as frequent preemption and migration.
5.7 Semi-partitioned scheduling

To cope with the disadvantages of both the global and the partitioned scheduling approach, hybrids of the two have appeared, originally with the EDF-fm (fm denotes that a task is either fixed or migrating) algorithm [8].

Leontyev and Anderson [27] created a scheduling abstraction, the container, being a specified portion of the processing capacity of all cores in the system. Containers are organized hierarchically and may contain tasks or other containers. The approach can be thought of as clustering into a smaller number of faster cores, reducing the global queue length as well as fragmentation. The approach supports both hard and soft sporadic tasks, with a slight utilization loss incurred by ensuring hard deadlines. However, in systems with only soft real-time tasks, no utilization is lost.

5.7.1 The EDF-WM algorithm

Kato et al [26] presents a novel algorithm, EDF-WM (EDF with Window-constraint Migration), able to handle periodic and sporadic task-sets with arbitrary deadlines, aiming to reduce the number of context switches usually associated with global scheduling. The algorithm is characterized by only allowing a task to migrate to another core if there is not enough remaining capacity on any individual core. To reduce context switches, as well as better utilizing local caches, each task may only migrate between cores once each period.

By introducing these two migration constraints, Kato et al have addressed problems of prior algorithms such as EDF-SS [9] and EDDP [25], where the number of context switches may become prohibitive. Therefore, the algorithm will be briefly explained below:

- Each task is assigned to an individual core according to a first-fit heuristic, as long as it can be. These assignments are illustrated by the striped areas in figure 5.5.

- If a task cannot fit on any individual core, it is split in such a way that it attempts to fill the capacity of each core assigned a portion of the task in a first-fit manner, the case of $T_1$ in figure 5.5.

- Since the job of a task is not permitted to execute in parallel, a migratory task has its deadline split into the same number of windows as the cores it was migrated onto. Each portion of the task is then assumed to be
released at the beginning of its window, which is also the deadline of the previous portion of the task.

**Figure 5.5.** Semi-partitioned scheduling with the EDF-WM algorithm
Chapter 6

Power-aware multi-core scheduling

Adding power awareness to multi-core scheduling algorithms is dependant on both the hardware used as well as the chosen approach to scheduling in the system, global or partitioned. In the partitioned case, existing single-core techniques such as DVFS (Dynamic Voltage and Frequency Scaling) may be incorporated in the scheduling algorithm.

Global scheduling allows for actively distributing tasks among available cores, either to reduce the number of powered cores, or to balance work among all cores and apply more aggressive DVFS.

6.1 Dynamic voltage and frequency scaling

Modern processors often has hardware support for dynamically adjusting the processor frequency and supply voltage within a range. As we saw in section 2.1 this can greatly reduce power consumption. In multi-core, DVFS can further be categorized as follows:

Processor-wide

The processor shares the same voltage and frequency regulators for all cores. In this case, balancing the load evenly among all cores is usually preferred, so that it is possible to reduce the frequency while ensuring deadlines.

Per-core

Some processors allow each core to be set individually to certain frequencies and voltages. Research in the field, shows that this possibility is superior in terms of power savings compared to single regulators. Naturally, the complexity of the DVFS algorithm also grows.
6.1.1 The GRUB-PA algorithm

In a previous thesis project at Enea AB, a work-in-progress variant [7] of the power-aware GRUB-PA algorithm [35] was implemented. The algorithm is in the class of server based schedulers, why it’s able to handle periodic, aperiodic and sporadic tasks equally, meeting both hard and soft deadlines. Another advantage is the slack reclaiming ability of GRUB-PA, allowing it to apply further DVFS for tasks completing before their WCET.

The main properties of GRUB-PA are as follows:

- A server is assigned to every task, characterized by its maximum utilization and period. It keeps track of the deadline of a task, and a measure of how much of its maximum utilization has already been consumed, called virtual time.

