Debugging Software for Multi-core Systems

ERIK AXELSSON

August 30, 2011

Master Thesis at Enea
Supervisor: Barbro Claesson
Supervisor: Detlef Scholle
Supervisor: Kathrin Dannmann
Supervisor: Francesco Robino
Examiner: Ingo Sander

TRITA-ICT-EX-2011:109
Abstract

The world of computer science has seen a big change the last years. The physical limitations in just increasing frequency when trying to increase the speed in processors has lead to a new era - multi-core. Systems based on multi-core processors brings a much higher level of flexibility to the designer, possibilities to experiment with different frequencies and voltages on on single chip. Unfortunately this flexibility also leads to a more complex system, hard to monitor and debug. The software implemented in multi-core processors needs to be parallelized and distributed very efficiently to take advantage of the architecture of the processor. The way information is exchanged between units in the processors and how the complex memory architecture, often with several levels of cache, is accessed are essential factors for the performance. It is often the case that minor changes in the software lead to big differences in performance. To be able to analyze the software when it is running on the chip it is of utmost importance to have a system that monitor the chip. One drawback with multi-core processors is that the integration of more logic into one chip decreases the external observability of the system. Hardware manufacturers have been trying to develop solutions for this problem and nowadays many processors come with an integrated system with the only purpose to support debugging and monitoring of the chip. The debugging system can be seen as a separate layer integrated on top of the system, only running in the background without affecting the the target system. In the hunt for higher performance and at the same time higher visibility this solution can be of big interest for software tool vendors and software designers.

This master thesis is divided into two parts where the first one gives an overview of the concept with multi-core processors and problems with developing efficient software for them. It also addresses why a hardware based debugging and analyzing system can be beneficial during software development. In the second part a design is developed for a hardware based debugging system, implemented in a state of the art multi-core processor from Freescale. The parallel software running on the multi-core processor is executed on top of Enea’s real time operating system OSE.
Contents

List of Figures

List of Tables

List of Abbreviations

1 Introduction 1
  1.1 Background ............................................. 1
  1.2 Problem statement ...................................... 1
  1.3 Method .................................................. 2
  1.4 Delimitations ............................................ 2
  1.5 Purpose .................................................. 2

I Pre-study 3

2 Multi-core processors 5
  2.1 Introduction ............................................. 5
  2.2 Reasons for changing to multi-core architectures .............. 5
  2.3 General multi-core architecture ............................. 6
    2.3.1 Homogeneous and heterogeneous processors ................. 6
    2.3.2 Interconnection network ................................ 7
    2.3.3 Memory ............................................... 8
    2.3.4 Cache ............................................... 8
  2.4 Multi-core from a debugging perspective ....................... 8
  2.5 Summary ................................................ 9

3 Parallel software 11
  3.1 Introduction ............................................. 11
  3.2 Basic concepts for parallel software .......................... 11
  3.3 Challenges in parallel programming ............................ 12
    3.3.1 Memory accesses .................................... 12
    3.3.2 Cache coherency .................................... 12
    3.3.3 Communication between tasks ........................... 13
8.3 Tests 48
8.3.1 Test 1 48
8.3.2 Test 2 50
8.3.3 Test 3 52
8.4 Result 53

9 Discussion 59
9.1 Conclusion 59
9.2 Personal reflections 62
9.3 Further work 63

Bibliography 65

A OSE processes 67
A.1 perfCount 67
A.2 sieve 68

List of Figures

2.1 Example of a heterogeneous processor. The IBM Cell BE has nine cores where the Power Processor Element, PPE, has a supervisory role and the eight Synergistic Processor Elements, SPE, are used for computational tasks. 7

3.1 Two threads accessing the same variables can led to data races. 13
3.2 Locks can be used to prevent data races. 14
3.3 Thread 1 waiting for resource B hold by thread 2 which is waiting for resource A hold by thread 1. 14

4.1 Hardware debug architecture. 20

6.1 Freescale QorIQ P4080 Processor. 32
6.2 Overview of an example configuration of the memory map P4080. 33
6.3 Overview of the LAWs for the memory map described in figure 6.2. Note the there is always a boot window enabled at address 0x0FF800000 to 0x0FFFFFF. 33
6.4 Overview of the functional areas in the APDA. 34
6.5 The DCSR memory map contains the different functional and cross-functional areas. .................................................... 35
6.6 Register map of the e500mc processor core. The PMRs can be seen in the lower right corner and are mirrored in the user-level registers in the top of the figure. .................................................... 36
6.7 All register for the performance counters. ................................. 37
7.1 Architecture of OSE for a multi-core processor. ......................... 39
8.1 Overview of the design. .......................................................... 44
8.2 Code to open up LAW15 for access to DCSR. ............................... 45
8.3 The memory map of the P4080 after configuration for access to DCSR. 45
8.4 Assembly instructions for accessing the performance counters. ......... 46
8.5 Inline assembler functions for accessing the PMRs. ......................... 46
8.6 Code for installing a bios function. ............................................ 47
8.7 biosOpen searches for the bios function the first time it is called. ....... 47
8.8 Code for reading the value in PMR_PMC0. ................................ 48
8.9 Flowchart for test 1. .............................................................. 49
8.10 Code for Test 1. ................................................................. 49
8.11 Flowchart for test 2. .............................................................. 50
8.12 Code for Test 2. The program is running an empty for-loop 10 million times. .......................................................... 51
8.13 Flowchart for test 3. .............................................................. 52
8.14 Code for Test 3. The program sends a message to the Sieve process with the number to calculate. The Sieve process responds when the calculation is done. .......................................................... 53

List of Tables

2.1 Requirements for a multi-core processor’s debugging system. ........... 9
3.1 Requirements for a debugging system for parallel software. ............... 17
4.1 Requirements for a hardware based debugging system. ...................... 23
5.1 Requirements, found in the pre-study, for a debugging system for parallel software in a multi-core processor. ....................... 27
8.1 Table for test 1............................................................. 54
8.2 Table for Test 2 - case 1................................................. 55
8.3 Table for Test 2 - case 2................................................. 55
8.4 Table for Test 2 - case 3................................................. 55
8.5 Table for Test 3 - case 1................................................. 56
8.6 Table for Test 3 - case 2................................................. 57
8.7 Table for Test 3 - case 3................................................. 57

9.1 Requirements, found in the pre-study, for a debugging system for parallel software in a multi-core processor................................................. 61
List of Abbreviations

APDA  Advanced QorIQ Platform Debug Architecture
AMP  Asymmetric Multi-Processing
CCSR  Configuration, Control and Status Register
CSI  Computation Specific Instrument
DCI  Debug Control Interconnect
DCSR  Debug Configuration Control and Status Register
DPAA  DataPath Acceleration Architecture
DRAM  Dynamic Random Access Memory
EPU  Event Processing Unit
iFEST  industrial Framework for Embedded Systems Tools
ILP  Instruction Level Parallelism
IPC  Inter-Process Communication
ISA  Instruction Set Architecture
LAW  Local Access Window
NAL  Nexus Aurora link
NXC  Nexus Concentrator
OS  Operating System
PMR  Performance Monitor Register
RAM  Random Access Memory
RMM  Run-Mode Monitor
RTOS  Real Time Operating System
SMP  Symmetric Multi-Processing
SRAM  Static Random Access Memory
TAP  Test Access Port
TLP  Task Level Parallelism
Chapter 1

Introduction

1.1 Background

Enea and KTH are part of an international research project, iFEST\(^1\). iFEST is a large Artemis\(^2\) project running three years. The goal is to identify how to simplify the configuration of software development during a software product life-time. More precise is to make it possible to change the tool configuration. This will require that tools are easy to integrate in the tool-chain. Today’s trend in the field of embedded systems is that multi-core solutions and parallel software are becoming more and more common. As the number of cores increases, the complexity of the software increases accordingly. This creates new kind of requirements for software, tools and software engineers. Moving towards multi-core systems makes the software more complex, the increase in complexity of debugging this software is maybe even bigger. Enea’s debugging tool Optima is used for debugging the company’s multi-core real time operating system OSE\(^3\). At the moment updates for Optima are under development for taking advantage of hardware supported debug features such as the Advanced Platform Debug Architecture in Freescale’s multi-core platform QorIQ P4080.

1.2 Problem statement

This Master Thesis will focus on requirements for debugging and analyzing performance of parallel software on multi-core systems. One problem in traditional debugging is that debugging creates software overhead that intrudes on the target application. To remove parts of this overhead, hardware debug support such as Freescale’s Advanced Platform Debug Architecture, can be used. This master thesis will focus on investigating how such hardware support can make the debugging less intrusive.

---

\(^1\)http://www.artemis-ifest.eu/
\(^2\)http://www.artemis-ju.eu/
\(^3\)http://www.enea.com/
Questions that should be answered:

- **Q1** What is the difference in debugging sequential and parallel software?
- **Q2** What support is needed in the debugging tool and the real time operating system for debugging parallel software for multi-core systems?
- **Q3** Can integrated hardware debug support make the debug support in the operating system, in terms of software overhead, removable?

### 1.3 Method

The master thesis is divided into two parts. In the first part, pre-study, an academic study aiming to answer the above mentioned questions will be conducted. An evaluation of current features for debugging parallel software for multi-core systems in Optima, and how they can be improved with hardware debug support, will be done. This part also includes gathering relevant knowledge about requirements in the iFEST framework and to study the real time operating system OSE and Freescale’s multi-core platform P4080. In the second part, design and implementation, the gathered information from the pre-study will be implemented in a use case. In the use case features for debugging parallel software, with the hardware debug support in Freescale’s P4080 platform, will be designed for Optima. This can be done as changes to the current features or designing new ones.

### 1.4 Delimitations

The project time is limited to 20 weeks and, since learning a new hardware platform can be a very time consuming task, the design might be delimited to do an investigation for the P4080 platform.

### 1.5 Purpose

The purpose for the master thesis is to identify requirements for debugging multi-core real time operating systems with Optima. Furthermore, to investigate if hardware debug support can make the debugging less intrusive. The outcome and results from the work should contribute to iFEST framework specification.

