Comparison of computer architectures and design of a multi-core solution for an industrial control application

Brondsema, Y.W.

Award date:
2011

Eindhoven University of Technology

Department of Electrical Engineering
Master's Thesis Report

Comparison of computer architectures and design of a multi-core solution for an industrial control application

Yoran W. Brondsema

Supervised by
Prof. M.C.W. Geilen, Eindhoven University of Technology
Micha Nelissen, Prodrive BV

Comparison of computer architectures and design of a multi-core solution for an industrial control application

Son, August 26, 2011

Course	: Graduation project
Master	: Embedded Systems
Department	: Electrical Engineering - Eindhoven University of Technology
Faculty	: Embedded Systems - Eindhoven University of Technology
Supervisors	: *For Eindhoven University of Technology:* Prof. M.C.W. Geilen
	For Prodrive B.V.: Micha Nelissen

Author : Yoran W. Brondsema
Runstraat 28
5622 AZ Eindhoven
The Netherlands

Prodrive B.V.
P.O. Box 28030
Science Park 5501
5692 EM Son
The Netherlands
☎ +31 (0)40 2676 200
🖷 +31 (0)40 2676 201
🖱 http://www.prodrive.nl
✉ info@prodrive.nl

Eindhoven University of Technology
Den Dolech 2
5612 AZ Eindhoven
The Netherlands
☎ +31 (0)40 247 47 47
🖱 http://www.tue.nl
✉ info@tue.nl

Abstract

The purpose of this study is to investigate on the implementation of an industrial control system on various computer architectures. The system under consideration consists of two applications: the host and the HPPC application. The host application is responsible for the configuration and the control of the system while the HPPC application implements the control loop, i.e. it calculates the new set points for the actuators based on the sensor data. The objective of the first part of the study was to make a comparison of the performance of the two applications on different general-purpose processors. The choice for the processors was based on three criteria. First, the owner of the system was interested in a comparison of PowerPC and Intel processors in order to help determining their long-term roadmap. Secondly, the processors were chosen such that they represent a wide spectrum in the power-performance range. Thirdly, due to practical availability, we could only effectively benchmark four processors. For the host application, several benchmarks were selected to measure the performance of the memory hierarchy, the integer performance and the floating-point performance. To investigate the performance of the HPPC application, a model that simulates its computational behaviour was used as a benchmark. The results of the performance measurements revealed that both applications achieve their highest performance on an Intel Nehalem architecture. However, non-performance related criteria like power consumption, documentation and availability favour a PowerPC processor. Therefore, the owner of the system must evaluate the importance of the different criteria in order to make a decision.

The second part of the thesis investigates the implementation of the HPPC application on a multi-core platform. The first reason for an evolution to a multi-core platform is an increase in performance demanded from the hardware as the complexity of the application increases and the timing requirements become more stringent. Secondly, there is a demand to reduce the number of processor boards necessary to implement the system. By using multi-core processors, it is possible to combine several single-core boards onto a single board. First, two different alternatives for a mapping of tasks to cores were compared. Their performance was compared based on the amount of overhead that they introduce. Experimental measurements of the latencies of the different components of the overhead indicate that one of the alternatives offers more performance than the other. Next, using a model of its computational behaviour, the scalability of the performance of the application was analysed. It shows that, assuming a perfect division of the application over the cores, it scales in a supra-linear way. Lastly, the thesis investigates the usage of barriers as a synchronization mechanism between tasks executing on separate cores. Two different implementations of a barrier were proposed and their overhead was measured for the worst-case scenario. The results of these measurements will guide the task of parallelizing the application into threads that can be executed on separate cores. In overall, the study revealed that the HPPC application has the potential to be parallelized in an efficient way.

Acknowledgements

This thesis is the result of my graduation project which completes my master in Embedded Systems at the Eindhoven University of Technology. The project was performed within Prodrive BV, a company that I got to know via the recruitment days at the university and where I worked part-time prior to this master project.

First of all, I would like to thank Prodrive for giving me the opportunity to perform my master thesis project here. I would like to thank them for providing me with the topic of this thesis. It is not easy to come up with a subject for a thesis and I want to thank Roel Loeffen for introducing me to this interesting topic.

Next, I want to thank Steven, Eric and Micha for their guidance throughout the project. When I talk to my fellow students from the master's project, I realise that the guidance and supervision at Prodrive is very good compared to some other companies. Combined with the opportunity to work on interesting thesis topics, I can conclude that Prodrive is an excellent company for a student to perform his master thesis.

I would like to thank Nikola for helping me defining the outline and the goals of my thesis, for being a guide to the system and for answering my numerous questions. I also want to thank Jeroen for providing technical help. Along with them, I want to thank Jan, Erik and John for their input during the many meetings that we had.

Also, I would very much like to thank Marc Geilen, my supervisor at the university. He dedicated much time and attention to my project. Furthermore, his extensive and detailed feedback during our regular meetings was very helpful.

I want to thank Levent very much for our friendship during the whole master's program. He has been an amazing friend and I hope that, although I am sure that it will be the case, our friendship will last even after the Eindhoven era.

Finally, I want to warmly thank my mother, my father and my sister for everything they have done for me. I know that they want the best for me and their support has been very important for me. It is comforting to have a warm home to fall upon when things go less well.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

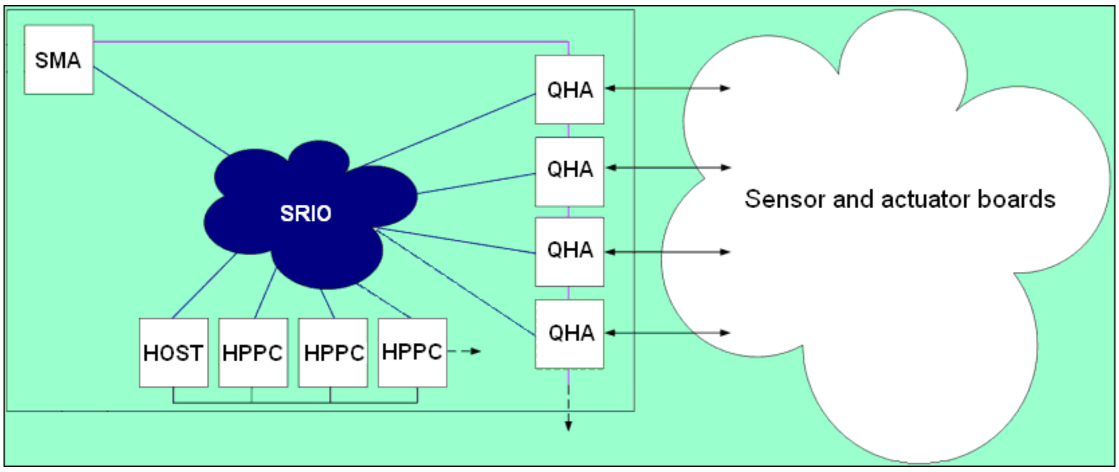


Figure 1.1: Overview of the system

One of Prodrive's clients currently has a platform that was designed to run a control loop. It is based on the Advanced Telecommunications Computing Architecture (AdvancedTCA), Serial RapidIO (SRIO) and the PowerPC architecture. An overview of the system is shown in Figure 1.1. It consists of several components that are implemented on Advanced Mezzanine Card (AMC) modules and which are interconnected with an SRIO network:

Single MoSync AMC (SMA) synchronizes the system.

Quad High Speed Serial Link AMC (QHA) are the interfaces between the system and the outside world (sensor and actuator boards).

Host configures and manages the components and the SRIO network.

High-Performance Process Controller (HPPC) processing elements that implement the control loop.

The SMA and the QHAs are implemented on dedicated Field-Programmable Gate Array (FPGA) hardware boards. The host and the HPPCs run on general-purpose single-core processor boards. At the moment, there are two motivations to upgrade the hardware platform of the host and the HPPCs to a different architecture.

performance The system that has to be controlled is becoming increasingly complex. Therefore, the application is constantly growing and an increase in performance is demanded from the hardware.

space The different components are stored as AMC modules in an AdvancedTCA shelf. Such a shelf has a limited number of slots for the AMC modules. Figure 1.2 shows a diagram of an AdvancedTCA shelf that can store 20 AMC modules. A different hardware architecture would allow to combine several modules onto a single module and therefore leave room for more modules.

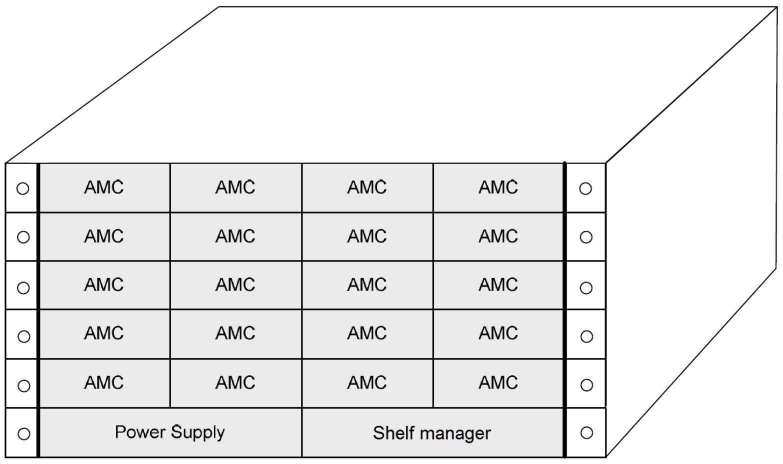


Figure 1.2: Diagram of an ATCA shelf.