- When a task is executing, the respective server increases the virtual time as the task’s fraction of the total current system utilization. This way unused processor utilization may be reclaimed and used to slow down the processor.

6.1.2 The GSSR algorithm

Zhu et al [48] describes GSSR (Global Scheduling with Shared Slack Reclamation), where reclaimed slack is distributed among cores and future tasks scheduled at a lower core frequency. It is a global scheduling algorithm, synchronising in critical sections through the shared memory. A frame-based real-time execution model is assumed, where a set of tasks is to execute in each frame, all of which are ready to run at the beginning of the frame and must complete before the end of the frame. The WCET of all tasks must be known beforehand, but the slack is reclaimed dynamically when a task finishes its execution. While the assumed task model is limited for a general system, this early work can be used to illustrate the concept of slack reclamation for multi-core systems. The authors shows that a purely greedy slack reclamation scheme may fail to meet the deadline, by assigning all slack to the next task in the run-queue (Figure 6.1), why the proposed shared slack extension is necessary (Figure 6.2). T₁ has a WCET of 10, but finishes execution in 4 time units. The length of the frame is 18, which is also the deadline of all tasks. GSSR is scalable to N cores, but simulations by the authors shows greatly reduced energy savings above 8 cores, due to lack of task parallelism.
6.2. LEAKAGE-AWARE SCHEDULING

Leakage current in a processor was described in 2.1.2, where it was shown that leakage may become an increasingly larger threat to processor design. By solving \( \frac{d(P_{dynamic} + P_{static})}{df} \), where \( f \) is the processor frequency, it can be seen that the static/dynamic power dissipation ratio is not linear, why research have proposed several algorithms where this ratio is taken into account. In [24], it is shown that for the Crusoe processor manufactured in 70 nm CMOS technology, static power dissipation dominates total power dissipation for voltage levels under 0.7 V. At this voltage the processor is able to operate at 1.26 GHz out of the maximum 3.1 GHz at 1 V supply voltage. In the work of [22] and [36] the same concept is denoted as the critical speed. In other words, for task-sets of high utilization a balanced assignment among cores if preferred. Under low loads, when cores are not utilized up to the critical speed factor, a balanced assignment is no longer power optimal.

By powering off a processor core, leakage currents can be reduced to a zero. Shutting down a core results in a certain shut-down and wake-up overhead, both in terms of time and power, why ultimately these scheduling techniques attempts to find, or create, a large enough idle time window to do so.
6.2.1 Shut-down overhead

While clock gating allows inactivation and activation of a core down to a cycle to cycle basis, power gating must be used with greater care. Even deadlines could be missed. Without power, a core loses its register contents and the data of any volatile memory connected by the same power supply, namely the local caches. Before shutting down a core, registers must be stored and dirty cache lines flushed to main memory. TLB’s and any processor architectural features such as branch history tables must be re-initialized, causing extra memory access or branch mispredictions at resumed execution. The work of Jejurikar et al [24] takes only cache misses into account and calculates a shut-down threshold of $2\text{ms}$, using the Crusoe processor with an idle consumption of $240\text{mW}$ and assuming a sleep state power of $50\mu\text{W}$.

6.2.2 Procrastination scheduling

Jejurikar and Gupta [23] makes the argument of the increasing leakage currents in processors. A single-core scheduling algorithm applicable on periodic task-sets is presented, where all tasks are able to meet their deadlines. Idle times in the task schedule is extended by merging with unused utilization, or slack time. Figure 6.3 and 6.4 illustrates how this can be achieved by rescheduling tasks at a later time. The task-set used is the same as that in 5.3.2, with the exception that $T_2$ always finishes executing before its WCET, creating an extra slack of 1 time unit. This time is to be used for either processor slowdown, or shut-down at a certain threshold. The proposed algorithm works as follows:

- When a job finishes execution its slack and priority is stored in a list called the free run-time list, keeping track of all slack in the system, sorted by the priorities.