The master thesis has one goal:

**G1** Implement improvements, using Freescale’s Advanced Multi-core Debug Architecture, for Optima for handling OSE.
Part I

Pre-study
Chapter 2

Multi-core processors

2.1 Introduction

During the period of 1986-2002 the performance growth of single-core processors was at its peak[1]. Despite this the concept with multi-processors became increasingly important during the 90’s, mainly because of designers looking for a way to build servers and supercomputers and at the same time taking advantage of the cost efficiency in microprocessors. And when the limitations in exploiting instruction level parallelism, ILP, and the problems with power consumption became too big a new era in computer science began - multi-core. The first commercial general purpose multi-core processor was released year 2000 by IBM. The years after other big companies like Sun Microsystems, AMD and Intel switched focus towards integrating several cores in one chip instead of developing more aggressive single-core processors. That this is a key point in the world of computer science is undoubted, not least showed by the following statement:

"We are dedicating all of our future product development to multi-core designs. We believe this is a key inflection point for the industry".

Intel president Paul Otellini,

describing Intel’s direction at the Intel Developers Forum in 2005.

2.2 Reasons for changing to multi-core architectures

The traditional way for increasing performance in electronics has been to increase clock frequency. This became very obvious during the 90’s when new versions of personal computers were released with very short intervals. But this advance in performance also led to several challenges for the semiconductor manufacturers. The toughest challenge is the increase in power consumption and thereby also the increase in generated heat. The dynamic power consumption grows linearly with the frequency and quadratically with the voltage$^1$. A system with lower frequency

$^1$Switching power dissipated by a CMOS device: $P = CV^2 f$
could also be operated at a lower voltage, thus dividing the application between several cores can lead to a significant reduction in power consumption.

Until the 80’s performance and cost were the biggest considerations when designing electronics [2]. But when battery-powered products became more popular the power consumption became one of the key factors when manufacturing attractive products. During the last decade people have also become more environmentally conscious which has also pushed the development of energy efficient systems. Multi-core is a very good solution for handling a system’s power consumption. Often single-core solutions only use one power domain for supplying the whole system. In a multi-core system different power domains can be used and parts that are not used can be shut off. Different parts can also operate at different frequencies and voltages depending on the application.

The power consumption and thereby the increase in generated heat has been the most urgent reason for the development of multi-core solutions. But as the frequency increased and the dimensions in semiconductor technology decreased new problems started to arise due the small margins in parasitic resistance, capacitance and inductance. This led to difficulties in predicting the timing for communication in the interconnection network.

2.3 General multi-core architecture

The word core is normally used for a unit that performs calculations and a multi-core processor is a processor containing two or several cores. It can sometimes be hard to decide which units that can be considered as cores. In traditional processors only one core, i.e. one unit, performed all calculations and this could be considered as "true" single-core. But to speed up the calculation designers started to implement accelerators to calculate the heaviest tasks in parallel. This solution can typically be found in digital signal processors, DSPs, where signal processing is performed, thus requiring fast calculation. In this thesis multi-core should be seen as a processor having multiple processor cores integrated, where each processor reads and executes instructions. In a typical multi-core system each core has one or two local caches. These caches are only used by that core and are often also called private caches. On top of that sometimes another level of cache can be found that is shared between all cores. Together with a main RAM memory this creates a memory called global or shared. Processor cores and memories are the main components of a multi-core processor. To connect them with peripherals or other units integrated in the same chip an interconnection network is used.

2.3.1 Homogeneous and heterogeneous processors

Multi-core systems are often grouped into homogeneous and heterogeneous systems. A homogeneous system contains identical cores and a heterogeneous system consists of cores with different architectures. Both these solutions have pros and cons, e.g. a homogeneous architecture might be easier to understand and when programming
no consideration about on which core the program is executed has to be taken. A heterogeneous system can be more efficient if the application is tailored for the architecture but the programming can be more complex if the cores have different instructions sets and are accessing the memory differently. An example of a heterogeneous processor is IBM Cell BE, depicted in figure 2.1.

![Figure 2.1](image)

**Figure 2.1.** Example of a heterogeneous processor. The IBM Cell BE has nine cores where the Power Processor Element, PPE, has a supervisory role and the eight Synergistic Processor Elements, SPE, are used for computational tasks.

### 2.3.2 Interconnection network

The interconnection network is the physical connections between components in a multi-core system. They can be divided into two different topologies; *static*, in which the components are connected directly and *dynamic*, in which the components are connected via links and switches. The *routing technique* decides how the message is transferred between sender and receiver and together with the topology they determine the performance of the interconnection network. The task of the interconnection network is to deliver the message, which could be a data or a memory request, in shortest possible way. For architectures with a shared memory the transferring of messages combined with the memory accesses is a big part of the operation of the system. Therefore the selection of interconnection networks is very important when designing multi-core processors.
2.3.3 Memory

The access time for DRAM memories has not decreased as fast as the processor performance has increased. Since 1980, access time for DRAM memories has decreased about 25% per year while the integer and floating point performance on the SPEC Benchmark suite\(^2\) has increased about 55% and 75% per year. Therefore the memory architecture has become a key factor for achieving high performance in multi-core systems. Year 2005 memory for a multi-processor system-on-chip occupied about 70% on the die and it is estimated that this number will grow to 92% by 2014 \([2]\). Data can be exchanged between cores by reading and writing shared variables in the global memory. Shared variables is an easy way for communicating but the interconnection network must provide a high bandwidth for efficient data transferring. That is why the global memory architecture is suitable for systems with few cores but a good result is hard to achieve if more than a few dozen cores are used \([4]\).

2.3.4 Cache

A cache memory is built with SRAM and is located between the processor and the main memory. It contains a subset of the main memory and data is exchanged between them in blocks called cache blocks or cache lines. The primary goal for caches is to decrease the average memory access time. But a faster memory also results in a higher price which has led to the use of different cache levels, often level 1(L1), 2(L2) and 3(L3), ranging from small, fast and expensive L1 caches to larger, slower and more inexpensive L3 caches. Typically the caches are private, at least the L1 and L2 caches, for each core. Using a cache also necessitates a cache coherency protocol. The protocol ensures that a cache has a valid copy of the main memory and also handles writes back to the memory. The need for a protocol become especially evident in multi-core systems, where different caches hold different copies of the main memory.

2.4 Multi-core from a debugging perspective

Having many different components integrated in the same chip might be a problem when it comes to debugging. It it not possible to access all components through I/O pins since many components are hidden in the chip without a connection to the I/Os. To debug a multi-core processor the designer must be able to access each processor core and the belonging debugging features. This is especially important when a processor with different architectures of the cores, i.e. a heterogeneous processor, is used. The multi-core processor must also provide access to the interconnection network and the memory architecture, both on cache level and main memory level. These debugging features apply to the basic structure of a multi-core processor described in section 2.3. But for efficient debugging it should also be possible for the

\(^2\)Well known program for measurement of processor performance \([3]\)
2.5. SUMMARY

designer to debug all components that is specific for the chosen platform, such as Ethernet and I²C.

2.5 Summary

The traditional development focused on increasing performance through higher clock frequency but with new requirements, mainly on efficient power consumption, new ways had to be explored. Multi-core processors seems to be the future in the development of embedded systems. It offers a high flexibility for the application designer to choose between homogeneous processors for standard applications or heterogeneous processors tailored for a specific task. But this flexibility also brings a higher level of complexity in terms of advanced interconnection networks and memory architectures. A multi-core processor also makes the debugging more complex since many components are hidden in the device without a connection to the outside world. Table 2.1 summarizes the requirements for a debugging system in a multi-core processor.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide access to each core’s debugging features</td>
<td>REQ-01</td>
</tr>
<tr>
<td>Provide access to each core’s cache memory statistics</td>
<td>REQ-02</td>
</tr>
<tr>
<td>Provide access to the memory on device level, i.e. shared main memory and</td>
<td>REQ-03</td>
</tr>
<tr>
<td>shared cache memory</td>
<td></td>
</tr>
<tr>
<td>Provide access to the interconnection network</td>
<td>REQ-04</td>
</tr>
<tr>
<td>Collect information from all processor cores simultaneously</td>
<td>REQ-05</td>
</tr>
</tbody>
</table>

Table 2.1. Requirements for a multi-core processor’s debugging system.
Chapter 3

Parallel software

3.1 Introduction

Parallel programming and efficient parallel software is not a new concept. It has been used for many years, especially in simulation of scientific computing intensive tasks. The, probably, most well-known area is the simulation of weather forecasts. It has also been used in the automotive industry for simulating a vehicle’s air resistance and for crash tests. With the arrival of multi-core processors the need for methods for developing efficient parallel software increases heavily.

3.2 Basic concepts for parallel software

The first step in parallelizing an application is to split the application into different parts, called tasks, which can be calculated in parallel. A task is assigned to a process or thread which in turn is assigned to a physical unit. The assignment of tasks to a process or thread is called scheduling and decides in which order the tasks are executed. The assignment of processes and threads to a physical unit is called mapping. A rough breakdown in memory architecture is to differentiate between distributed memory and shared memory. Distributed memory refers to a system with multiple processors in which each processor has its own memory and a shared memory is a memory that is used by all processors or cores. Threads are often connected to shared memory and processes to distributed memory [4]. In this thesis focus will be on threads and shared memory. In a shared memory environment each thread can access data in the global memory in terms of shared variables. It is also possible for a thread to have private variables, which are not accessible for other threads. In contrast, in distributed memory architectures cores can exchange information with message passing. In message passing data is transferred between two cores’ local memory and to achieve this the two communicating cores must use send and receive operations.

To handle the organization of the parallelized software most often an operating system, OS, is used. Many OSs used today are developed for single-core processors
and still much work has to be done for taking advantage of the power of parallelized hardware. Similar to hardware the OSs for parallel software can be divided into two categories; symmetric multiprocessing, SMP and asymmetric multiprocessing, AMP. In SMP one OS instance is used, distributing the load equally over all of the cores, without taking into account on which core a specific task is executed. In AMP on the other hand different operating systems can be used on each of the cores or some cores can be used for specifically dedicated tasks. The categorization of parallel software into SMP and AMP has similarities with that for hardware. As for a homogeneous processor SMP might be easier to understand since it only involves one OS instance. It might also be easier to develop applications for SMP since not as much care needs to be taken about the underlaying hardware. But an AMP application can be more efficient if it is well adopted to the hardware. If this categorization is to be followed SMP can only be performed on a homogeneous processor and AMP can be performed on both homogeneous and heterogeneous processors. Hybrids also exists where e.g. half of the cores are running SMP and the other half AMP.