1.2 Problem statement

The thesis is centred around two problems.

comparison of different processor architectures for both the host and the HPPC application. Performance is the most important criterion but non-performance related criteria dictated by the environment in which the applications must run are also taken into account in order to come to a conclusive comparison. We only perform a core-to-core comparison, i.e. for multi-core platforms we only measure the performance of a single core.

mapping the application to a multi-core system motivated by the performance and space reasons mentioned above. We consider only the HPPC application because it is a bigger performance bottleneck than the host and due to its nature it is more challenging to parallelize.

1.3 Thesis outline

Chapter 2 introduces the host and the HPPC applications, both from a software and from a hardware perspective. Chapter 3 discusses the topic of computer architectures. Factors that affect the performance are discussed and a selection of different computer architectures is presented. Furthermore, a first comparison of the architectures that are going to be benchmarked is made. Chapters 4 and 5 present the results of the comparison for respectively the host application and the HPPC application. Then, Chapter 6 discusses the mapping of the HPPC application to a multi-core platform. It compares two alternatives for a mapping of tasks to cores, investigates the scalability of the HPPC application and discusses the topic of inter-core synchronization. Chapter 7 formulates a conclusion of the thesis. Finally, Chapter 8 discusses future work arising from this project.

Chapter 2

Overview of the applications

This chapter starts with the description of the HPPC and the host application in respectively Section 2.1 and 2.2. Section 2.3 explains the relationship between the host and the HPPC application. Finally, Section 2.4 introduces the hardware platform on which the applications run. It will be described in more detail in the next chapter.

2.1 The HPPC application

Section 2.1.1 discusses the structure of the HPPC application and Section 2.1.2 explains how the application is implemented in software.

2.1.1 Structure of the application

The HPPC modules implement the control loop and do the actual computations for the control system. Each HPPC application is different as each controls a different mechatronic device; however, they all have the same structure. This structure is shown in Figure 2.1.

Each application implements a control loop that runs at a specific frequency. This means that periodically, sensor data is read, new set points are computed and sent to the actuators. The frequency of operation is defined externally by the SMA. At the start of every new execution period, the SMA sends a doorbell notification to each HPPC module via the SRIO network. During each period of execution, the following steps are performed:

post-processing of sample N The state-space of the controlled system is continuous. Since the processing system is discrete, we have to take samples of the system state by reading the sensors at discrete moments in time. At the moment of the doorbell notification that indicates the start of a new period, the sensor data that represents the current state of the system is available. With this data, the new set points destined for the actuators are computed. The computation is modelled as a computational network. Blocks represent units of computation that take an input and produce an output. Arrows between blocks define dependencies. As soon as the new set points are ready, they are sent to the actuators.

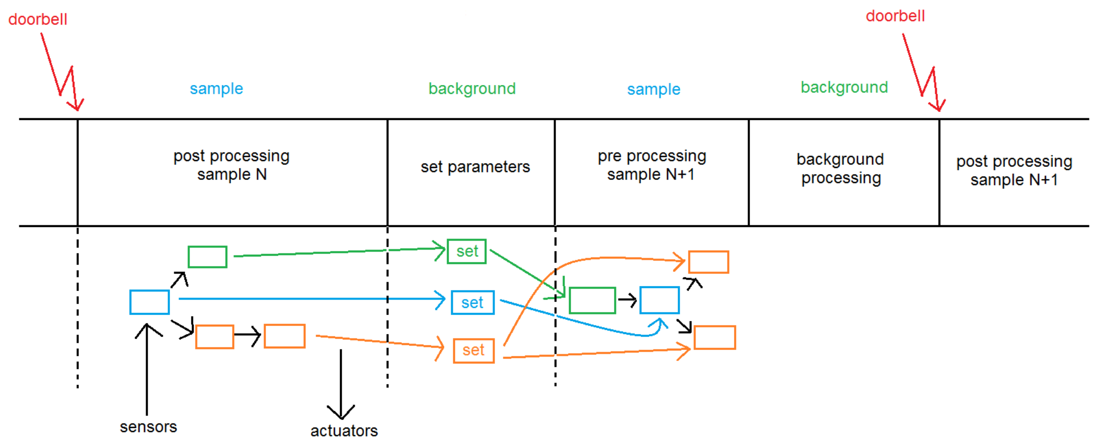


Figure 2.1: Structure of a HPPC application.

set parameters The blocks of the computational network are parametrisable. Before we start computing the next sample, we may have to update the parameters of some of these blocks. In order to avoid consistency problems, we can only update the parameters of a block when that block has finished its execution and has not started its next execution.

pre-processing of sample N+1 Before the next sensor data has arrived (corresponding to sample $N + 1$), we can already perform a pre-processing step for the next sample. The parameters of the blocks have been updated in the previous step. Just like for the post-processing step, the computation is implemented as a computational network.

background processing During the slack time between the end of the pre-processing step and the arrival of the next doorbell interrupt, non-critical background tasks are executing.

At deployment, the different HPPC applications are mapped onto physical HPPC modules.

2.1.2 Software implementation

On a software level, the loop depicted in Figure 2.1 is implemented by means of two types of tasks:

sample task This is the task that is triggered by the doorbell notification. It effectively implements the control loop because it runs the computation networks for the calculation of new set points (i.e. the *pre-processing* and *post-processing* steps).

background task These tasks serve as an interface between the host and the sample task. They fulfil two functions.

- They implement Remote Procedure Calls (RPCs) coming from the host. When the host issues an RPC call, a new background task that executes the appropriate function is spawned. The background task is only allowed to execute during the *background processing* step in Figure 2.1.

- They set the parameters of the computation blocks between the post- and pre-processing steps (see the *set parameters* step in Figure 2.1).

The HPPC software runs bare-board, meaning that it does not run a conventional operating system. It is supported by a custom library, denoted by HPPC-OS, that was built to offer the OS services needed by the application. In the first place, this was done to improve the predictability of the implementation. The requirements specify an upper bound of 5 μs on the interrupt latency of the system. The reason for the requirement is that there will not be enough time left to execute all the steps described above if the interrupt latency is longer. Experiments showed that the Linux kernel can not guarantee a jitter that is low enough to satisfy the requirement. With the bare-board-based solution, the jitter on the interrupt latency is low and the maximal latency is low enough for the application. Another advantage of a custom operating system is a reduction in overhead and memory footprint. This leaves more execution time for the actual applications running on the system.

2.2 The host application

The host application acts as a controller for the whole system shown in Figure 1.1. It consists of a large number of soft real-time tasks (about 30) that run as processes on a Linux kernel. Since there are many different processes running concurrently, there is not a clear dominating type of computation. Both floating-point and integer computations are performed, as well as communication over Ethernet and SRIO. The distribution of the load over time is bursty. This means that the average load is low but it can become very high at specific moments.

2.3 Relationship between the host and the HPPC application

Figure 2.2 shows the relationship between the host and the HPPCs. There are 11 HPPC applications in total, each mapped onto one physical AMC module. The host configures and manages the different components of the system, including the HPPCs. Due to its size, the host application is divided over three physical AMC boards. The HPPCs are connected to the SRIO network in order to communicate with the sensors and the actuators. The communication between the host and the HPPCs happens only over the Ethernet bus by means of RPC calls.