- When a job starts execution, it is assigned a time budget which is its WCET scaled by the current processor slowdown factor. Any job is allowed to use its own time budget and the slack of any equal or lower priority, from the free run-time list.

- Any time when a task is not executing, time is consumed from the free run-time list.

- Procrastination for a job is limited by the total amount of slack available in the free run-time list, ensuring deadlines are met.
6.2. LEAKAGE-AWARE SCHEDULING

![Figure 6.3. Slack reclaiming EDF](image)

![Figure 6.4. Slack reclaiming EDF with procrastination scheduling](image)

### 6.2.3 Task migrating algorithms

To be able to migrate a task to another core, a global or semi-partitioned scheduling scheme is required, that controls all tasks. A large amount of research exists for power-aware task migration, where tasks can be assigned such to optimize the level of DVFS.

Extending DVFS-enabled task migrating scheduling to account for leakage have shown [36] [22] [16] that further energy savings can be achieved.

Seo et al [36] solves a processor power model using constants of a 70 nm CMOS technology and concludes a frequency threshold where $P_{\text{leakage}}$ is greater than $P_{\text{dynamic}}$, in this case 0.4 times the maximum performance. The authors denotes this factor the critical speed. Under the critical speed, at high load, load balancing and DVFS is used and power savings are achieved. At low loads above the critical speed however, the frequency is not scaled further down. Instead, an algorithm is proposed for attempting to maximize the utilization of all cores, therefore minimizing the number of cores with scheduled tasks. Freed cores will be put in a sleep state where $P_{\text{dynamic}}$ is zero and $P_{\text{leakage}}$ only a small fraction of $P_{\text{leakage}}$ in the active state.
CHAPTER 6. POWER-AWARE MULTI-CORE SCHEDULING

Fu and Wang [16] extends a global scheduling scheme, and assumes per-core DVFS. Each core monitors its utilization and applies DVFS accordingly. The monitor reports the utilization of the core to a processor task consolidation manager, attempting to merge task to fewer cores and shut down unused cores. Figure 6.5 shows the proposed system model. At each invocation of the task consolidation manager, an assignment of tasks among cores, and each core’s individual frequency is found while solving the following problem:

\[
\min \sum_{i=1}^{n} x_i(k)[P_{ind}^i + \alpha_i f_i(k)^\beta_i]
\]  

(6.1)

where \(x_i(k)\) represents the state of a core. If a core is powered on, \(x_i = 1\); otherwise \(x_i = 0\). \(P_{ind}^i\) is the static power consumption of a core and does not change with core utilization or frequency. \(\alpha_i f_i(k)^\beta_i\) is the dynamic power consumption of a core, where \(\alpha_i\) and \(\beta_i\) are system dependent parameters.

The problem of a global task assignment problem is analogous to the bin packing problem [30], and Fu and Wang analyses four heuristics and decides on a first-fit algorithm for its low overhead.

![Figure 6.5. Global scheduling with DVFS and leakage awareness. System design proposed in [16].](image)

6.3 Conclusions

The focus of this thesis is many-core systems with a large amount of cores. A likely use case is a varying work load, often below the maximum capacity. For this scenario, leakage scheduling was shown to be the most effective in a variety of papers studied during this thesis. In the design proposed by Fu and Wang (Figure 6.5), using core shut-down achieved 15% further savings
6.3. CONCLUSIONS

compared to DVFS only on a physical test setup with 2 cores. In simulations of 128 cores, 64% power was saved compared to no power management. Only using DVFS achieved 49% savings for the same task-set. In the test setup, the utilization bounds for the cores were set to that of the RM bounds (Eq. 5.1), and no deadlines were missed as long as the upper bounds was not violated.

Overhead might be the most important consideration that has to be made implementing task migration algorithms, as well as for powering off cores.
Chapter 7

Implementation: Task mapping on the TILEPro64

A proof-of-concept demo of task mapping has been developed to analyse the performance and expected power savings on the TILEPro64.