3.3 Challenges in parallel programming

3.3.1 Memory accesses

As discussed in section 2.3.3 the access time for memories has not decreased in the same rate as processor performance has increased. Because of this the organization of accesses to memory has become a very important factor for developing efficient programs. This is especially true for parallel programs where a shared memory is used. To decrease the average memory access time a cache is used. But if a shared memory is used data can be replicated in several caches. If a core performs a write data must be written back to the shared memory before any other core is allowed to read the same data. The problem of maintaining a coherent view of the memory is referred to as the cache coherency problem. To keep track of data residing in different caches a cache coherency protocol is used.

3.3.2 Cache coherency

The cache coherency problem is the problem to keep a valid copy of the main memory in the cache. A cache can significantly improve performance for memory accesses but the characteristics of the performance gain in a multi-core processor does not look the same as for single-core processors [5]. One problem that caches for multi-core system introduce is false sharing. False sharing is the decrease in performance when two processors are operating on the same memory block managed by the caching mechanism. Although they do not use each others addresses in the memory block the mechanism forces them to reload their cache.
3.3.3 Communication between tasks

When threads are executing in parallel they most often need to exchange information. This is referred to as inter-process communication, IPC. IPC can be done in several ways e.g. shared memory where the processes communicate by writing to the same variables or processes communicate by sending messages or signals to each other. If a shared memory is used care needs to be taken about accesses to the memory according to what is described in section 3.3.1. If two processes perform calculations and want to exchange results via shared variables, the memory accesses should not be the most time consuming part of the total execution. In that case it might be a better solution to include both calculations in the same process. The same principle applies to message passing, the most of the time should be spent on calculations, not sending messages.

3.3.4 Data races

Concurrent access to a shared memory can lead to race conditions. Due to timing in two or several threads the application outputs different results. A simple example could be two threads executing the code in figure 3.1.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Thread 1 & Thread 2 \\
\hline
\(t = x\) & \(u = x\) \\
\(x = t + 1\) & \(x = u + 2\) \\
\hline
\end{tabular}
\caption{Two threads accessing the same variables can led to data races.}
\end{figure}

Depending on the timing of the two threads the value of \(x\) after execution can be 1,2 or 3, assuming that all variables are initialized to 0.

To get rid of this problem a method needs to be used to decide which process that is allowed to access the memory or any other resource that is shared between processes. This is called synchronization and for that locks, e.g. semaphores, can be used. If the same example as in figure 3.1 is executed but implemented with semaphores it should look like in figure 3.2.

If variable \(sem\) can have the values 0 and 1 and the acquire function decreases the value by 1 and the release function increase the value by 1 it can always be guaranteed that only one thread is executing. In this scenario the value of \(x\) after execution can only be 3.
CHAPTER 3. PARALLEL SOFTWARE

Thread 1
acquire(sem)
t = x
x = t + 1
release(sem)

Thread 2
acquire(sem)
u = x
x = u + 2
release(sem)

Figure 3.2. Locks can be used to prevent data races.

3.3.5 Deadlocks

Section 3.3.4 describes a method to get rid of the problem with race conditions by using locks. Unfortunately locks can introduce new problems. The first one is called deadlocks and occurs when two threads have locked a resource and at the same time wait for the resource locked by the other thread. If the same example as in section 3.3.4 is used it could look like in figure 3.3.

Thread 1
acquire(resA)
t = x
acquire(resB)
x = t + 1
release(resB)
release(resA)

Thread 2
acquire(resB)
u = x
acquire(resA)
x = u + 2
release(resA)
release(resB)

Figure 3.3. Thread 1 waiting for resource B hold by thread 2 which is waiting for resource A hold by thread 1.

Locks also have another disadvantage. If many locks are used for synchronization this can lead to sequentialization, which means that only one or a few threads can be running since the others are waiting. It is also important to make the critical section, which is executed between the lock and unlock operation, as small as possible to reduce waiting times.

3.3.6 Load balancing

Load balancing is the distribution of workload between processors. The load balancing depends heavily on the applications task level parallelism, TLP, which describes the ability of the application to be split up in different parts that can be executed in parallel. The ideal case is to get as many parts as there are cores implemented in the processor and all parts should be of the same size. As the number of cores
3.4.2 Debugging

Debugging of a single-core system is known to be a challenging task. The transition to multi-core solutions has further complicated this task. The flexibility that multi-core brings, discussed in chapter 2, also brings some new concerns regarding debugging. If an entire system is stopped it can relatively easy be analyzed but e.g. if only one core is stopped the rest of the cores can continue sending data to or wait for data from the stopped core. In single-core systems bugs are often deterministic, caused by a special set of input [7]. These bugs also appear in multi-core systems and can be debugged the same way as in single-core systems. But parallel programs also bring a new kind of bugs, caused by the interaction between tasks. These bugs often depend on the precise timing in the execution or the communication between tasks. The precision in the timing makes the bug very hard to provoke and reproduce. And even if the bug is reproducible it can be hard for the designer to understand the problem, since small margin in communication is hard to monitor.

3.4.2 Tracing

An indirect method of debugging is to gather information about the system during execution. This method is called tracing and brings many advantages. If a system displays an erroneous behavior and tracing is enabled, the trace hopefully can provide information to the designer to solve the problem. Tracing is also used in performance analysis where the collected information gives information e.g. how
often an instruction in the code is executed. This can be used to calculate the speed of the program. Tracing means collecting information, thereby creating a need for the information to be stored somewhere. This might be the biggest problem with tracing. A trace with a lot of information can be beneficial for the designer since more information makes it easier to create a better overview of the system. But more information also brings the cost of a bigger trace buffer. One example is discussed by Chung-Fu Kao, saying that a microprocessor running at a frequency of 100 MHz can generate trace data at the size of 400 MB per second [8]. If the tracing is implemented in software it intrudes on the program under observation, leading to a decrease in performance. When a trace buffer is implemented in hardware an extra memory buffer is needed to store the information. One way of using tracing is for post-mortem debugging. In post-mortem debugging information is gathered during run-time and if a system failure occurs information can be read from the trace buffer to create a scenario of what happened just before the failure.

3.4.3 Types of debugging

According to Chi-Neng Wen et al. [9] the solutions for debugging parallel software can be divided into two categories: software based solutions and hardware based solutions. In software based solutions extra code is integrated in the program with the only purpose to support debugging or performance analysis. This is called instrumentation code and in its simplest form it can be using the printf function or switching LEDs on and off. This technique is many times sufficient but also has some disadvantages. Except the fact that it leads to a degradation in performance since extra instructions have to be executed it also cause probe effects which is a phenomenon that Dieter Kranzlmüller et al. [10] describe as:

"Program executions are affected by observation mechanisms introduced for program analysis. Monitoring a program affects the probability of choices at nondeterministic events, thus yielding different results than without monitoring."

This can be a severe problem, especially when analyzing a hard real-time system. There is always a possibility to leave the instrumentation code in the program when it is running in its intended environment but at the cost of performance described earlier. And when the scope of the analysis is increased a point is always reached where the instrumentation code takes too large share of the total program.

3.5 Summary

Writing code for parallel applications is well-known as a challenging task. Trying to optimize the threads is not sufficient since the interaction between tasks and the competing for resources heavily influence the performance. Examples of what can degrade the performance are false sharing and spending too much time on message passing between cores. Problems such as implementing an efficient communication
between tasks or the risk of deadlocks can exist both for single-core and multi-core processors. In a single-core processor a multitasking operating system can be used, thus introducing a way for executing threads in parallel even if they are not executing at the same time. But all problems with software executing in parallel have a tendency to be worse when executing on a multi-core processor since it is much harder for the designer to predict when a certain step in the code is reached. For a multi-core processor it is also possible to chose different kind of RTOSs, depending on the processor’s hardware architecture and the wanted software characteristics.

The higher level of complexity in software makes the debugging much more complicated. An useful parallel software debugging system should be able to handle different kinds of RTOSs and also be able to monitor the communication in the application and the accesses to memory. One potential big disadvantage with tools for debugging and performance analysis is the overhead created in software. The most basic method of evaluating the impact of the overhead is to compare the execution time with and without the overhead introduced in the code. But the impact of the overhead can be very different and a small overhead can heavily decrease the performance whilst a large overhead might not decrease the performance at all. Techniques exist for analyzing this impact[11] but a better solution would be to use a non-intrusive technique for collecting information. Table 3.1 summarizes the requirements for a debugging system for parallel software applications.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide support for different kinds of RTOS models, e.g AMP and SMP RTOSs</td>
<td>REQ-06</td>
</tr>
<tr>
<td>Provide support for monitoring the communication between tasks in an parallel software application</td>
<td>REQ-07</td>
</tr>
<tr>
<td>Provide support for monitoring a parallel software application’s accesses to a shared memory</td>
<td>REQ-08</td>
</tr>
</tbody>
</table>

Table 3.1. Requirements for a debugging system for parallel software.
Chapter 4

Debugging software for multi-core processors using hardware based debug support

4.1 Introduction

In so-called hardware based debugging extra hardware support is used for debugging and tracing the application. This hardware support is implemented as logic in the hardware and its only task is to support debugging and observation of the chip. Several different solution from the big manufacturers exist e.g. Multi-Core Debug Solution from Infineon[12] and CoreSight On-chip Trace & Debug Architecture from ARM[13]. This master thesis presents a general model of how a system described above can look like. This model is mostly derived from the model described by Georgios Kornaros et al. [2].