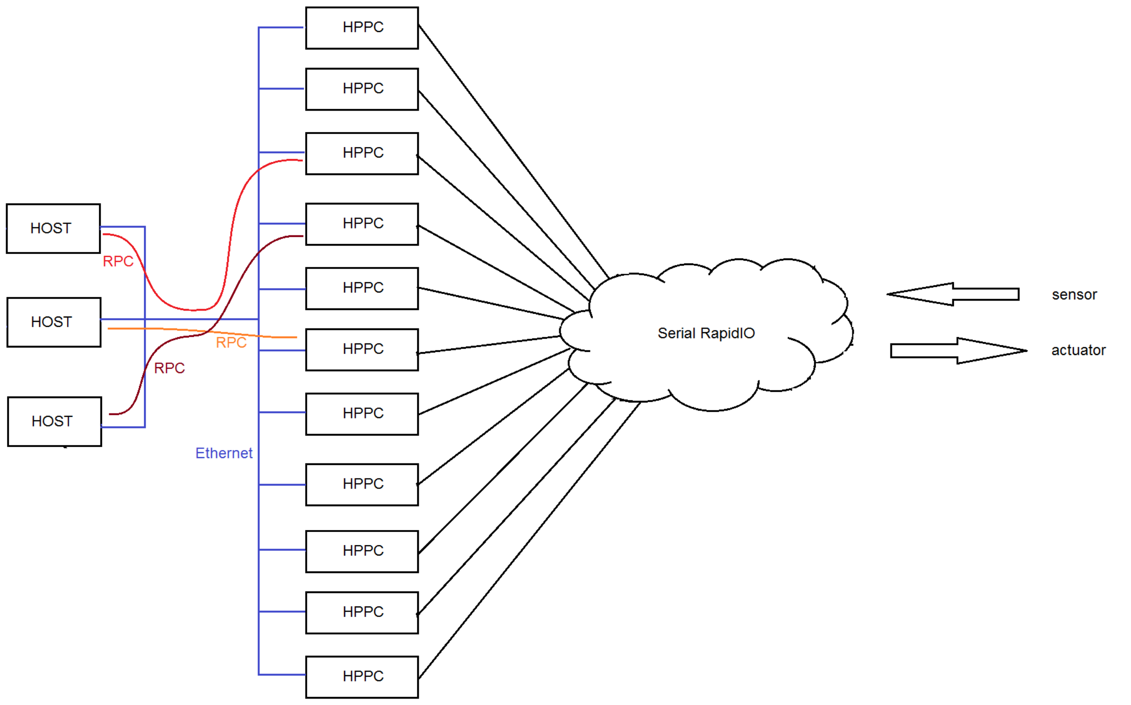


Figure 2.2: Relationship between the host and the HPPC modules.

2.4 Hardware implementation

Both the host and the HPPC run on the same type of AMC modules. The processing component of the board is the Freescale MPC8548 System-On-a-Chip (SoC) [1]. This processor architecture will be discussed in more detail in the next chapter.

Chapter 3

Description of the architectures

This chapter discusses different computer architectures. In Section 3.1, important factors that influence the performance of an architecture are described. A selection of candidate hardware architectures is presented in Section 3.2. Then in Section 3.3 the actual architectures that are going to be benchmarked are presented and a first comparison is made in Section 3.4.

3.1 Factors of performance

Without assuming anything about an application, we can state characteristics of microprocessor architectures that influence the performance of any application. An important measure that allows us to assess the impact of a factor quantitatively is the Cycles Per Instruction (CPI) metric. Any processor architecture has a fixed Instruction Set Architecture (ISA) which defines the set of native commands implemented by the processor. Let $Instr$ be the set of instructions in the ISA. Then the CPI is defined as the average latency (in clock cycles) per instruction.

$$CPI = \frac{1}{|Instr|} \sum_{i \in Instr} \text{clock cycles}(i)$$

A lower CPI indicates a better architecture in overall. Techniques that are used to optimize computer architectures affect the CPI in some way. However, there are limits to the usage of the CPI as a measure of performance. First, not all instructions are equally likely to be present in an application. For example, load and store instructions are much more common than integer divide instructions. Secondly, the performance of an architecture is also dependent on the application that is running. Some architectures are tailored for a particular kind of application. Therefore, the CPI measure should not be taken as an absolute metric for the comparison of architectures but it can be used to quantify the effect of an optimization technique for an architecture.

To define a processor architecture, the following aspects are important and they will be discussed in the next paragraphs.

- clock frequency
- memory hierarchy

- exploitation of instruction-level parallelism
- exploitation of data-level parallelism
- simultaneous multithreading
- multiple execution cores

3.1.1 Clock frequency

The clock frequency defines the operating speed of the processor. Therefore, it determines the speed at which the instructions are executed. An instruction can have the same latency on two different processors in terms of clock cycles but it will execute faster on the processor that has a higher clock frequency. However, the clock frequency is, strictly speaking, not a property of a computer architecture. Instead, it is mainly defined by the Complementary Metal-Oxide-Semiconductor (CMOS) technology that is used to implement the processor. For this reason, we will normalize the results of the measurements performed in this thesis in order to discard the effect of the clock frequency.

3.1.2 Memory hierarchy

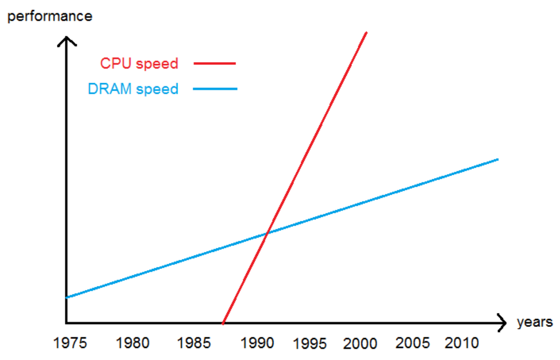


Figure 3.1: The disparity in speed between CPUs and DRAM modules causes the memory wall.

The memory hierarchy is an important factor in the performance. In the middle of the 1980s, Central Processing Units (CPUs) became faster than Dynamic Random-Access Memory (DRAM) memories and the difference in speed has kept increasing ever since (see Figure 3.1). The disparity in operating speed between the CPU and the DRAM is called the memory wall. The main consequence is that the CPU wastes hundreds of cycles when it has to wait for data from the main memory. This causes very big stalls. Modern superscalar processors try to hide this latency by exploiting instruction-level parallelism (ILP) or by applying techniques such as out-of-order execution. These mechanisms enable the simultaneous execution of independent instructions. Therefore, when the processor is stalling on a memory access, instructions that do not depend on this memory access can be executed concurrently. However, insurmountable data and control dependences put a fundamental limitation on these methods. That is why caches have become increasingly important.

Caches exploit the spatial and temporal locality of memory accesses by temporarily storing data that is prone to be used again soon in a faster, smaller memory that is close to the CPU and that can be accessed within a few clock cycles. Most processors have multiple levels of cache; the caches close to the processor are fast but have a small capacity while the levels higher up have a greater capacity but also have longer access times. The average latency for instructions that perform a memory access is reduced by the presence of multiple cache levels. Therefore, cache levels have a positive effect on the CPI.

When the CPU loads data from memory, the cache checks whether it has a copy of the data. If that is the case, we speak of a cache hit. Otherwise, the result of the operation is a cache miss and the data will have to be fetched from either the next cache in the memory hierarchy or from main memory. The performance of a cache is indicated by its hit rate, i.e. the percentage of cache hits compared to the total number of memory accesses. The following properties of a cache are important for its performance:

cache size A larger cache can store more items and therefore produces more hits.

cache line size A cache stores data as blocks of a certain size (typically 32 bytes or 64 bytes). A larger cache line size is therefore good for the performance of sequential access as more data can be loaded from the cache before having to fetch the next cache line.

associativity The replacement policy decides where in the cache a copy of an entry of main memory will go. If the replacement policy is free to choose any entry in the cache to hold the copy, the cache is called fully associative. At the other extreme, if each entry in main memory can go in just one place in the cache, the cache is direct mapped. A fully associative cache has the lowest miss rate but it requires to search the entire cache when looking for an entry. Therefore, implementations make a compromise; if an entry can go to any of N places in the cache, then that cache is N-associative.

3.1.3 Instruction-level parallelism

In order to increase the number of executed instructions per cycle, microprocessor designers have tried to exploit ILP. This technique consists of executing instructions of a same thread in parallel. Not all instructions can be executed in parallel; they must be independent, i.e. there must not be a data dependency or a control dependency between them. The effect of the exploitation of ILP is a reduction of the CPI by a factor.

$$\text{CPI} = \frac{\text{CPI}_{\text{base}}}{\text{exploited ILP}}$$

The main goal of ILP is to keep the instruction pipeline running. When the processor is stalling on a high-latency instruction, for instance a memory load, ILP allows it to execute independent instructions in parallel. Therefore, ILP is also a solution for the memory wall. In order to be able to execute instructions in parallel, a CPU that exploits ILP needs

- Multiple execution units that can execute in parallel.

- The ability to look ahead in the instruction flow and make a distinction between instructions that depend on the outcome of the current instruction and independent instructions. For most architectures that exploit ILP, this is done by the hardware at run-time. These processors are then called superscalar processors.
- Support of out-of-order execution: the execution of the instructions does not have to follow the instruction flow but the results have to be written back in program order.

For these reasons, exploitation of ILP does not come for free but requires complex hardware that adds up to the power consumption of the processor.

3.1.4 SIMD architectures

When a same operation has to be performed on many independent pieces of data, these computations can be executed in parallel. This type of parallelism is called data-level parallelism. A typical field where this is heavily used is computer graphics. Many algorithms from the graphics domain consist of doing the same computations on all pixels. In Flynn's taxonomy [2], an architecture that exploits this type of parallelism is a Single Instruction, Multiple Data (SIMD) architecture.

These architectures provide instructions that perform an operation on a vector of input data instead of on a scalar. What needs to be specified in a loop on a scalar processor can be stated with a single instruction on an SIMD processor. Graphical Processing Units (GPUs) fall in this category. Due to the field of application they were designed for, they have a vast amount of floating-point units that can be used in parallel. This allows them to perform an operation on many data inputs simultaneously.