The demo is based on the communication energy task mapping from section 3.3, where the communication between tasks is assumed to be known a priori. The energy oblivious first-fit mapping is compared to the suggested algorithm in two different core layouts.

A Linux version provided with the Tilera simulator was used to execute the demo on the TILEPro64. Tasks are written as pthreads, each bound to a core, allowing data words to be sent between tasks over the UDN network. The results are derived from simulator profiling data, and the simulator configured as cycle accurate.

7.0.1 Example tasks

The demo can be described using the same task model as of that in section 3.3.1. $C = \{(t_i, t_j, w_{ij})\}$ denotes communication between two tasks, and each task sends data to two other tasks. There are 8 tasks in the demo, $T = \{A, B, C, D, E, F, G, H\}$, each located on one core. Figure 7.1 illustrates the communication between tasks, where lines in bold are between the tasks both sending to each other, resulting in a communication weight of 2.

7.0.2 The algorithm

A simple best-effort algorithm is used to map the tasks in order of communication weight.
CHAPTER 7. IMPLEMENTATION: TASK MAPPING ON THE TILEPRO64

\[ C = \{(A, E, 1), (A, H, 1), (B, D, 1), (B, G, 1), (C, F, 1), (C, H, 1), (D, A, 1), (D, E, 1), (E, D, 1), (E, G, 1), (F, B, 1), (F, C, 1), (G, B, 1), (G, F, 1), (H, A, 1), (H, C, 1)\} \]

*Figure 7.1. Task communication graph*

1. Sort all tasks in order of total communication weight with other tasks in a list of unmapped tasks

2. Find the task \( t_i \) with the highest communication weight with another task and map it to core \((0,0)\) and remove it from the unmapped list

3. Find a task from the unmapped list with the largest communication weight with a mapped task. Map this task to the core with the minimum distance to the mapped task.

4. Repeat step 2 and 3 until all tasks are mapped.

For tasks with the same communication weight, as in this example, the first match in the list is picked. Cores are chosen in the order east, south, west, north, should many be free.

### 7.0.3 Results

To calculate the average hop count for a test run, the hop count for all elements of \( C \), are added and averaged. The profiling data is visualized using an Eclipse plugin provided by Tilera. The delay for sending data between two nodes is an interesting indicator on to what extent the network is congested along the paths between specific nodes. However, only an average for all communication is included in these results. The total execution time includes only the communicating portion of the program. The results are given in figure 7.3.
7.1. CONCLUSIONS

Figure 7.2. Task mappings for the tasks in figure 7.1

7.1 Conclusions

Figure 7.3 shows some expected but interesting results. It shows that a simple algorithm can greatly outperform an oblivious mapping in all three measurements. An assumption made is that the average hop count is connected to the execution time. With the help of section 3.2, we can also conclude that the hop count also directly affects power consumption. It remains to be measured how much power is actually saved on the physical hardware.

By not limiting the proposed algorithm to a 2x8 core setup, a different mapping is generated where the hop count is actually increased. However, the average delay is lowered due to reducing network congestion along some paths. It is clear that a more complex algorithm can improve the desired result.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Execution time</th>
<th>Average Delay</th>
<th>Average Hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oblivious</td>
<td>$137 \times 10^6$ cycles</td>
<td>10.6 cycles</td>
<td>10.5</td>
</tr>
<tr>
<td>Energy, 2 rows</td>
<td>$129 \times 10^6$ cycles</td>
<td>9.6 cycles</td>
<td>5</td>
</tr>
<tr>
<td>Energy, 3 rows</td>
<td>$130 \times 10^6$ cycles</td>
<td>7.4 cycles</td>
<td>5.9</td>
</tr>
</tbody>
</table>

Figure 7.3. Result of task mappings from figure 7.2
Chapter 8

Enea OSE

OSE (Operating System Embedded) is a distributed real-time operating system designed around a message passing based programming model. Both soft and hard real-time are supported. A micro-kernel architecture offers a modular system customizable in size and functionality by the user. The multi-core edition of OSE incorporates a hybrid SMP/AMP kernel technology exploiting advantages of both models. The information is this chapter is derived from internal Enea OSE documents.