4.2 Architecture

In hardware based debugging dedicated IP blocks are integrated and interconnected in the chip to form a system for debugging and tracing. The IP blocks connect to different targets such as processor cores, shared memory and interconnection network. The system consists of different components, depicted in figure 4.1:

- Monitors — Observe calculations and communication and generate events.
- Trace buffer — collect trace information from the targets.
- Computation specific instruments (CSI) — located inside or very close to the target and react on events and control calculations inside the target.
- Debug control interconnect (DCI) — Handling the programming of the IP blocks and to where the debug data should be routed.
4.2.1 Monitors

Monitors observe the behavior of the chip during execution. They can be programmed to generate events at some point or when a certain state of the system is reached. The events can be distributed across the chip and can also be combined with other events to affect the execution of the chip or what information should be logged. Monitors are often divided in computation monitors and communication monitors. Computation monitors monitor the calculations inside a core and can generate an event when e.g. the CPU executes some line of code or a certain number of clock cycles after an event has occurred. Communication monitors monitor the communication between cores and can e.g. monitor an address- or data range of interest.
4.3. NEXUS 5001

4.2.2 Trace buffer

In the trace buffer trace data is gathered from the different targets. This data can be saved in an on-chip buffer which is the fastest but also the smallest memory. The size is typically in kB. The data from the trace can also be sent to an external memory. This way more data can be saved but this will also affect the application since it shares the bus with the targets. The third option is to send the trace data to an external tool via a communication interface. In that case the bandwidth of the interface decides how much data that can be sent.

4.2.3 Computation-Specific Instrument

CSIs are located inside or close to the target. It controls the core and can start, stop and single step the core after the receipt of an event. It also sends information and events to the monitor and the trace buffer. The CSI is often a big and complex component in itself considered the fact that it can be the entire set of debugging features for a single processor core.

4.2.4 Debug Control Interconnect

The DCI provides a way to control and query the different IP blocks in the debug system. Special registers called test point register are connected to the test access port, TAP, controller, which is the connection to the outside world, and can be programmed this way. Controlling the architecture is a low-level task it means programming on register level. The configuration also needs to be changed when a new platform is released, in the same way as a new processor needs support in the intended compiler. Some researchers have made attempts to design high-level languages for this task [14] [15] but so far no distinct standard have reached the market.

4.3 Nexus 5001

A standardized interface for accessing the hardware based debugging system would be very helpful the software designer. This section describes an example of such an interface, Nexus 5001.

Nexus 5001 Forum [16], former known as Global Processor Debug Interface Standard Consortium, GEPDISC, is a group of companies that was formed with the purpose to define a standard debug interface for control applications. The defined standard is called Nexus 5001 Forum Standard for Global Embedded Processors, here called Nexus 5001. Nexus 5001 is an open industrial standard interface for development and debugging of software for embedded systems.

Due to the higher level of integration in today’s processors the Nexus 5001 states some features it should address:
CHAPTER 4. DEBUGGING SOFTWARE FOR MULTI-CORE PROCESSORS USING HARDWARE BASED DEBUG SUPPORT

- Program and data trace visibility is needed for development tools with acceptable impact to the system under development.

- A standard development methodology and tool set is needed for embedded applications.

- An independent processor pin interface standard is needed.

To support this, the Nexus 5001 has defined an Application Programming Interface, API. The API has been designed to suit many different kinds of systems, independent of hardware platform, RTOS and compiler. Nexus 5001 also provide recommendations for registers that could be implemented in hardware for exchanging control and status information. However, this is not a requirement. Nexus 5001 also has definitions for the pin interface on the hardware.

4.4 Summary

Systems based on multi-core are becoming more popular and the cost for software development for these systems increases accordingly. This means that developing optimized code without bugs is increasingly important. Unfortunately this is hard to achieve with todays software tools[17] [18]. Software based debugging means that extra code is introduced in the application to perform tasks linked to debugging and observing the application. This extra code can create a new, undesirable behavior of the application. Examples of what it could affect is how well the program fits in the cache[18] and the communication on the common bus. To be able to debug and observe a system non-intrusively complex on-chip hardware could be used[14]. This hardware is implemented as components for different targets forming a system which can be configured. The configuration is a rather complex task since it means programming on register level. Attempts have been made to define a high level language[14] [15] but so far no distinct standard have reached the market. One drawback with hardware based debugging is that the debug tool must be ported towards the platform’s specific debug hardware. Multi-core is still in the beginning of its lifetime and since debugging features typically are a step behind these systems could be expected to undergo heavy changes in the upcoming years. Table 3.1 summarizes the requirements for a hardware based debugging system for a multi-core processor.
4.4. SUMMARY

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide a non-intrusive debugging method</td>
<td>REQ-09</td>
</tr>
<tr>
<td>Provide a standardized interface for accessing the hardware based debugging system, both from an external tool and from within the multi-core processor</td>
<td>REQ-10</td>
</tr>
<tr>
<td>A high level language should exist for configuration of the hardware based debugging system</td>
<td>REQ-11</td>
</tr>
</tbody>
</table>

Table 4.1. Requirements for a hardware based debugging system.
Chapter 5

Conclusion from pre-study

Multi-core definitely seems to be the future for processors used in embedded system. The flexibility in hardware introduce a new way for the designer to vary the frequency and voltage across the platform. The designer can also choose between different architectures such as homogeneous or heterogeneous processors to fully meet the requirements of the application. To take advantage of the power of multi-core solutions the designer needs to take several concerns into account. The probably most important factor is to access the memory in best possible way. Section 2.3.3 showed that the access time for memories is a very limiting factor and section 3.3.1 further discussed how this is important in parallel software. From this it can easily be understood that lots of accesses to main memory together with an inefficient use of the cache memories, which e.g. can be both false sharing and that the software does not fit well into the cache, can create a significant decrease in the performance of the system. Parallel software introduces new software problems such as keeping a coherent memory and not introducing deadlocks in the system.

A solution based on a multi-core processor

- **Conclusion 1**
  - gives the designer many tools to tailor the application to special requirements, reaching a high performance with a low consumption in power and area.

but it also

- **Conclusion 2**
  - introduce the designer to new problems such as keeping efficient memory accesses and distributing the load across the platform. These problems can have severe impact of the system’s performance and thus removing the benefits with a multi-core processor.

A system based on a multi-core processor also introduce problems when it comes to debugging, discussed in section 3.4.1 and 3.4.2. Now parts of the system can be
stopped and analyzed while other parts still run at full speed. All problems found in a single-core processor can also be found in a multi-core processor plus the fact that new problems arise, often dependent of the timing in the communication between tasks. It can be said that

- **Conclusion 3**
  - the move from single-core towards multi-core has complicated an already complex task.

The traditional way for debugging is to insert instrumentation code in the software, discussed in section 3.4.3. The instrumentation code becomes part of the program since it is instructions executing in the same way as all ordinary instructions. This means that

- **Conclusion 4**
  - software based debugging introduce an non-desired overhead that has a negative impact of the performance and in worst case can lead to bugs only apparent when the system is running in its intended environment.

The new way for developing debugging systems is to implement a complete system running besides the target system and only performs tasks for debugging and observation. This is a powerful tool when developing multi-core application, therefore the whole chapter 4 is dedicated for that. Hardware based debugging is a new concept and needs to undergo several big changes before being accepted across the industry of embedded systems.

- **Conclusion 5**
  - A standardized, non-intrusive, hardware based debugging system can be very helpful for the designer (REQ-09 - REQ-10). This is especially true for multi-core systems and systems running under hard real-time requirements.

- **Conclusion 6**
  - The debugging system should provide access to the components integrated in the processor, e.g. processor cores, interconnection network and memory (REQ-01 - REQ-05), thus providing an alternative for debugging parallel software running on the processor (REQ-06 - REQ-08).

Configuration of the hardware based debugging system is a complex and time consuming tasks since it introduce programming on register level. Therefore

- **Conclusion 7**
  - a standardized high level language for configuration of the hardware based debugging system is of great importance to simplify and decrease the time consumption for debugging (REQ-11).

Table 5.1 summarizes the requirements, found in the pre-study, for a debugging system for parallel software in a multi-core processor.
5.1. FEEDBACK TO PROBLEM STATEMENT

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide access to each core’s debugging features</td>
<td>REQ-01</td>
</tr>
<tr>
<td>Provide access to each core’s cache memory statistics</td>
<td>REQ-02</td>
</tr>
<tr>
<td>Provide access to the memory on device level, i.e. shared main memory and shared cache memory</td>
<td>REQ-03</td>
</tr>
<tr>
<td>Provide access to the interconnection network</td>
<td>REQ-04</td>
</tr>
<tr>
<td>Collect information from all processor cores simultaneously</td>
<td>REQ-05</td>
</tr>
<tr>
<td>Provide support for different kinds of RTOS models, e.g. AMP and SMP RTOSs</td>
<td>REQ-06</td>
</tr>
<tr>
<td>Provide support for monitoring the communication</td>
<td>REQ-07</td>
</tr>
<tr>
<td>between tasks in an parallel software application</td>
<td></td>
</tr>
<tr>
<td>Provide support for monitoring a parallel software application’s accesses to a shared memory</td>
<td>REQ-08</td>
</tr>
<tr>
<td>Provide a non-intrusive debugging method</td>
<td>REQ-09</td>
</tr>
<tr>
<td>Provide a standardized interface for accessing the hardware based debugging system, both from an external tool and from within the multi-core processor</td>
<td>REQ-10</td>
</tr>
<tr>
<td>A high level language should exist for configuration of the hardware based debugging system</td>
<td>REQ-11</td>
</tr>
</tbody>
</table>

Table 5.1. Requirements, found in the pre-study, for a debugging system for parallel software in a multi-core processor.

5.1 Feedback to problem statement

The purpose of the pre-study was to investigate how debugging of multi-core systems is done. As a help for solving this the problem statement in section 1.2 stated three questions that should be answered. With the knowledge gathered from the pre-study the questions can now be answered.

- Q1 What is the difference in debugging sequential and parallel software?

  Answer There is no distinct difference between the support needed for debugging sequential and parallel software. But the traditional methods used for debugging sequential software, i.e. software based debugging, might have a larger influence when it comes to parallel software, thus making them inappropriate.

- Q2 What support is needed in the debugging tool and the real time operating system for debugging parallel software for multi-core systems?

  Answer The best way is to use a hardware based debugging system. A
standardized high level language for configuration of the hardware based debugging system is of great importance to simplify and decrease the time consumption for debugging.

• **Q3** *Can integrated hardware debug support make the debug support in the operating system, in terms of software overhead, removable?*

**Answer** Yes, a debugging system such as the one described in chapter 4 can be used. The debugging system runs without affecting the target application, thus creating a non-intrusive alternative for debugging.
Part II

Design and Implementation
Chapter 6

QorIQ P4080 processor

Freescale’s QorIQ P4080 processor [19] is a high performance multi-core processor well-suited for highly compute-intensive or I/O-intensive applications. This makes it a good choice for communication platforms such as routers or base band stations. The P4080 processor is built in 45 nm technology and has:

- eight e500mc cores built using the Power Architecture\(^1\) and possible to operate at frequencies up to 1.5 GHz. The P4080 processor is highly configurable and can operate with various degrees of symmetry, from eight symmetric cores to eight asymmetric cores. The cores can run independently of each other with different frequencies and different operating systems.