3.1.5 Simultaneous multithreading

Simultaneous Multithreading (SMT) is a technique that improves the overall efficiency of superscalar CPUs (i.e. processors that exploit ILP dynamically) by hardware multithreading. A normal superscalar processor runs a single thread of execution at a time. Therefore, every clock cycle, it can only issue instructions from a single thread. Due to dependencies between instructions, an application has only a limited amount of ILP. Therefore, the idea of hardware multithreading is that the processor can issue instructions of several threads simultaneously, instead of considering only one thread. If a thread is waiting on data from the main memory and thus has to stall, instructions from another thread can be executed instead. The mechanism is shown in Figure 3.2. Note that the overall CPI of the processor increases by the use of SMT but the CPI of an individual thread actually decreases because it has to share the processor with another thread.

3.1.6 Multi-core architectures

Multiprocessor systems exploit another type of parallelism called task parallelism. It focuses on the distribution of execution threads across different parallel computing nodes. Compared to instruction-level and data parallelism, the granularity of task parallelism is much higher.

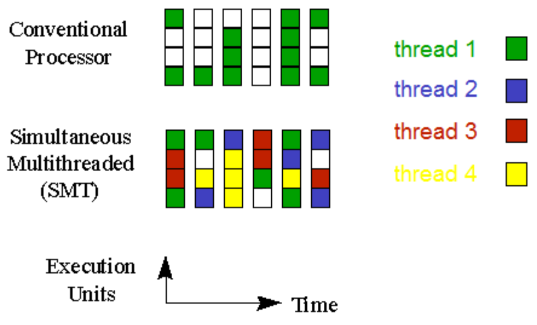


Figure 3.2: Simultaneous multithreading enables parallel scheduling of instructions from several threads [3].

The most widely used multi-core architecture is the Symmetric Multiprocessing (SMP) architecture. There, multiple identical processors are connected to a single shared main memory. A network connects the cores with each other and with the memory. Figure 3.3 shows a diagram of a typical SMP system.

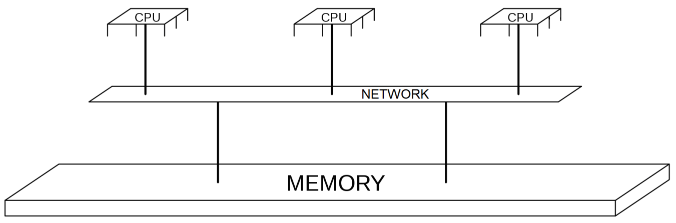


Figure 3.3: Diagram of an SMP system [4].

3.2 Candidate hardware architectures

In this part, we will discuss several candidate hardware architectures. The processors were selected on the basis of:

- The roadmap of the company that owns the system. Currently, the system uses PowerPC processors. However, to support their decisions on the long-term roadmap, the company is interested in how well Intel processors perform on their applications compared to PowerPC processors. Therefore, processors from both Intel and PowerPC were included in the study.
- Their position on the power-performance line. A low-power processor typically also has a lower performance and vice-versa for a high-power processor. The intention is to compare processors on a wide range of this line.
- Two architectures that are optimized for specific applications: the Cell architecture and a digital signal processor. They were included for their expected performance on the applications.

The following architectures were selected.

Intel Atom Z530 low power consumption, low performance

PowerPC MPC8548 current hardware platform, mid power consumption, mid performance

PowerPC P4080 octo-core evolution of the MPC8548, mid power consumption, mid performance

PowerPC P5020 dual-core evolution of the P4080, mid power consumption, mid performance

Intel Xeon E5540 quad-core, high power consumption, high performance

Cell mix of a general-purpose and an SIMD processor, high power consumption, high performance

digital signal processor processor dedicated for Digital Signal Processor (DSP) applications, low power consumption, mid performance (for certain applications)

3.2.1 Intel Atom Z530

The Intel Atom architecture is a 32-bit single-core general-purpose processor architecture that aims to provide modest performance with a low power-consumption. It implements the x86 instruction set.

Memory hierarchy

The Z530 has two levels of cache with a 64 bytes cache line size.

L1 instruction cache It has a 32 KiB 8-way associative L1 cache dedicated for instructions.

L1 data cache It has a 24 KiB 6-way associative L1 cache dedicated for data. The motivation for this asymmetric L1 cache is power saving. For the data cache, Intel decided to use 8 transistors to store one bit instead of the conventional 6 transistors. The addition of two extra transistors allows to reduce the voltage applied to the cache resulting in a reduced power consumption.

L2 unified cache It has a 512 KiB 8-way associative unified L2 cache.

Instruction-level parallelism

The Intel Atom has a limited form of instruction-level parallelism. It follows an in-order execution flow, meaning that the execution of instructions strictly follows the program order. However, it can dispatch two instructions simultaneously. This allows for parallel execution of two instructions. This can only happen under strict conditions however, such as no dependencies between the instructions and the usage of different types of execution units.

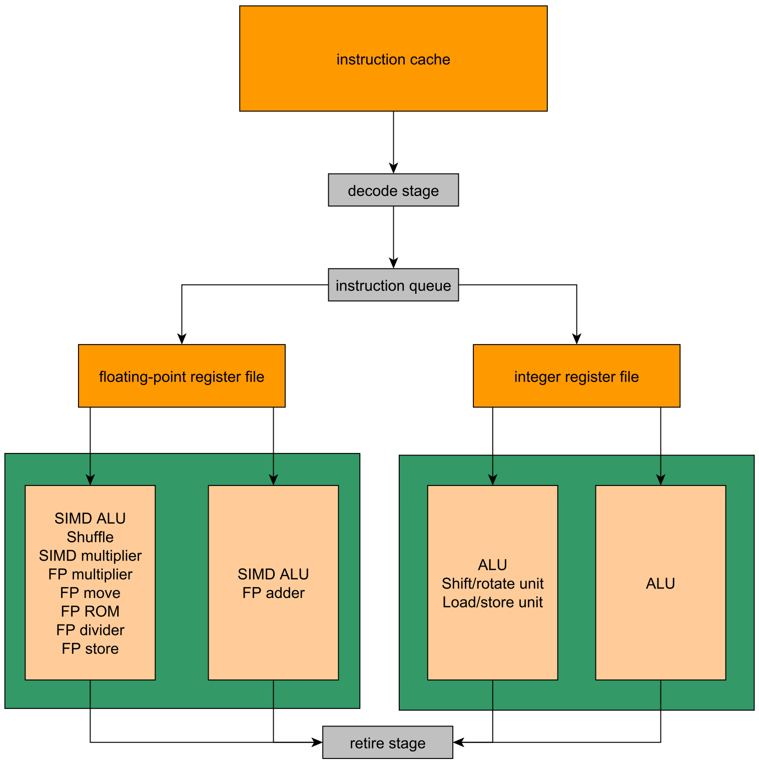


Figure 3.4: Architecture of the Intel Atom core [5].

SIMD

Figure 3.4 shows the architecture of the Intel Atom core. The execution unit consists of two clusters: a SIMD/floating-point cluster and an integer cluster. The Intel Atom supports a limited form of SIMD because the SIMD/floating-point cluster can execute instructions where multiple data items are packed in a 128-bit register. Computations are done on all data items in parallel. For instance, it is possible to do four 32-bit integer additions in parallel.

Simultaneous multithreading

The Intel Atom supports hyper-threading, which is Intel's implementation of simultaneous multithreading. The core can execute instructions from two threads simultaneously.

Strengths and weaknesses

Due to its low-power properties, the Intel Atom processor is typically used in mobile applications. In these environments, power consumption is equally or more important than the computational performance. Therefore, the Intel Atom lacks two important features for performance:

- The smaller L1 data cache (24 KiB instead of the conventional 32 KiB) will cause more L1 cache misses and in the end, will also result into more L2 cache misses.

- The in-order execution flow is a weakness for computation-intensive applications, i.e. applications that have a relatively low factor of branching.

On the other hand, the Intel Atom also has some strengths:

- it provides limited multiprocessing capabilities due to its support of SMT.
- floating-point computations can be accelerated by using the SIMD instructions.

3.2.2 PowerPC MPC8548

The MPC8548 is a SoC manufactured by Freescale Semiconductor. It contains an e500v2 core, which is a 32-bit Power Architecture-based microprocessor core.

Memory hierarchy

Just like the Intel Atom Z530, the e500v2 core has two levels of cache. One difference is that its cache line size is 32 bytes.

L1 instruction cache It has a 32 KiB 8-way associative L1 cache dedicated for instructions.

L1 data cache It has a 32 KiB 8-way associative L1 cache dedicated for data.

L2 unified cache It has a 512 KiB 8-way associative unified L2 cache.

Instruction-level parallelism

Figure 3.5 shows the architecture of the e500v2 core. The e500v2 is a true superscalar processor. It can dispatch at most 2 instructions per cycle to the various reservation stations. When an instruction has executed in its appropriate execution unit, it is sent to the completion and write-back stage. These stages maintain the correct architectural machine state and commit results to the registers in the proper order.

SIMD

Just like the Atom, the e500v2 core supports a limited form of SIMD. However, as the Atom supported packed data of 128 bits, the e500v2 supports only 64-bit data. It allows to perform arithmetic operations on two 32-bit integers that are packed in a 64-bit register.