8.1 Architecture overview

The micro-kernel of OSE features a relatively small mandatory kernel component. This component is responsible for system features such as real-time scheduling, message passing, interrupt handling etc. So called core components completes the functionality of the kernel where some are mandatory (darker text in figure 8.1), and others optional. Higher level components are also optional and can be added to the system should they be required (Figure 8.3).

8.2 Processes and load modules

An OSE system consists of a monolith containing the kernel, and where OSE components and applications may also be linked during compile time, then called a core module. A module is one or several processes making up an application and may also be separately linked and loaded dynamically at runtime, then called a load module. The process is the most fundamental building block of OSE and can be seen as a thread such as POSIX threads.
CHAPTER 8. ENEA OSE

Figure 8.1. OSE kernel components

8.3 Priority and scheduling

Processes in OSE are divided into three priority bands (Figure 8.2). Interrupt processes have a higher priority than all other processes.

All priority and interrupt processes must be assigned a specific priority in the range 0-31 when created, but may have their priorities assigned according to the rate monotonic scheduling scheme. All timer interrupts execute at a certain priority, Z, and are scheduled once each specified period.

Background processes run at the lowest priority band where processes are not scheduled by priority. Instead a background process has a specified time slice, a time for which it is allowed to run before it is automatically preempted and put at the end of the ready queue.

OSE is a fully preemptive operating system.

8.3.1 Shared resources

When a set of processes share a common resource that can only be accessed by one process at a time, a mutual exclusion mechanism is needed. This functionality is provided in OSE (Section 8.5). However, if a process A of high priority, and a process B of low priority share a resource X, the problem of priority inversion may occur. Priority inversion occurs when a A is ready to run and waits for X that is used by B. But B is preempted by processes with priorities in between those of A and B. These in between processes can then be seen as running at a higher priority than A, hence priority inversion.
8.4. SIGNALS

The recommended way of designing around this problem in OSE is by avoiding shared resources being accessed by different priorities.

Another way is to implement a protocol for controlling the process priorities during access of a resource. A classic protocol is the priority inheritance protocol [37]. The basic idea is that when a shared resource is accessed by a process, this process is set to the highest priority of all processes accessing that resource. When a process is finished with the resource and unlocks the mutex, its priority is set to its original value.

8.4 Signals

A signal is a message buffer that can be sent directly between processes. Each process has an in-queue for signals sent to the process that may be searched to find a specific signal. New signal types can be defined and differently from UNIX signals, OSE signals may contain data. The main piece of information in the signal data structure is the signal number, which is used by the receiver to recognize the signal. Signals are sent by specifying the PID (Process Identifier) of the destination process. Processes of statically linked modules have their respective PID available as global variables. If the PID is not known, the hunt() system call can be used to find the PID by specifying the process’s name. The same call is used to find a process on a remote processor, after which signals can be sent without having to be aware of the physical location of the receiving process.
8.5 Semaphores and mutexes

OSE implements semaphores and mutexes to protect access to shared resources for truly parallel applications running simultaneously on different cores.

8.5.1 Semaphores

A semaphore can be seen as a global variable that is checked for its state through system calls. OSE implements counting semaphores, where a semaphore has a value. If the value is greater than zero, a process calling the waiting system call, deducts one from the value, and continues its execution. If the value is less than or equal to zero, the process will instead be blocked. The signalling system call increases the semaphore’s value.

8.5.2 Fast semaphores

A fast semaphore has the same behaviour as an ordinary semaphore, except it is not counting. Every OSE process has a built-in fast semaphore which is located in the PCB (Process Control Block) of the process. To signal the semaphore, only the PID of the process must be known. The fast semaphore is binary, and can only have either 1 or 0 as its value.