- a memory architecture consisting of a private L1 data and instruction cache of 32 kB each and a 128 kB L2 cache. P4080 also has a 2 MB L3 cache shared between all cores and two memory controllers for a DDR main memory.

- a Datapath Acceleration Architecture, DPAA, used for accelerating the processing of network traffic.

- high speed peripheral interface for PCI Express and serial RapidIO communication.

- additional peripheral interfaces such as Ethernet, USB and I\(^2\)C.

- a hardware based debugging system called Advanced QorIQ Platform Debug Architecture.

- a high speed interconnection network called CoreNet, used to connect all hardware blocks of the P4080.

\(^1\)http://www.power.org/home
6.1 Memory map

P4080 has several address domains but this report will be limited to three of them:

- the internal local address space, i.e. the entire address space for the processor.
- the internal configuration, control and status register (CCSR) address space, which is a subset of the internal local address space
- the internal debug control and status register (DCSR), which is another subset of the internal address space.

The internal local address space is the physical 36-bit address space in the device. The e500mc cores can access the internal local address space through a local access window, LAW. The LAW defines the path between the source and the target for a transaction, i.e. the route in the CoreNet Coherency Fabric. P4080 has a set of 32 LAWs and each LAW can be used to access a target such as a DDR memory controller or a RapidIO port. LAWs are configured by writing to the associated register that reside in the CCSR register map.
6.2 ADVANCED QORIQ PLATFORM DEBUG ARCHITECTURE

In the P4080 platform an infrastructure is integrated to support run control, performance monitoring and tracing. The infrastructure is called Advanced QorIQ Platform Debug Architecture, APDA [20]. P4080 has five different functional areas; CoreNet, DDR, Data Path, OceaN and e550mc processor cores. Each functional area has its own function-specific debug logic which generates information to the cross-functional components. The cross-functional components operate across the platform, sending and receiving information to and from the different functional areas. Four cross-functional debug components are integrated; Event Processing Unit (EPU), Nexus Port Controller (NPC), Nexus Concentrator (NXC) and Nexus Aurora Link (NAL). As the name suggest the NPC, NXC and NAL are compliant to Nexus 5001. The NXC and NAL will not be covered in this report but in brief the NAL is a high speed interface to an external debugger and the NXC works as a kind of filter for trace messages. Among the functional areas only the e500mc pro-
CHAPTER 6. QORIQ P4080 PROCESSOR

The processor cores will be covered. The APDA is setup and configured by programming dedicated registers in the Configuration, Control and Status Register map, CCSR, and the Debug Configuration Control and Status Register map, DCSR. Relevant data can be sent to memories on- or offchip or to an external tool. An overview of the APDA can be seen in figure 6.4 and the architecture can be compared with the architecture discussed in chapter 4.

6.2.1 Event Processing Unit (EPU)

The EPU is the heart of the APDA and can be referred to as the monitor in chapter 4. The EPU is not designed according to Nexus 5001 but interacts with the Nexus 5001 facilities in P4080. The EPU houses a number of event counters and also registers for controlling the EPU. In total 2048 different event signals can be routed to 32 performance counters in the EPU. The events can also be combined with each other, producing a new event. The counters can be chained if the original 16 bits
6.3. E500MC PROCESSOR CORE

are not enough. This creates a very high flexibility for the user to combine various events for performance monitoring or debugging. Some examples are matching values in different counters, sending interrupts to the interrupt controller for the device or sending a stop request to any of the cores.

6.2.2 Nexus Port Controller (NPC)

The NPC can be referred to as the trace buffer in chapter 4. The NPC has a 16 kB tracebuffer implemented but also handles the connection to external debuggers through the NAL and the connection to external memory.

6.2.3 DCSR memory map

To support debug and performance operations in the device Freescale has implemented an entirely separate address space called DCSR. The DCSR houses almost all registers needed to configure and control the APDA. As seen in figure 6.5 the memory map contains all the different functional and cross functional units.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Block</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000_0000 - 0x0000_0FFC</td>
<td>Event Processing Unit (EPU)</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>0x0000_1000 - 0x0000_1FFC</td>
<td>Nexus Port Controller (NPC)</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>0x0000_2000 - 0x0000_2FFC</td>
<td>Nexus Concentrator (NXC)</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x0000_8000 - 0x0000_8FFC</td>
<td>CoreNet Debug Section 1 (Section 2 is listed below at 0x0008_0000)</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>0x0000_8000</td>
<td>DPAA Debug</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x0001_1000 - 0x0001_1FFC</td>
<td>OCeInN Debug</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>0x0001_2000 - 0x0001_2FFC</td>
<td>DDR1 Debug</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>0x0001_3000 - 0x0001_3FFC</td>
<td>DDR2 Debug</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x0001_8000 - 0x0001_8FFC</td>
<td>Nexus Aurora Link Layer (NAL)</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x0008_0000 - 0x0008_0FFC</td>
<td>CoreNet Debug Section 2 (Section 1 is listed above at 0x0000_8000)</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>0x0100_0000 - 0x0100_7FFC</td>
<td>NPC Trace Buffer (16KB effective storage space after ECC bits are discounted)</td>
<td>32 Kbytes</td>
</tr>
</tbody>
</table>

Figure 6.5. The DCSR memory map contains the different functional and cross-functional areas.

6.3 e500mc processor core

The P4080 platform contains eight full featured e500mc processor cores [21] [22]. The e500mc core is implemented using the Power Architecture technology. Power Architecture can be divided into two categories; Desktop PowerPc architecture and
Power ISA. The Power ISA can further be categorized into an embedded part and a server part. The e500mc core belongs to the embedded part. The Power Architecture aims for reaching highest possible compatibility on application-level independently of the category of the core.

The e500mc register set has a special set of registers for performance monitoring called performance monitor registers, PMRs. The PMRs consists of counter regis-

---

**Figure 6.6.** Register map of the e500mc processor core. The PMRs can be seen in the lower right corner and are mirrored in the user-level registers in the top of the figure.
6.3. E500MC PROCESSOR CORE

ters, local control register and a global control register. Each register comes in two
versions, one that is read- and writable in supervisor mode and one that is readable
from user mode. These facilities are implemented in the core and are not part of
the APDA. But APDA can interact with the PMRs by core watchpoints and core
interrupts.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Abbreviation</th>
<th>PMR Number</th>
<th>pmr[0–4]</th>
<th>pmr[5–9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter 0</td>
<td>PMC0</td>
<td>16</td>
<td>00000</td>
<td>10000</td>
</tr>
<tr>
<td>Counter 1</td>
<td>PMC1</td>
<td>17</td>
<td>00000</td>
<td>10001</td>
</tr>
<tr>
<td>Counter 2</td>
<td>PMC2</td>
<td>18</td>
<td>00000</td>
<td>10010</td>
</tr>
<tr>
<td>Counter 3</td>
<td>PMC3</td>
<td>19</td>
<td>00000</td>
<td>10011</td>
</tr>
<tr>
<td>Local control a0</td>
<td>PMLCa0</td>
<td>144</td>
<td>00100</td>
<td>10000</td>
</tr>
<tr>
<td>Local control a1</td>
<td>PMLCa1</td>
<td>145</td>
<td>00100</td>
<td>10001</td>
</tr>
<tr>
<td>Local control a2</td>
<td>PMLCa2</td>
<td>146</td>
<td>00100</td>
<td>10010</td>
</tr>
<tr>
<td>Local control a3</td>
<td>PMLCa3</td>
<td>147</td>
<td>00100</td>
<td>10011</td>
</tr>
<tr>
<td>Local control b0</td>
<td>PMLCb0</td>
<td>272</td>
<td>01000</td>
<td>10000</td>
</tr>
<tr>
<td>Local control b1</td>
<td>PMLCb1</td>
<td>273</td>
<td>01000</td>
<td>10001</td>
</tr>
<tr>
<td>Local control b2</td>
<td>PMLCb2</td>
<td>274</td>
<td>01000</td>
<td>10010</td>
</tr>
<tr>
<td>Local control b3</td>
<td>PMLCb3</td>
<td>275</td>
<td>01000</td>
<td>10011</td>
</tr>
<tr>
<td>Global control 0</td>
<td>PMGC0</td>
<td>400</td>
<td>01100</td>
<td>10000</td>
</tr>
</tbody>
</table>

Figure 6.7. All register for the performance counters.
Chapter 7

Enea OSE for multi-core processors

Enea OSE is a real-time operating system mainly used in soft real-time systems. It is developed to suit many different applications, ranging from communication infrastructure to industrial control systems. Enea OSE is a widespread RTOS that can be found in one third of all mobile phones sold worldwide and more than half of all mobile calls in the world are switched through software from Enea [23]. The latest version of OSE, 5.5, includes extensions to run on multi-core processors. This report will focus on the extensions for the adaption of OSE to multi-core processors.

7.1 Architecture

OSE can not be categorized as a pure SMP or AMP RTOS. Instead Enea describes OSE as a hybrid between the two categories. A typical OSE system consists of some essential parts, depicted in figure 7.1.

Figure 7.1. Architecture of OSE for a multi-core processor.
The main part of the OSE kernel is implemented on one core, typically core 0.

The rest of the cores only houses a few kernel services, mainly for memory management and scheduling on the specific core.

Load modules are the highest level of building blocks that the application programmer uses when building an OSE system. The load module is assigned a portion of the memory of the system.

Processes resides within the load module. The processes are assigned priorities and share memory within the load module.

Exchange of information is done by message passing between processes. The message passing is performed in the same way even if the processes are located on different cores.

The Run-mode monitor (RMM), executed on core 0, offer metrics to the user through Optima.

The rest of the cores has a smaller version of the RMM.