Simultaneous multithreading

The e500v2 core does not support simultaneous multithreading. At any moment, only one thread is using the processor.

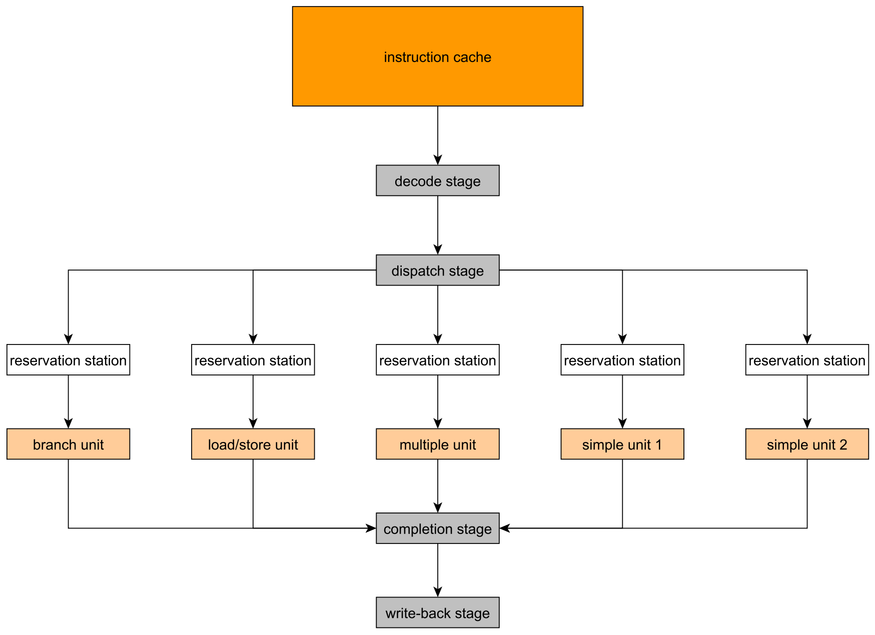


Figure 3.5: Architecture of the e500v2 core in the MPC8548 [6].

Strengths and weaknesses

The MPC8548 is quite similar to the Intel Atom Z530. One important difference is that it exploits ILP with an out-of-order execution flow. This provides advantages for applications with low branching, where a large scheduling window allows many independent instructions to be executed in parallel.

The main disadvantage of the MPC8548 is the lack of SMT. The processor executes only one thread of execution at a time. This reduces the overall throughput of the software running on the processor but it has one advantage: it increases predictability of the software. Indeed, when using SMT, the performance of a thread becomes dependent on the behaviour of the other thread that is running simultaneously. This is bad for real-time behaviour and the lack of SMT support illustrates the domain that the MPC8548 targets, i.e. mainly industrial applications that often have real-time requirements.

3.2.3 PowerPC P4080

The P4 series is the evolution of the MPC8548 SoC. The P4080 is an octo-core SoC based on the e500mc core. The architecture of the SoC is shown in Figure 3.6. The e500mc core is very similar to the e500v2 core of the MPC8548 except for a few differences.

Memory hierarchy

Each core has its private two-level cache hierarchy. Furthermore, there is an additional L3 cache that is shared among all cores. The cache line size is 64 bytes, as opposed to 32 bytes for the e500v2

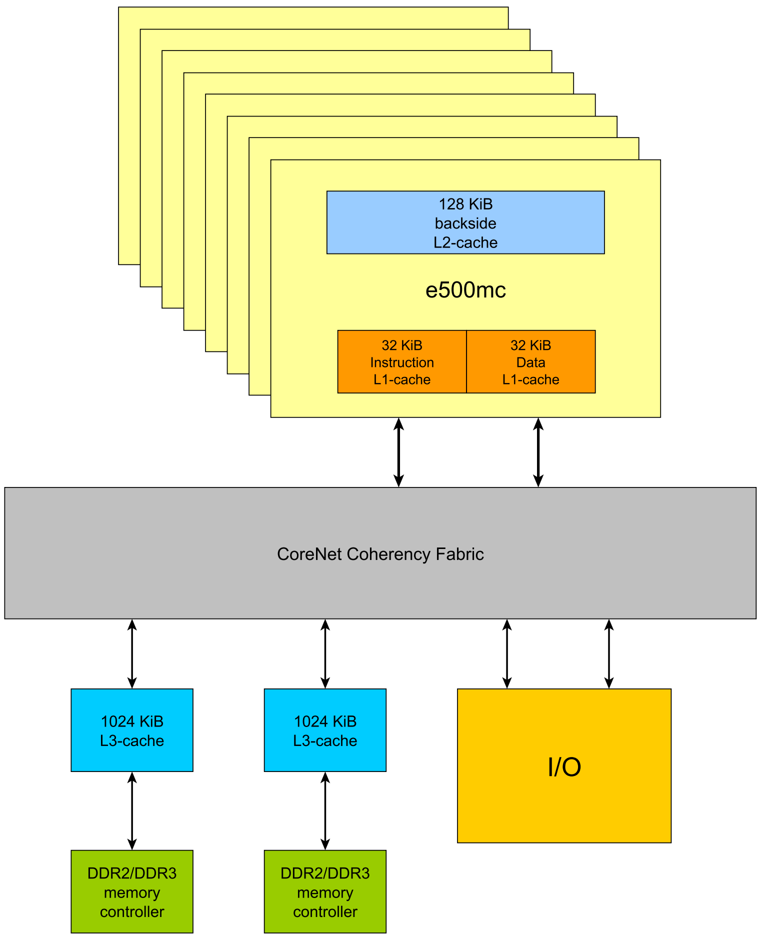


Figure 3.6: Architecture of the P4080 SoC [7].

core.

L1 instruction cache Each core has a 32 KiB 8-way associative L1 cache dedicated for instructions.

L1 data cache Each core has a 32 KiB 8-way associative L1 cache dedicated for data.

L2 unified cache Each core has a 128 KiB 8-way associative unified L2 cache.

L3 unified cache All eight cores share 2 MiB of 32-way associative unified L3 cache. The cache is split in two blocks of 1 MiB, where each one is attached to a separate memory controller. The mapping of memory address to memory controller is determined by the interleaving mechanism. Some possible configurations are cache line interleaving (meaning 64 bytes are mapped onto one controller, the next 64 bytes onto the other controller, and so on), bank interleaving or page interleaving.

Instruction-level parallelism

Just like the e500v2 core, the e500mc core exploits ILP and can execute instructions out-of-order. It can dispatch 2 instructions per cycle. When comparing Figures 3.5 and 3.7, we can notice a difference in the execution units. The ’multiple unit’ of the e500v2, which executed both integer multiplies and divides as well as floating-point instructions, has been replaced by a ’complex unit’ and a ’floating-point unit’ in the e500mc. Now the complex integer operations (multiply and divide) are executed by the ’complex unit’ while the floating-point instructions execute in a dedicated ’floating-point unit’.

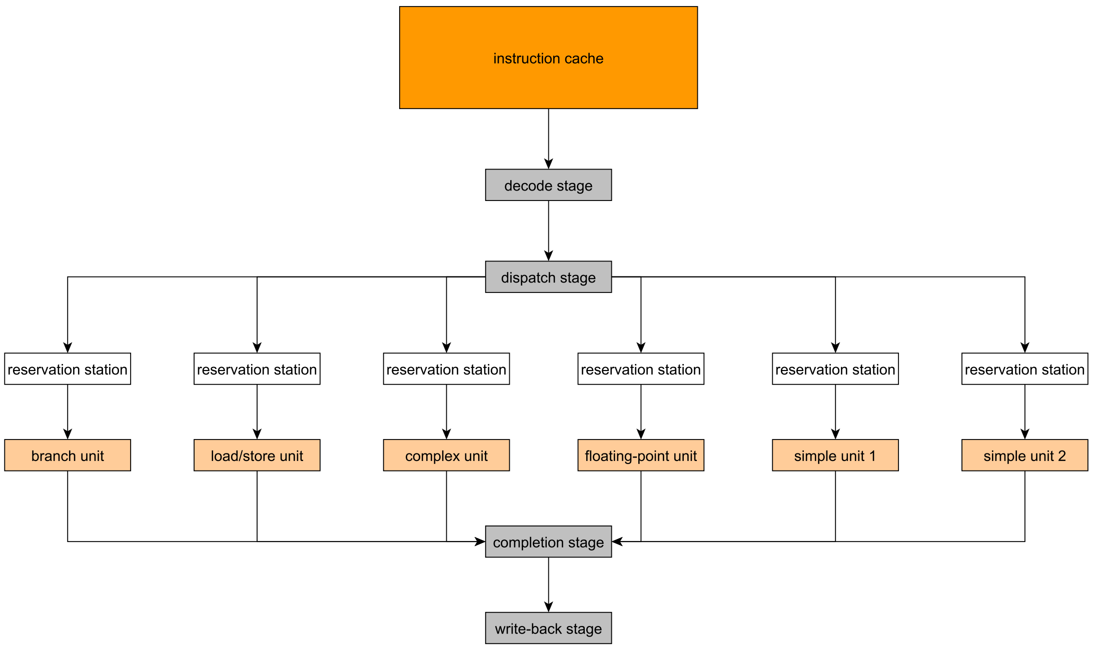


Figure 3.7: Architecture of the e500mc core in the P4080 [8].