8.5.3 Mutexes

A mutex can be seen as a binary semaphore. In OSE, semaphores are accessed in kernel space, while the control block of the mutex is located in user space. Because of this, mutexes can only be shared by processes in the same memory domain. A function call to the mutex is usually faster than operating on a semaphore.

8.6 Power management

Power management in OSE is incorporated with its real-time scheduling policies. If the kernel detects that no process is scheduled to run for a certain minimum time, a power-save handler may be invoked, typically putting the processor in a low power sleep mode. The processor wakes up to run the next scheduled process, or a sporadic interrupt may awake the processor immediately. Most processors implements this "wait until interrupt" functionality. In this idle state, time measurement of a real-time system must not be lost, and
8.7. HARDWARE ABSTRACTION LAYER

Hardware timers are kept running, periodically interrupting the system. However, OSE is aware of future timed events, and will predict the next time it needs to wake-up, allowing to ignore these timer interrupts to minimize unnecessary wake-ups that consumes power. The threshold at which the power-save handler is invoked can be specified as the minimum time until the next planned timer event in milliseconds. The handler itself must be implemented by the user.

8.7 Hardware abstraction layer

The target layer (Figure 8.3) is the hardware dependent layer of OSE. The HAL (Hardware Abstraction Level) contains functionality such as MMU and cache support. The BSP (Board Support Package), contains system initialization code and device drivers. This layer must be ported to each target platform, which is a part of this thesis.

![OSE Architecture](image)

Figure 8.3. OSE architecture

8.8 OSE for multi-core

In a multi-core environment, the hybrid AMP/SMP kernel of OSE runs as a single OS image. A master core runs the main functionality of the kernel, such as managing PCB’s and system memory. Each core is instantiated with its own scheduler in a scalable AMP fashion, eliminating the need for synchronous scheduler synchronization between cores.
Synchronization between cores is instead done through a lightweight kernel inter-process communication called kernel events.

### 8.8.1 Migration and load balancing

OSE provides interfaces to measure the load of an application or core, and functionality to migrate applications between cores. Applications can also be locked to a specific core. Load balancing is expected to be customized and implemented by the user. Currently, no default load balancer is provided with OSE. Interrupt processes and timer interrupt processes cannot be migrated and are locked to the master core.

### 8.9 Conclusions

**Real-time scheduler**

The real-time scheduling policies of OSE guarantees soft and hard real-time constraints, but have no option of improving power consumption. Extensive research exists on power-aware real-time scheduling that does not jeopardise real-time constraints. We saw that OSE uses partitioned scheduling to improve scalability, and it can be assumed that a move to global scheduling is not desired.

Power-aware real-time scheduling would be possible to implement as a kernel modification of the current OSE scheduler, or as an application module. As an application, not all tasks in the system can be managed, only those explicitly added to and recognized by the application. The priority of the application must also be higher than those of the tasks it wishes to schedule.

**The power-save handler**

The OSE power save handler is offered to handle idle time in the system. The power save handler is CPU dependent and must be implemented in the HAL. It is also up to the developer to choose at what threshold time the handler should be invoked and design the handler itself. It is possible that this functionality can be used to ease the implementation of DVFS techniques.

**Load balancing**

From a load balancing perspective in a multi-core system, the distributive and modular design of OSE makes it easy to spread applications and even
8.9. CONCLUSIONS

OS services among cores with provided system calls. The communication transparency of the message-passing model is well suited, even should a service move to another node in a network. It is worth exploring to what extent the OS itself could be spread on a multi-core processor.
Chapter 9

Porting OSE to TILEPro64

This chapter describes the porting process of Enea OSE onto the TILEPro64 target platform. The port is a vision by the project to be able to evaluate OSE on a many-core platform, as well as deploying power management techniques.