### 7.2 Parallelism

On the kernel level OSE can be seen as an AMP RTOS since most of the OSE services are executed on core 0. On application level OSE offers parallelism at load module level, meaning that a load module is located on one core. This also means that no parallelism is offered at process level but it is possible to move load modules between cores. OSE does not contain a load balancer and the designer decides on which core the load module is being executed. The designer can receive information about the load of the system by using a signal interface to the program manager. The program manager is an essential component in OSE, responsible for load module management. Since no load balancer exists and it is up to the designer to distribute the load OSE could be seen as an AMP RTOS on application level. But at the same time the load modules can be moved to any of the cores which could be a sign of a SMP RTOS. OSE also has a sort of bare metal mode. In this mode the core starts as normal but only one process is running on the core and without pre-emption. These factors makes it hard to categorize OSE as either an AMP or s SMP RTOS, therefore Enea describes it as a hybrid between the two categories.

### 7.3 Run-Mode Monitor

The Run-Mode Monitor, RMM, is a feature in OSE for providing information for debugging and analysis of the system. It is designed for exchanging information with
7.3. RUN-MODE MONITOR

Optima, which is Enea’s debugging and profiling tool. The RMM is implemented as an OSE process and executes during context switches between other processes. Together with Optima it can provide metrics to the designer such as:

- Information about the load on a CPU, program or priority level.
- Event logging.
- Breakpoint handling.
- Debug information.

As stated above, the RMM is implemented in software, categorizing itself as a software based debugging and analyzing feature.
Chapter 8

Design and use case

8.1 Design specification

In the design the QorIQ P4080 multi-core processor from Freescale, described in chapter 6, is used. The idea with the design is to take advantage of the APDA to collect performance measurements for software running on the system. The CSIs, in this case the processor cores, should be configured to count the number of cache misses in L1 data and instruction cache and L2 cache for each core. The number of clock cycles should also be counted. It is important to separate the design into what is performed in the APDA and what is performed inside the CSIs since the design on CSI level is done according to the instruction set architecture, ISA, for the cores. This is invisible for the APDA but the CSIs interacts with the APDA by sending trace data. The APDA should be designed to receive data from the CSIs into its trace buffer. Core 0 is used to get the results trace buffer. An overview of the design can be seen in figure 8.1.

As use case a programming exercise, building on the Sieve of Eratosthenes [24] algorithm, will be used. It computes the prime numbers up to a certain number and is well suited for practicing exploiting of parallelism. The task is to run the program with different numbers for calculation and to schedule the program on one and several cores. In all different sets of program and scheduling the number of cache misses should be measured with support from APDA. The same program will be executed on core 1 to core 7. Core 0 handles the distribution of work to the other cores by sending messages with a number that the core should use for executing the program. When receiving the message the core activates the counting of cache misses and clock cycles. After execution each processor presents the result by printing the values to the console.
8.2 Implementation

The implementation section describes what is actually implemented in terms of code and registers. Due to technical problems it has not been possible to configure the ADPA and that part is omitted from this section. Section 8.2.1 describes how the P4080 processor should be configured for access to the APDA and even if it is not working it is left in the report as a reference for further work. However the configuration of the e500mc cores works and thereby it is possible to perform the measurements and print the result from one of the cores. The problem with the APDA is further described in section 8.3.

8.2.1 P4080

The first thing that has to be done in the design is to open up a LAW for access to the DCSR memory map. Each LAW has three register used for configuration. LAWBARHn, where n is the number of the LAW, contains the four MSBs of the 36-bit address and LAWBARLn contains bit 4-23 of the 36-bit address. The last 12 bits do not need to be taken care of since the minimum size of a LAW is 4kB. In the third register, LAWARN, the LAW can be enabled and this register is also used for specification of the target. The configuration of the LAWs are done in the CCSR register map, which is actually also accessed through a LAW. To access the DCSR
8.2. IMPLEMENTATION

register map a LAW needs to be opened with the DCSR as target and a specified address. Note that the size of the LAW for the DCSR must be 4 MB. The following example shows how to open up LAW15 for access to the DCSR:

```c
/* **************************************************
 CCSR base address is 0xe0000000
 LAW15 is located at CCSR + 0xcf0
 DCSR base address will be located at 0x90000000
 DCSR target id is 0x1d
 DCSR size is 0x15
 ************************************************** */

*((U32 *)(0xe0000000 + 0xcf0)) = 0x0;
*((U32 *)(0xe0000000 + 0xcf4)) = 0x90000000;
*((U32 *)(0xe0000000 + 0xcf8)) = 0x81d00015;
}
```

Figure 8.2. Code to open up LAW15 for access to DCSR.

LAW15 is now opened for access to the DCSR with the base address 0x90000000. This means that e.g. a read from address 0x90000000 will read the first address in the DCSR memory map.

Figure 8.3. The memory map of the P4080 after configuration for access to DCSR.
8.2.2 e500mc

The e500mc have two special instructions for accessing the PMRs; \texttt{mfpmr} and \texttt{mtpmr}. To use the instructions in OSE inline assembler is used. The assembly function is then called through a bios call, described in 8.2.3. The inline assembler functions is depicted in figure 8.5 and the instructions in 8.4.

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move from Performance Monitor Register</td>
<td>mtpmr</td>
<td>rD,PMRN</td>
</tr>
<tr>
<td>Move to Performance Monitor Register</td>
<td>mtpmr</td>
<td>PMRN,rS</td>
</tr>
</tbody>
</table>

Figure 8.4. Assembly instructions for accessing the performance counters.

//Get values in PMR regs
#define get_pmr(pmr_nr)
    __extension__(
        U32 __val;
        __asm__ __volatile__(
            " mfpmr %0," STRSTR(pmr_nr) :
            "=r"(__val) :
        __val;));

//Set values in PMR regs
#define set_pmr(pmr_nr, val)
    __asm__ __volatile__(
            " mtpmr " STRSTR(pmr_nr) ",%0" :
        /* no output */ : "r" (val))

Figure 8.5. Inline assembler functions for accessing the PMRs.

8.2.3 Enea OSE

Processes

Two processes have been developed in OSE. One of them, \texttt{perfCount}, handles the performance counters by accessing the inline assembler functions described in section 8.2.2. To access the PMRs a bios call must be performed in OSE. A bios call can access resources on supervisor-level and is described in the next section. The other process, \texttt{sieve}, performs the calculation of prime numbers. An overview of the processes is depicted under Tests, section 8.3, and the code for the processes can be found in appendix A.
8.2. IMPLEMENTATION

Bios functions

Sometimes applications running in user mode need to access resources that are only accessible from supervisor mode. In that case bios functions can be used. A bios function switches from user to supervisor in both OSE and on the hardware for a special piece of code. The following steps shows how to implement a bios function in OSE.

1. At initialization of the system a bios function is installed by calling the biosInstall function. biosInstall decides permissions, which function to install and what name to search for when accessing the function.

   ```c
   biosInstall("USER_GET_SPR", (BiosFunction *) biosmodule_get_spr, BIOS_USERMODE_ALLOWED);
   ```

   Figure 8.6. Code for installing a bios function.

2. The first time the bios function is accessed the biosOpen function needs to be called. biosOpen searches for a bios function (in this case USER_GET_SPR) and returns a handle, which can be seen as a path to the bios function. biosCall can than be called with the handle as one of the arguments.

   ```c
   unsigned long usermode_get_spr(unsigned long spr_no)
   {
       static unsigned long handle = 0;
       // First time, fetch the function handle.
       if(handle == 0)
       {
           handle = biosOpen("USER_GET_SPR");
           // Handle the case that the function did not exist.
           if(handle == 0)
           {
               printf("Error in biosOpen\n");
               error(0x19);
               return 0;
           }
       }
       // Get the wanted register value
       return biosCall(handle, spr_no);
   }
   ```

   Figure 8.7. biosOpen searches for the bios function the first time it is called.
3. A *biosCall* is done every time the user wants to access the desired resource and this is needed when wanting to access the PMRs for configuration.

```c
usermode_get_pmr(PMR_PMCO);
```

*Figure 8.8. Code for reading the value in PMR_PMCO.*

### 8.3 Tests

The intention with the use case was to evaluate the impact of having a software based debugging tool connected to OSE and if the APDA is a suitable solution for multi-core platforms. But due to technical problems with the configuration of LAWs and access to the DCSR memory map it has not been possible to use the APDA in this thesis. Lots of time has been spent on solving the problem but unfortunately it has not been possible. A support case has been started at Enea to provide a working configuration for further work. Due to the time spent on the technical problems it has not been possible to do an evaluation for the debugging tool. Neither has it been possible to explore the parallelism in OSE, instead that is suggested as further work. This has also made the purpose of using a program such as Sieve of Eratosthenes to disappear. But it has still been used and can work as an idea for further work. The configuration of th APDA does not work. However, the configuration of the performance counters for each processor core works and tests have been performed for them. The tests have been simplified to just give an overview of how they work. The result from the tests are not verified but discussed briefly in section 8.4.

#### 8.3.1 Test 1

In test 1 the performance counters are first started and than immediately stopped. The purpose to measure the setup time for the performance counters. From table 5.1 REQ-01 and REQ-02 are tested.
8.3. TESTS

Figure 8.9. Flowchart for test 1.

```c
//Init counters
init_counters();

//Run all counters
usermode_set_pmr(PMR_PMGC0, 0x0);

//Stop all counters
usermode_set_pmr(PMR_PMGC0, 0x80000000);

//Print counter values
print_performance_counters();
```

Figure 8.10. Code for Test 1.
8.3.2 Test 2

In the second test an empty for-loop is introduced for measurement. The for-loop executes 10 million times and during the execution OSE will interrupt for performing OSE services. During the interrupts OSE can execute the RMM and therefore the program is running in two different modes; one with Optima active and one with Optima inactive. The purpose is to investigate the impact of the extra software overhead used when the RMM is executed. Test 2 tests the same requirements from table 5.1 as test 1; REQ-01 and REQ-02.

Figure 8.11. Flowchart for test 2.
8.3. TESTS

```c
//Init counters
init_counters();

//Run all counters
usermode_set_pmr(PMR_PMGC0, 0x0);

//Empty for-loop (test 2)
for(k=0; k<10000000; k++);

//Stop all counters
usermode_set_pmr(PMR_PMGC0, 0x80000000);

//Print counter values
print_performance_counters();
```

Figure 8.12. Code for Test 2. The program is running an empty for-loop 10 million times.
8.3.3 Test 3

The third test is similar to the second but instead of a for-loop the process sends a message to another process, which is calculating prime numbers according to the Sieve of Eratosthenes algorithm. The message sent contains the numbers up to which the second process should calculate primes numbers. The purpose is the same as in test 2 but also includes message passing. Test 3 tests the same requirements from table 5.1 as test 1 and test 2; REQ-01 and REQ-02.