SIMD

In contrast to the e500v2 core, the e500mc does not support SIMD instructions.

Simultaneous multithreading

Just like the e500v2 core, the e500mc does not support simultaneous multithreading.

Strengths and weaknesses

The P4080 is essentially an octo-core version of the MPC8548 except for three main differences:

- each core possesses less private cache: the L2 cache is only 128 KiB compared to 512 KiB on the MPC8548.

- it does not support SIMD instructions which makes it less suitable for applications with some form of data-parallelism.
- the 'multiple unit' on the MPC8548 has been split into two separate units on the P4080: a 'complex unit' for complex integer operations (division and multiplication) and a 'floating-point unit' for floating-point operations.

The latter can be an advantage if an application performs integer and floating-point computations in parallel. However, we do not expect this to be a big advantage in practice as an application is often divided into parts where only a single type of computation is performed in each part.

Due to the high number of cores and the relatively small private cache of each core, it is expected that the P4080 performs well for applications that can be divided into many light-weight threads where each thread has a small memory footprint. It is not suited for large sequential applications as many cache misses will degrade the performance and the available cores will be underutilized.

3.2.4 PowerPC P5020

The P5 series is also a successor to the MPC8548. The P5020 is a dual-core SoC based on the 64-bit e5500 core. Its architecture is shown in Figure 3.8. When comparing it with the architecture of the P4080 in Figure 3.6, we can notice that they have the same structure. The main difference is that the amount of cache per core is much higher on the P5020 than on the P4080.

Memory hierarchy

Just like the P4080, each e5500 core of the P5020 has two levels of cache. Also, the cache line size is 64 bytes. Furthermore, there is an additional L3 cache that is shared by both cores.

L1 instruction cache Each core has a 32 KiB 8-way associative L1 cache dedicated for instructions.

L1 data cache Each core has a 32 KiB 8-way associative L1 cache dedicated for data.

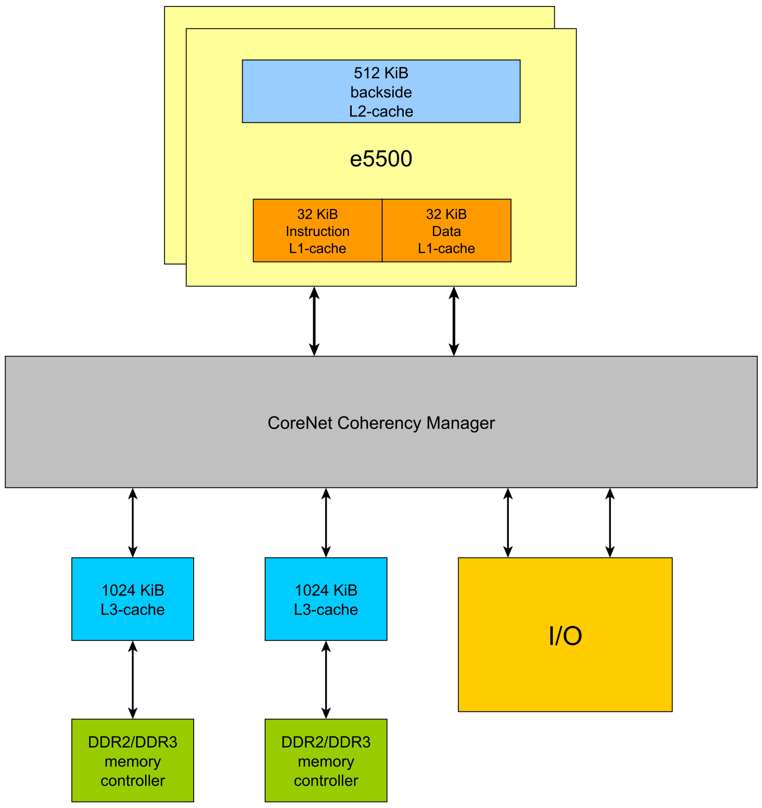
L2 unified cache Each core has a 512 KiB 8-way associative unified L2 cache.

L3 unified cache The two cores share 2 MiB of unified L3 cache with a 32-way associativity. The cache is split in two blocks of 1 MiB, where each one is attached to a separate memory controller. Just like for the P4080, the mapping of memory address to memory controller is determined by the interleaving mechanism.

On average, each core in the P5020 has 1600 KiB of cache. This is only 448 KiB per core on the P4080.

Instruction-level parallelism

As can be seen in Figure 3.9, the structure of the e5500 core is the same as the e500mc core. Also, the ILP mechanism is the same.



512 KiB backside L2-cache
e5500
32 KiB Instruction L1-cache
32 KiB Data L1-cache
CoreNet Coherency Manager
1024 KiB L3-cache
1024 KiB L3-cache
I/O
DDR2/DDR3 memory controller
DDR2/DDR3 memory controller

Figure 3.8: Architecture of the P5020 SoC [9].

SIMD

Just like the e500mc core, the e5500 does not support SIMD instructions.

Simultaneous multithreading

Just like the e500mc core, the e5500 does not support simultaneous multithreading.

Strengths and weaknesses

Whereas the P4080 targets applications that are composed of many small threads, the P5020 is more suited for applications that can be decomposed into fewer but larger threads. Its main strength compared to the other processors is that each core has a very large cache: on average, each core has 1312 KiB cache of which 576 KiB is private.

3.2.5 Intel Xeon E5540

The Intel Xeon 5500 series are microprocessors based on the Nehalem-EP architecture. The processor we consider is the E5540 model. This is a quad-core multiprocessor. The architecture of the

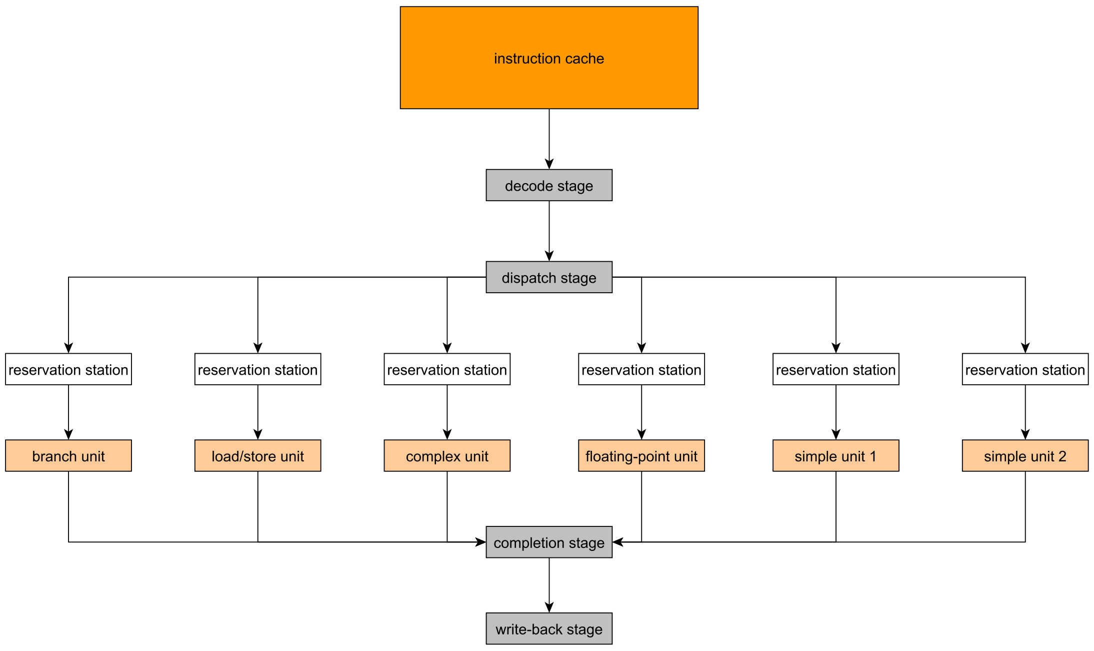


Figure 3.9: Architecture of the e5500 core in the P5020 [10].

processor is shown in Figure 3.10.

Memory hierarchy

Each core has two levels of cache, with a cache line size of 64 bytes. Furthermore, there is an additional L3 cache that is shared by both cores.

L1 instruction cache Each core has a 32 KiB 4-way associative L1 cache dedicated for instructions.

L1 data cache Each core has a 32 KiB 8-way associative L1 cache dedicated for data.

L2 unified cache Each core has a 256 KiB 8-way associative unified L2 cache.

L3 unified cache The cores share 8 MiB of 16-way associative L3 cache.