9.1 Previous state of the port

Porting of an operating system can be considered an extensive and incremental task. In the initial thesis project [39], the task was therefore split into shorter milestones, each implementing some new functionality. An OSE build environment had been set up to work with the Tilera simulator. A minimal build of OSE was able to run in the simulator and write into the ramlog.

9.1.1 Target interface

The most fundamental functionality of the target interface are creating and switching a process context. This had been implemented in assembler code. This means processes could be created and scheduled, as long as context switch was event driven.

9.1.2 Board support package

The board support package contains any required board specific initialization code and device drivers. Small hardware configurations such as enabling of caches had been implemented here.
CHAPTER 9. PORTING OSE TO TILEPRO64

9.2 Porting tasks

To get a single core variant of OSE up and running on the target, a system tick and UART driver should be included in the BSP. An MMU may be implemented in the BSP, but this is not required. The target interface must be completed with interrupt context handling.

Task 1: Interrupt handling

To access the hardware interrupt vectors a virtual addressing mode was desired. Entry code and context handling for external interrupts was decided to be worked on in the parallel thesis project [34]. An interrupt controller driver must be included in the OSE device driver hierarchy to act as the interface between the CPU and drivers.

Task 2: Tick driver

OSE requires regular calls to the system call tick() to support any timed functionality, including scheduling. This should be implemented as a timer driver in the BSP.

Task 3: UART driver

To be able to interact with the OSE console, a serial driver is needed in the BSP.

9.3 Implemented tasks

Device Driver Architecture

The heart of the OSE device driver architecture (DDA) framework is the device manager, a process named devman. When the system starts, devman attempts to find and install drivers specified in the configuration. When choosing to use the DDA a certain device structure may be required. For the timer and UART driver produced in this thesis, an interrupt controller must be present to act as the interface between the CPU and device drivers (Figure 9.1).

Task 1: Interrupt handling

A virtual memory addressing mode was achieved by statically mapping memory in the TLB early in the OSE boot process. The OSE system is configured to use use 128 MB of RAM, and the entire logical address space could fit in
the available TLB entries. This way no real MMU is needed yet. Interrupt vectors for certain interrupts, such as the timer and software interrupts, were installed in the target interface.

**Task 2: Tick driver**

It was decided to use a down-counting special purpose register to serve as the timer. The hardware generates an interrupt as the counting register wraps, and the interrupt service routine calls the tick() system call. The development was eased by the well defined DDA framework.

**Task 2: Console driver**

Due to lack of time, a real UART driver for the TILEPro64 was not implemented. Instead, the complete driver structure was installed so that the driver could be used through the OSE system calls. But writing to the driver used Tilera API to output directly to the simulator. This eased further development by not having to read the ramlog.

### 9.4 Conclusions

To be able to port OSE to a new hardware, the main source of guidance has been source code for other supported hardware. OSE exists for a variety of processors. The development was heavily slowed down due to simulator problems where support could not be given, and unexpected difficulty supporting interrupts. Therefore future work remains to confirm the functionality of the implemented parts, together with that of the contribution made in parallel with this thesis [34].
Chapter 10

Conclusions and future work

10.1 Conclusions

As core count increases, operating at lower voltages and frequencies, the room for DVFS shrinks, and we may also have more cores on the chip than can be utilized under average operation. In addition, the large static power dissipation of a core can not be eliminated by DVFS techniques. If computational load instead is merged to as few cores as possible, freed-up cores can be put in a very low power sleep-mode, or be completely disconnected from the power supply.

10.1.1 Q1: What power management techniques should be considered by OSE on a many-core system and TILEPro64 respectively?

It was seen in chapter 3 and 6 that power-aware scheduling and in particular task mapping is well suitable for many-core. OSE is a real-time operating system and deadlines must always be considered.