![Flowchart for test 3.](image)

Figure 8.13. Flowchart for test 3.
8.4 Result

As mentioned in 8.3, no verification is made of the result. This is only a brief discussion about the information gathered.

- **Test 1**

  When no code is executed in between start and stop of the counters no context switch is made in OSE. The numbers of clock cycles is the same for all cores, which is a sign of determinism. The results can be seen in table 8.1 and shows that the debugging system meets REQ-01 and REQ-02.
### Test 2

Test 2 has been executed in three different ways:

- Case 1. Program compiled as a release, meaning that software overhead for debugging should be removed. The results can be seen in Table 8.5.
- Case 2. Program compiled with debug features and Optima is connected to the target. The results can be seen in Table 8.6.
- Case 3. Program compiled with debug features but not connected to Optima. The results can be seen in Table 8.7.

The results show as expected that the number of cache misses increases when Optima is used. But the difference in clock cycles between the release version and the version running with Optima connected to the target is around 60000 of 120 millions. Due to the time limits for this thesis the reasons for this has not been possible to investigate. However, it might indicate that the numbers of cache misses do not have a big impact on the overall performance. The higher numbers of cache misses on core 0 can be explained by the fact that most of the OSE services are executed on core 0, thereby generating more misses. Test 2 also shows that the debugging system meets REQ-01 and REQ-02.

<table>
<thead>
<tr>
<th>Core</th>
<th>Clock cycles</th>
<th>L1 D misses</th>
<th>L1 I misses</th>
<th>L2 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>197</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>197</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>197</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>197</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>197</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>197</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>197</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>197</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.1. Table for test 1.
8.4. RESULT

<table>
<thead>
<tr>
<th>Core</th>
<th>Clock cycles</th>
<th>L1 D misses</th>
<th>L1 I misses</th>
<th>L2 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>120655680</td>
<td>3167</td>
<td>419</td>
<td>164</td>
</tr>
<tr>
<td>1</td>
<td>120157436</td>
<td>941</td>
<td>262</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>120158316</td>
<td>951</td>
<td>253</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>120157607</td>
<td>948</td>
<td>266</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>120156983</td>
<td>946</td>
<td>255</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>120156766</td>
<td>929</td>
<td>263</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>120157402</td>
<td>934</td>
<td>256</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>120157170</td>
<td>937</td>
<td>267</td>
<td>11</td>
</tr>
<tr>
<td>Average</td>
<td>120219670</td>
<td>1219</td>
<td>280</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 8.2. Table for Test 2 - case 1.

<table>
<thead>
<tr>
<th>Core</th>
<th>Clock cycles</th>
<th>L1 D misses</th>
<th>L1 I misses</th>
<th>L2 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>120845454</td>
<td>3325</td>
<td>560</td>
<td>226</td>
</tr>
<tr>
<td>1</td>
<td>120199750</td>
<td>1030</td>
<td>348</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>120198766</td>
<td>1023</td>
<td>351</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>120197745</td>
<td>1028</td>
<td>350</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>120200900</td>
<td>1014</td>
<td>346</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>120198250</td>
<td>1006</td>
<td>356</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>120201203</td>
<td>1023</td>
<td>361</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>120198210</td>
<td>1020</td>
<td>347</td>
<td>26</td>
</tr>
<tr>
<td>Average</td>
<td>120279933</td>
<td>1308</td>
<td>377</td>
<td>51</td>
</tr>
</tbody>
</table>

Table 8.3. Table for Test 2 - case 2.

<table>
<thead>
<tr>
<th>Core</th>
<th>Clock cycles</th>
<th>L1 D misses</th>
<th>L1 I misses</th>
<th>L2 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>120782112</td>
<td>3162</td>
<td>435</td>
<td>169</td>
</tr>
<tr>
<td>1</td>
<td>120184184</td>
<td>949</td>
<td>260</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>120182996</td>
<td>946</td>
<td>255</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>120183587</td>
<td>945</td>
<td>256</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>120184007</td>
<td>942</td>
<td>259</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>120183422</td>
<td>936</td>
<td>257</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>120183362</td>
<td>942</td>
<td>263</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>120182698</td>
<td>938</td>
<td>255</td>
<td>12</td>
</tr>
<tr>
<td>Average</td>
<td>120258296</td>
<td>1220</td>
<td>280</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 8.4. Table for Test 2 - case 3.
• **Test 3**

Test 3 has been performed in the same three ways as in Test 2.

The results from the executions are notable. It shows as expected that the number of cache misses increases when Optima is used. But at the same time the number of clock cycles decreases. As mentioned in the result for Test 2 the verification of the result was limited due to lack of time. However, it once again shows the difficulty to fully understand software running on multi-core processors. The unexpected results are an interesting phenomena and is suggested as further work. Another thing that is notable is that the numbers of cache misses is lower for the program with debugging features but not connected to Optima than for the release version. To investigate this a closer look at the difference in compiler optimizations, and how the RMM is executed, could be done. Test 3 also shows that the debugging system meets REQ-01 and REQ-02.

<table>
<thead>
<tr>
<th>Core</th>
<th>Nr to calc.</th>
<th>Clock cycles</th>
<th>L1 D misses</th>
<th>L1 I misses</th>
<th>L2 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>7650002</td>
<td>709</td>
<td>1453</td>
<td>341</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>7820927</td>
<td>325</td>
<td>1088</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
<td>7830159</td>
<td>338</td>
<td>1104</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>900</td>
<td>7849236</td>
<td>413</td>
<td>1205</td>
<td>62</td>
</tr>
<tr>
<td>4</td>
<td>1600</td>
<td>7854359</td>
<td>415</td>
<td>1109</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>2500</td>
<td>7869805</td>
<td>450</td>
<td>1083</td>
<td>113</td>
</tr>
<tr>
<td>6</td>
<td>3600</td>
<td>7894918</td>
<td>555</td>
<td>1181</td>
<td>158</td>
</tr>
<tr>
<td>7</td>
<td>4900</td>
<td>7902798</td>
<td>626</td>
<td>1083</td>
<td>225</td>
</tr>
<tr>
<td>Average</td>
<td>7834026</td>
<td>479</td>
<td>1163</td>
<td>131</td>
<td></td>
</tr>
</tbody>
</table>

**Table 8.5.** Table for Test 3 - case 1.
## 8.4. RESULT

<table>
<thead>
<tr>
<th>Core</th>
<th>Nr to calc.</th>
<th>Clock cycles</th>
<th>L1 D misses</th>
<th>L1 I misses</th>
<th>L2 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>7629704</td>
<td>863</td>
<td>1577</td>
<td>424</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>7808798</td>
<td>370</td>
<td>1198</td>
<td>44</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
<td>7819255</td>
<td>364</td>
<td>1169</td>
<td>55</td>
</tr>
<tr>
<td>3</td>
<td>900</td>
<td>7829548</td>
<td>385</td>
<td>1182</td>
<td>65</td>
</tr>
<tr>
<td>4</td>
<td>1600</td>
<td>7844893</td>
<td>411</td>
<td>1145</td>
<td>82</td>
</tr>
<tr>
<td>5</td>
<td>2500</td>
<td>7858271</td>
<td>479</td>
<td>1177</td>
<td>119</td>
</tr>
<tr>
<td>6</td>
<td>3600</td>
<td>7874722</td>
<td>537</td>
<td>1166</td>
<td>165</td>
</tr>
<tr>
<td>7</td>
<td>4900</td>
<td>7891770</td>
<td>645</td>
<td>1163</td>
<td>185</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>7819620</td>
<td>507</td>
<td>1222</td>
<td>142</td>
</tr>
</tbody>
</table>

Table 8.6. Table for Test 3 - case 2.

<table>
<thead>
<tr>
<th>Core</th>
<th>Nr to calc.</th>
<th>Clock cycles</th>
<th>L1 D misses</th>
<th>L1 I misses</th>
<th>L2 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>7646513</td>
<td>700</td>
<td>1449</td>
<td>339</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>7819910</td>
<td>328</td>
<td>1095</td>
<td>34</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
<td>7831582</td>
<td>345</td>
<td>1096</td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td>900</td>
<td>7840886</td>
<td>369</td>
<td>1105</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>1600</td>
<td>7853753</td>
<td>407</td>
<td>1083</td>
<td>70</td>
</tr>
<tr>
<td>5</td>
<td>2500</td>
<td>7870297</td>
<td>447</td>
<td>1084</td>
<td>106</td>
</tr>
<tr>
<td>6</td>
<td>3600</td>
<td>7884222</td>
<td>526</td>
<td>1080</td>
<td>157</td>
</tr>
<tr>
<td>7</td>
<td>4900</td>
<td>7903104</td>
<td>643</td>
<td>1091</td>
<td>234</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>7831283</td>
<td>471</td>
<td>1135</td>
<td>129</td>
</tr>
</tbody>
</table>

Table 8.7. Table for Test 3 - case 3.
Chapter 9

Discussion

9.1 Conclusion

This master thesis discusses debugging of parallel software for multi-core processors. The thesis is divided into two parts, where the first one, pre-study, gives an introduction to multi-core processors and why they are used in today’s embedded systems. It also discusses parallel software and how debugging of parallel software can be improved with extra hardware integrated in the chip, with the only purpose to support debugging and monitoring of the system.

The traditional development focused on increasing performance through higher clock frequency but with new requirements, mainly on efficient power consumption, new ways had to be explored. Multi-core processors seems to be the future in the development of embedded systems. It offers a high flexibility for the application designer to choose between homogeneous processors for standard applications or heterogeneous processors tailored for a specific task. But this flexibility also brings a higher level of complexity in terms of advanced interconnection networks and memory architectures. A multi-core processor also makes the debugging more complex since many components are hidden in the device without a connection to the outside world.