Instruction-level parallelism

The Nehalem-EP architecture is a superscalar architecture. This means that it exploits instruction-level parallelism. Its execution pipeline is shown in Figure 3.11. When comparing it with the pipeline of the e5500 (see Figure 3.9), we can notice several differences. First of all, the reservation station for the execution units is unified, i.e. there is one queue for all execution units instead of a separate queue for each execution unit. Secondly, the computational operations are divided among the execution units in a different way. In the PowerPC architectures, the division is 'cleaner': each execution

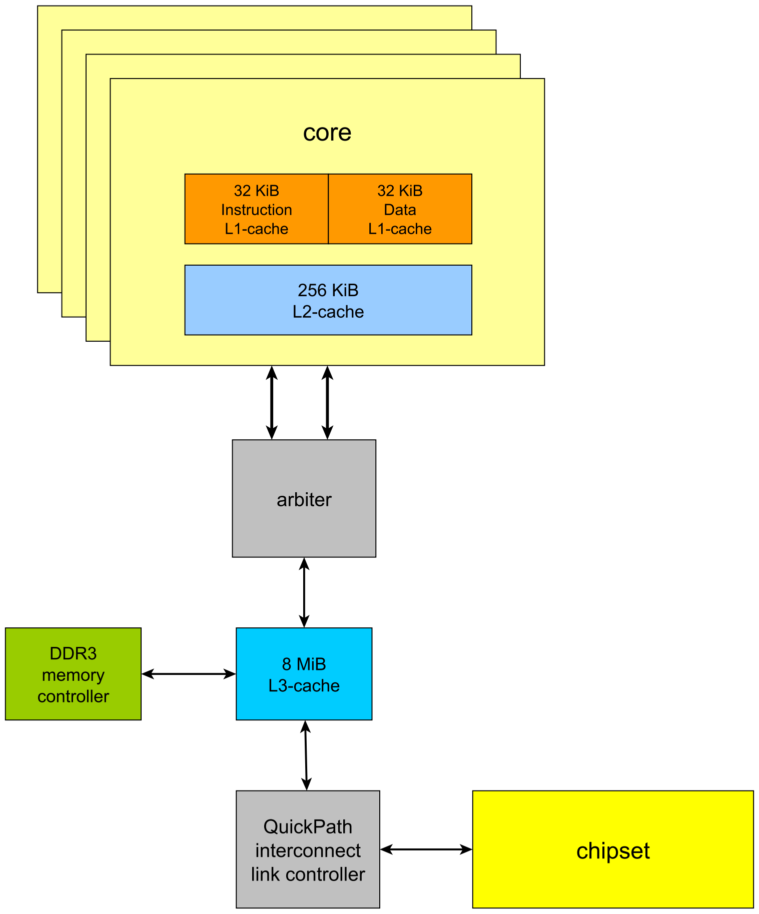


Figure 3.10: Architecture of the Intel Xeon E5540 processor [11].

unit has a clear function. For instance, the 'floating-point unit' solely executes floating-point instructions. In the Nehalem architecture, this distinction is much less clear. For instance, the floating-point operations are divided over three different execution units. The reason is performance as different instructions that belong to the same domain (for instance floating-point operations) can be scheduled on different execution units, therefore allowing parallel execution. The downside is duplication of hardware as similar logic must be implemented in multiple execution units. The following properties are characteristic for the Xeon execution pipeline:

- All execution units that perform computations (i.e. not the load and store units) perform both integer and floating-point computations.
- There is no separate branching unit but it is integrated in one of the execution units.
- There is a separate execution unit for loading data, storing addresses and storing data. These three operations can be executed in parallel.

Another architectural difference is that the decoding stage is much more complex in the Nehalem architecture. The PowerPC ISA follows a Reduced Instruction Set Computing (RISC) architecture. However, the x86 instruction set is a Complex Instruction Set Computing (CISC) architecture. For performance reasons, the x86 instructions (called macro-ops) are converted to a sequence of smaller RISC-instructions (called μ-ops) that are executed instead. This conversion happens during the decoding stage.

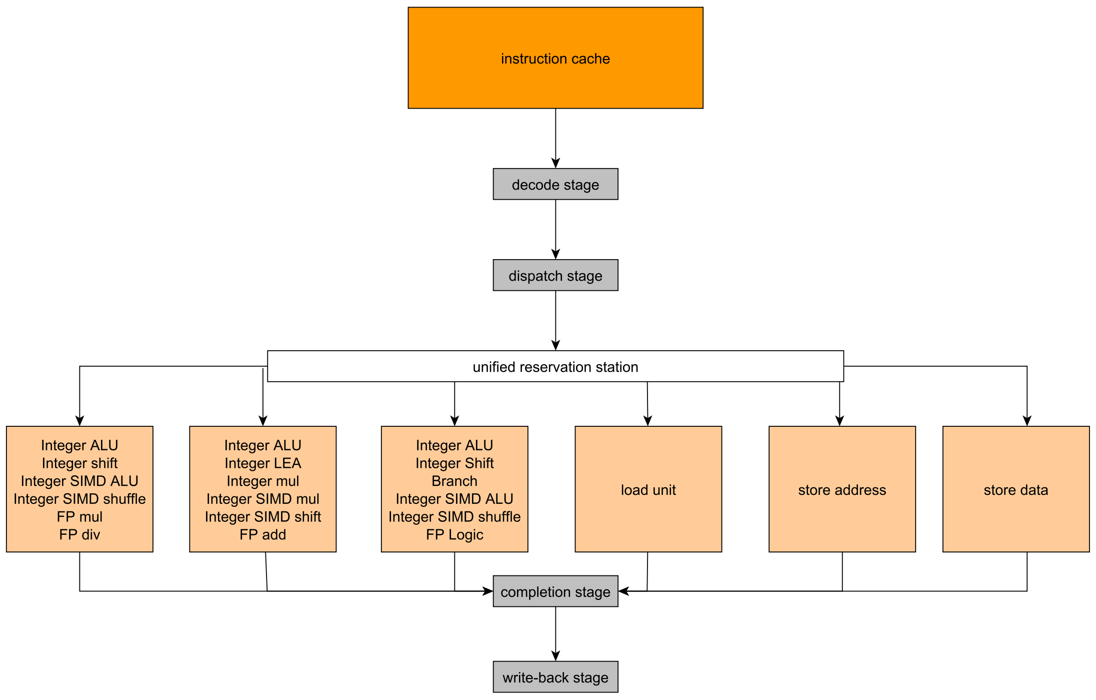


Figure 3.11: Architecture of the Intel Xeon E5540 execution pipeline [11].

SIMD

Nehalem supports the Streaming SIMD Extensions (SSE) instruction set. Just like the Atom architecture, it has support for data items that are packed in 128-bit registers. Then it is possible to do simultaneous operations on two double-precision 64-bit floating-point elements or four single-precision floating-point elements.

Simultaneous multithreading

As the Intel Atom, the Xeon processor supports SMT through hyper-threading. A core can execute instructions from two thread simultaneously.

Strengths and weaknesses

The Nehalem architecture is often used for server applications, for instance to host a database or a web server. This has several reasons:

- the total amount of cache in the processor is high which is important to speed up memory-intensive applications.

- the memory performance is expected to be good because the conventional load/store unit is implemented as three separate execution units.

- due to its four cores and support of SMT, the degree of multiprocessing that can achieved is high. This allows it to handle multiple server requests concurrently.

Moreover, we also expect it to perform well computation-wise. For instance, the division of the floating-point operations over the execution units (multiplication and division are on one unit, the addition is on another and floating-point logic is on a third one) combined with an out-of-order execution flow allows it to achieve a high performance on floating-point applications.

The biggest weakness of the Xeon processor is its high power consumption. This can become a problem when the cooling capacity of the environment is limited.

3.2.6 Cell processor

Cell is a multiprocessor architecture jointly developed by IBM, Sony and Toshiba. It combines a general-purpose PowerPC host processor, called the Power Processing Element (PPE), with 8 co-processors, called Synergistic Processing Elements (SPEs). In an application, the programmer can indicate parts of the program that should execute on an SPE. SPEs are fast SIMD processors and can therefore perform integer and floating-point operations much faster than the PPE. The basic idea is that the PPE is used for the control flow of the application while the SPEs are used to accelerate certain portions of the application. The Cell architecture is shown in Figure 3.12.

Memory hierarchy

As can be seen in Figure 3.12, the PPE has two levels of cache with a cache line size of 128 bytes.

L1 instruction cache It has a 32 KiB 2-way associative L1 cache dedicated for instructions.

L1 data cache It has a 32 KiB 4-way associative L1 cache dedicated for data.

L2 unified cache It has a 512 KiB 8-way unified L2 cache.

An SPE has no hardware-controlled cache but instead has a fast 256 KiB Static Random-Access Memory (SRAM) scratchpad. The motivation is to reduce hardware complexity and increase performance. However, it also makes it harder to program because the task of managing the memory hierarchy is assigned to the programmer.

Instruction-level parallelism

Neither the PPE nor the SPEs exploit ILP. Both have an in-order execution flow. The main motivation is power consumption because, as was mentioned before, out-of-order execution requires complex hardware.