Power-aware scheduling

The TILEPro64 offers the "nap" instruction as the only power saving capability usable in software. When the power save handler in OSE is invoked for a certain core, this instruction may be run to enter the lowest power state of the core. This method must study the OSE power save handler in more detail. At least one study has shown that power can be saved compared to not using the "nap" instruction [38].
Unfortunately, the powerful DVFS scheduling techniques can not be used on the TILEPro64 as adjusting voltage and frequency of cores is not available.

Other modern multi-core processors often supports DVFS. An example is the eight core Freescale P4080 which supports both per-core voltage and per-core frequency adjustments. In this scenario, the largest power savings can be achieved by combining DVFS with a leakage aware scheduling algorithm. Enea OSE in fact fully supports the P4080 platform, why implementing a scheduler with DVFS capabilities, with the P4080 as a demo platform could be interesting. The implementation of Seo et al [36] and Fu and Wang [16] was shown to be most effective, combining DVFS and leakage scheduling. Of these [16] gave the highest savings for assuming per-core DVFS.

Task mapping
In chapter 3 we saw how communication distance affects power consumption. The results of chapter 7 gave promising results of this concept for a simple implementation of task mapping on the TILEPro64. The large number of cores on this platform may be able to run several applications that uses several processor boards in today’s OSE systems, and optimized through task mapping. Using the UDN network for certain communication can reduce accesses to the main memory controllers as well as bypassing the cache system.

In the results in chapter 7, execution time, hop count and network congestion could be reduced. Power remains to be measured on the actual TILEPro64 hardware, which was not available during the period of this thesis. While it’s assumed that the power consumption for the actual communication is related to the hop count, is is the reduction of the total processor power that is of interest.

10.2 Future work
The future work of this thesis project remains twofold. Porting OSE to the TILEPro64 is the prerequisite to demo power management on a many-core platform using OSE. Secondly, the power management implementation must be designed and evaluated on the hardware.

10.2.1 Power management
To approach power management directly, using a Linux environment, which is already provided by Tilera is possible. Another option is executing developed
10.2. FUTURE WORK

power management software in OSE on another multi-core system, such as the eight core Freescale P4080 processor, which is already fully supported by OSE. This is due to the fact that DVFS scheduling may not be suitable on the TILEPro64, and developing the scheduler itself has a higher priority in the project by which this thesis was hosted. In the scheduler, both soft and hard real-time, as well as supporting sporadic tasks should be considered and supported to the extent it is possible.

For task mapping, the TILEPro64 is of particular interest. Mapping of tasks could be considered online or offline, and it should be investigated how communication should be defined and measured. Network congestion and defining the real-time support are other factors. In addition to map user applications, it would be interesting to see how certain OSE components could be moved to run on other cores, communicating over the on-chip network using the message-passing model.

10.2.2 Porting OSE to the TILEPro64

Work remains to complete the porting of OSE to the TILEPro64. This task mainly includes bootstrapping OSE to execute on all cores. Secondly, all desired device drivers must be developed. The following paragraphs contains unfinished parts of the port that can be used as guidelines for future work.

Target interface

Some work remains in the target interface for completing the single-core functionality. Mainly, external interrupts must be handled. The C run-time must be complemented with atomic memory manipulation, which is hardware dependent. This is required for multi-core.

Board support package

The BSP should contain all desired hardware support, such as hardware configuration and device drivers. The timer driver must be verified and the serial driver redesigned to work with the actual hardware UART, should this be desired. Multi-core support might in fact be of higher priority at this stage.

Multi-core bootstrap

OSE already exists in a multi-core variant for several target platforms. Supporting OSE for all cores of the TILEPro64 is the final goal of the porting
process this thesis has been part of. Unfortunately there was no time to complete this task.

Memory management unit
An MMU is optional, but desired at some point in development. Currently a virtual memory mode is active only with a static memory mapping.
Bibliography


[22] Hongtao Huang, Feng Xia, Jijie Wang, Siyu Lei, and Guowei Wu. Leakage-Aware Reallocation for Periodic Real-Time Tasks on Multicore
BIBLIOGRAPHY


BIBLIOGRAPHY