Writing code for parallel applications is well-known as a challenging task. Trying to optimize the threads is not sufficient since the interaction between tasks and the competing for resources heavily influence the performance. Examples of what can degrade the performance are false sharing and spending too much time on message passing between cores. Problems such as implementing an efficient communication between tasks or the risk of deadlocks can exist both for single-core and multi-core processors. In a single-core processor a multitasking operating system can be used, thus introducing a way for executing threads in parallel even if they are not executing at the same time. But all problems with software executing in parallel have a tendency to be worse when executing on a multi-core processor since it is much harder for the designer to predict when a certain step in the code is reached. For a
multi-core processor it is also possible to chose different kind of RTOSs, depending on the processor’s hardware architecture and the wanted software characteristics.

The higher level of complexity in software makes the debugging much more complicated. An useful parallel software debugging system should be able to handle different kinds of RTOSs and also be able to monitor the communication in the application and the accesses to memory. One potential big disadvantage with tools for debugging and performance analysis is the overhead created in software. The most basic method of evaluating the impact of the overhead is to compare the execution time with and without the overhead introduced in the code. But the impact of the overhead can be very different and a small overhead can heavily decrease the performance whilst a large overhead might not decrease the performance at all. Techniques exist for analyzing this impact but a better solution is to use a non-intrusive technique for collecting information.

To be able to debug and observe a system non-intrusively complex on-chip hardware could be used. This hardware is implemented as components for different targets forming a system which can be configured. The configuration is a rather complex task since it means programming on register level. Attempts have been made to define a high level language but so far no distinct standard have reached the market. One drawback with hardware based debugging is that the debug tool must be ported towards the platform’s specific debug hardware. Multi-core is still in the beginning of its lifetime and since debugging features typically are a step behind these systems could be expected to undergo heavy changes in the upcoming years.

From the pre-study seven conclusion have been listed:

1. A multi-core processor gives the designer many tools to tailor the application to special requirements, reaching a high performance with a low consumption in power and area.

2. A multi-core processor introduce the designer to new problems such as keeping efficient memory accesses and distributing the load across the platform. These problems can have severe impact of the system’s performance and thus removing the benefits with a multi-core processor.

3. The move from single-core towards multi-core has complicated an already complex task.

4. Software based debugging introduce an non-desired overhead that has a negative impact of the performance and in worst case can lead to bugs only apparent when the system is running in its intended environment.

5. A standardized, non-intrusive, hardware based debugging system can be very helpful for the designer. This is especially true for multi-core systems and systems running under hard real-time requirements.
9.1. CONCLUSION

6. The debugging system should provide access to the components integrated in the processor, e.g. processor cores, interconnection network and memory, thus providing an alternative for debugging parallel software running on the processor.

7. A standardized high level language for configuration of the hardware based debugging system is of great importance to simplify and decrease the time consumption for debugging.

The investigation performed in the pre-study phase also found eleven requirements that should be fulfilled for an efficient debugging system in a multi-core processor. The requirements can be found in table 9.1.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide access to each core’s debugging features</td>
<td>REQ-01</td>
</tr>
<tr>
<td>Provide access to each core’s cache memory statistics</td>
<td>REQ-02</td>
</tr>
<tr>
<td>Provide access to the memory on device level, i.e. shared main memory and shared cache memory</td>
<td>REQ-03</td>
</tr>
<tr>
<td>Provide access to the interconnection network</td>
<td>REQ-04</td>
</tr>
<tr>
<td>Collect information from all processor cores simultaneously</td>
<td>REQ-05</td>
</tr>
<tr>
<td>Provide support for different kinds of RTOS models, e.g AMP and SMP RTOSs</td>
<td>REQ-06</td>
</tr>
<tr>
<td>Provide support for monitoring the communication between tasks in an parallel software application</td>
<td>REQ-07</td>
</tr>
<tr>
<td>Provide support for monitoring a parallel software application’s accesses to a shared memory</td>
<td>REQ-08</td>
</tr>
<tr>
<td>Provide a non-intrusive debugging method</td>
<td>REQ-09</td>
</tr>
<tr>
<td>Provide a standardized interface for accessing the hardware based debugging system, both from an external tool and from within the multi-core processor</td>
<td>REQ-10</td>
</tr>
<tr>
<td>A high level language should exist for configuration of the hardware based debugging system</td>
<td>REQ-11</td>
</tr>
</tbody>
</table>

Table 9.1. Requirements, found in the pre-study, for a debugging system for parallel software in a multi-core processor.

In the second phase of the master thesis a design was developed to investigate if a specific hardware based debugging system could be used together with Enea’s products. The processor used was a state of the art multi-core processor, with the belonging hardware based debugging system, from Freescale. The idea with the design has been to develop an application and test the requirements listed in table 9.1. Due to technical problems it has not been possible to evaluate the entire debugging system. The debugging configuration for each core in the processor works but the configuration for the entire processor does not work. Because of this the
tests also have been simplified to only test the processor on core-level, REQ-01
and REQ-02. However the result from the tests shows some interesting facts and
emphasizing the difficulties in predicting cache performance for parallel software.
Even if it has not been possible to test the entire debugging system it is obvious
that Enea would benefit from using a hardware based debugging system together
with their real time operating system OSE.

9.2 Personal reflections

The work towards parallel systems in terms of multi-core processors and the belong-
ing software is a very interesting area. The flexibility and possible solutions have
grown heavily and offers a lot of interesting, and challenging, opportunities for the
design engineer.

It is my belief that the importance of debugging will grow in the future. To meet
this need, debugging systems based on hardware, further developed from systems
such as the one described in chapter 4, will play a key role.

During the work I have experienced lots of problems connected to multi-core
solutions. I already in the beginning, section 1.4, said that 'since learning a new
hardware platform can be a very time consuming task, the design might be delim-
ited to do an investigation for the P4080 platform'. This was also the case since
I got stuck into trying to configure the platform-level debugging system (APDA).
I spent almost four weeks on this problem without achieving any positive result.
However, I had to explore many areas of the processor platform. This has given me
many useful insights about multi-core processors and real-time operating systems.
I also had to read many of the 3500 pages of the processor’s reference manual over
and over again. This was not the only manual used and this once again confirmed
that the work of an hardware or software engineer consists of browsing manuals in
the search for relevant information.

When talking to people in the industry you often hear 'we tend to think se-
quentially and the industry is in need of engineers who can think in parallel'. I
can only confirm this. When dealing with multi-core systems and parallel software
you constantly have to ask yourself 'what happens where and when does it happen?'.

In this thesis I have focused towards hardware based debugging systems. As
argued before, I think this solution will play a key role in future embedded systems.
A great help would be if the manufacturers could agree on a hardware standard for
such systems. Then a high level language or API could be created to simplify the
work for the software engineer. I think that working towards standardized hardware
based debugging systems is an ideal task for the iFEST project.
9.3. FURTHER WORK

9.3 Further work

The area of multi-core processors, parallel software and hardware based debugging is huge and much work still has to be done within the field. Below some examples are given that could be of interest for Enea.

1. Investigate a common network traffic processing application used at Enea today and see if it could benefit from implementing the Datapath Acceleration Architecture, DPAA. Especially look at the CSI for the DPAA and what benefits it could result in.

2. Investigate the relations of cache hits/misses in different levels of cache. What is most important for the performance of the application; few misses in L1, L2 or L3? Look at how OSE is adopted to keep a good cache performance and which improvements that can be made.

3. Develop a load balancer for OSE. Take advantage of the information provided by the APDA to distribute load modules in the most efficient way. It would also be beneficial if the load balancer could change frequency or even power down cores not needed for the applications.

4. Develop a GUI in Eclipse/Optima for configuration of the APDA.

5. Investigate if Nexus is a good standard for hardware based debugging and if it is used or will be used in many hardware platforms. Evaluate if Enea should focus on Nexus and how the company could benefit from creating an API for the configuration of Nexus features.
Bibliography


Appendix A

OSE processes

A.1 perfCount

```
OS_PROCESS(perfCount)
{
    struct sieveSignal *sieveNr;
    PROCESS sieveProcess_;
    int m = 0;
    volatile int k, l = 0;
    int i, j = 0;

    // ************** Setup performance counters in e500mc **************
    // Allocate memory for signal and hunt for Sieve process
    sieveNr = alloc(sizeof(struct sieveSignal), PRINT_SIG);
    sieveNr->primeToCalc = (int)ose_cpu_id() * (int)ose_cpu_id() * 100;
    // while(hunt("Sieve", 0, &sieveProcess_, NULL) == 0) {
    // Init counters
    init_counters();

    // Run all counters
    usermode_set_pmr(PMR_PMGC0, 0x0);

    // Send message(test 3)
    // send(&sieveNr, sieveProcess_);

    // For-loop only calling delay function(test 2)
    for(k = 0; k < 10000000; k++) {

    // Receive message(test 3)
```
APPENDIX A. OSE PROCESSES

//sieveNr = receive(PRINT_SIG);

//Stop all counters
usermode_set_pmr(PMR_PMGCO, 0x80000000);

//Print counter values
print_performance_counters();

ffmpeg("Killing myself...\n");

kill_proc(current_process());
}

A.2 sieve

OS_PROCESS(sieve)
{
    PROCESS perfProcess_;  
    struct sieveSignal *sieveNr;

    ffmeg("Waiting for number to calculate.\n");
sieveNr = receive(PRINT_SIG);
    int *prime;
    int i = sieveNr->primeToCalc;

    ffmeg("Calculating Sieve of Eratosthnes for %d\n", i);

    //create prime list
    prime = (int *)malloc((i+2)*sizeof(int));
    int c1, c2, c3;

    //fill list with 0 - prime
    for(c1 = 2; c1 <= i; c1++)
    {
      prime[c1] = 0;
    }

    //set 0 and 1 as not prime
    prime[0]=1;
A.2. SIEVE

prime[1]=1;

//find primes then eliminate their multiples (0 = prime, 1 = composite)
for(c2 = 2; c2 <= (int)sqrt(i)+1; c2++){
    if(prime[c2] == 0){
        c1=c2;
        for(c3 = 2*c1; c3 <= i+1; c3 = c3+c1){
            prime[c3] = 1;
        }
    }
}

/*@ 
//print primes
for(c1 = 0; c1 < i+1; c1++){
    if(prime[c1] == 0) printf("%i\n",c1);
}
*/

while(hunt("PerfCountProcess", 0, &perfProcess_, NULL) == 0){
    send(&sieveNr, perfProcess_);
}

kill_proc(current_process());
}