The execution pipeline of the PPE is shown in Figure 3.13. It features a load/store unit, a fixed-point unit and a branch unit in one cluster and two AltiVec (SIMD) and two floating-point units in the other cluster. It can issue up to two instructions per cycle, i.e. one instruction per cluster.

The execution pipeline of the SPE is shown in Figure 3.14. It consists of 7 execution units that are organized into an even and an odd pipe. Just like the PPE, it can issue an instruction in both pipes every cycle.

SIMD

Just like the Intel Xeon and the Intel Atom, the PPE supports a limited form of SIMD. It supports AltiVec instructions, which is the PowerPC variant of the SSE instruction set for x86. This means that it can operate on data items that are packed in 128-bit registers.

The SPE is an SIMD architecture. It has one register file that contains 128 128-bit registers. Each instruction can read up to three 128-bit source operands and write back one 128-bit result. Alternatively, it can also do operations on single scalar 128-bit elements.

Simultaneous multithreading

The PPE supports SMT in the same way the Intel processors support it with hyper-threading. The SPE is single-threaded however.

Strengths and weaknesses

Originally, Cell is an architecture for high-performance distributed computing. It targets applications that are computationally intensive, often data-parallel and that can be divided into threads that can execute concurrently. The SPEs must be seen as units that can accelerate the execution of certain portions of the application.

For these reasons, the Cell is not suited for applications that have a complicated control flow, i.e. with many branches, and that have few parts that require heavy computations. Such an application would run on the PPE only and would leave the SPEs unutilised.

One disadvantage of the Cell compared to other high-performance processors such as the Intel Xeon E5540 is that it requires a lot of effort from the programmer to achieve a high performance on the platform. For instance, the SPEs have scratchpad memories instead of hardware controlled caches. Therefore, the programmer must manage the memory in an efficient way. However, if the application is programmed wisely, the fast SIMD computations, the fast scratchpad memories and the high-bandwidth interconnect of the Cell architecture can provide a very high performance.

3.2.7 Digital signal processor

Compared to the general-purpose processors that were discussed before, DSPs relinquish flexibility for more efficiency. They are targeted towards a specific application domain, i.e. digital signal processing applications, which makes them more performant in that domain than a general-purpose CPU but less efficient in other domains.

They have special hardware features that allow them to perform DSP operations in an efficient way. Typical techniques are

- fast multiply-accumulate units
- memory architectures that support several accesses per instruction cycle (modified Harvard architecture)
- special addressing modes such as bit-reversed addressing (for instance to calculate FFTs) or modulo addressing
- special loop controls that reduce the loop overhead

However, they often lack units that increase the flexibility of the processor. For instance, DSPs often do not have a memory management unit. They have no support for virtual memory or memory protection because it increases the context switching latency. Also, they often have limited I/O capabilities. Because a DSP has an architecture targeted to provide acceleration for certain parts of an application, there is limited support of operating systems for DSPs. Also, like the Cell, it is required that the application is specially optimized by hand in order to take advantage of the features of a DSP. Therefore, porting an application is very time-consuming.

As a conclusion, a DSP is most often used as a co-processor to accelerate a certain portion of an application and rarely used stand-alone.

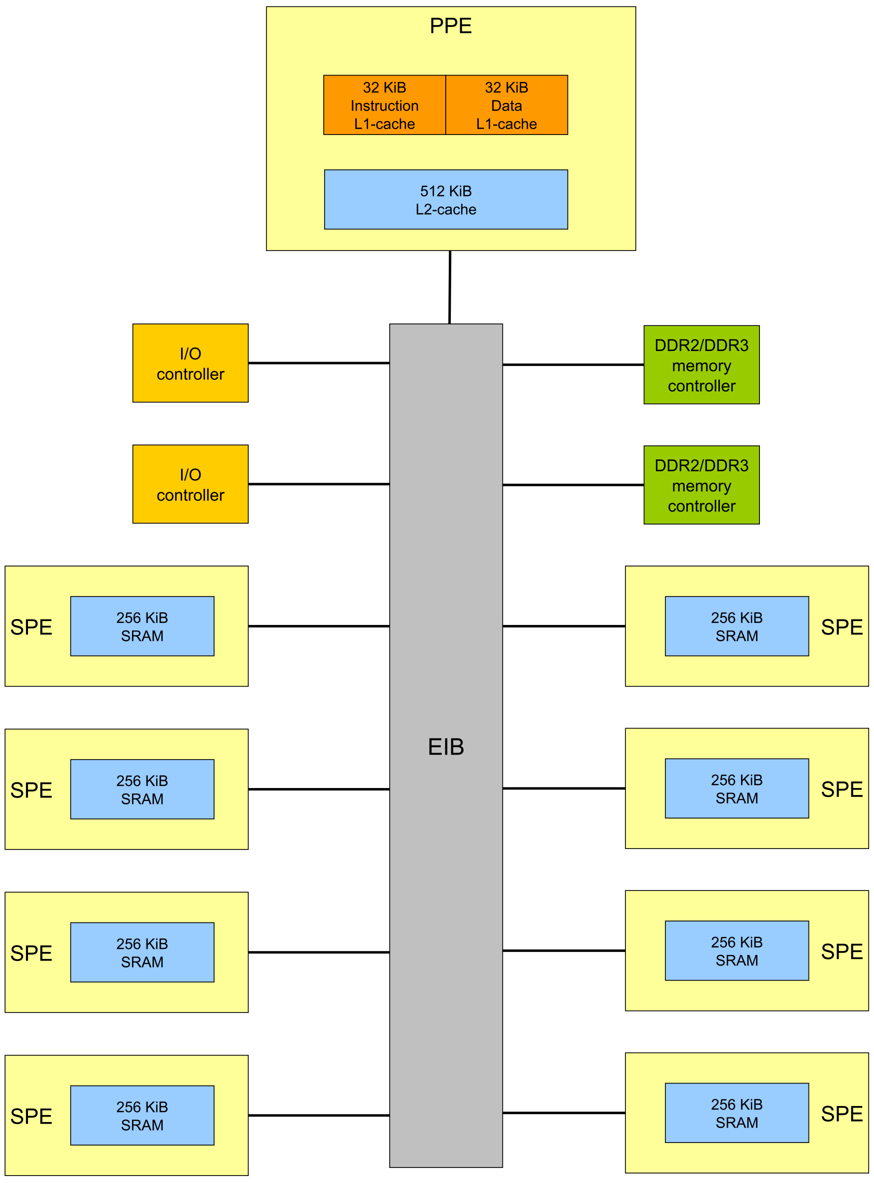


Figure 3.12: Architecture of the Cell processor [12].

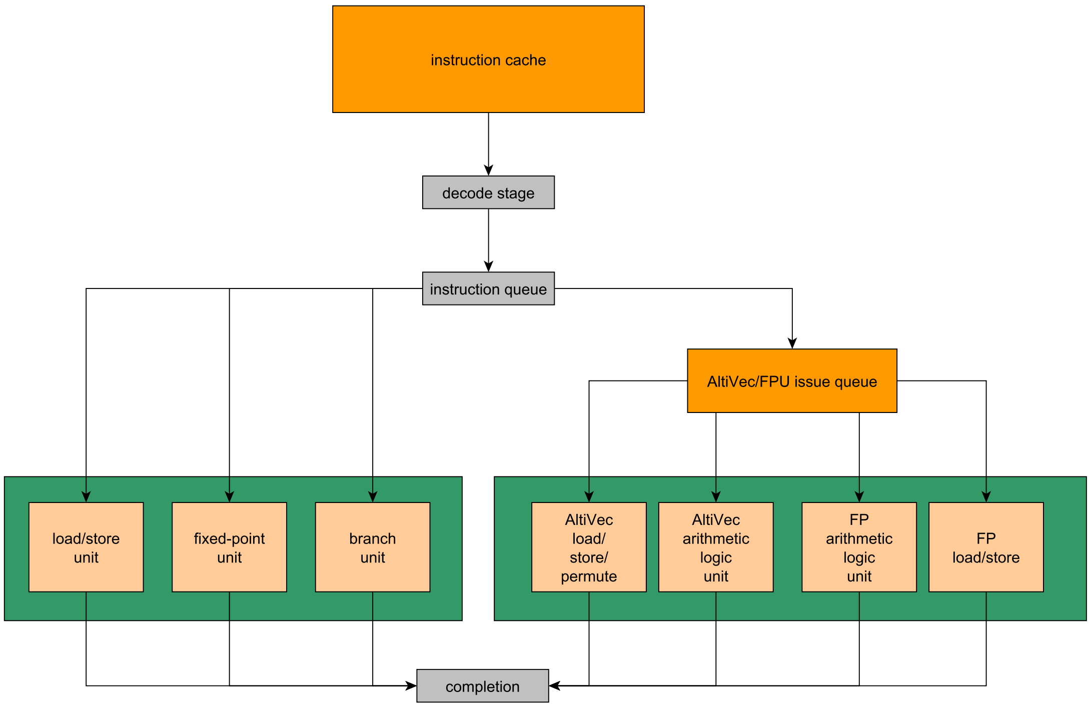


Figure 3.13: Architecture of the PPE in the Cell processor.

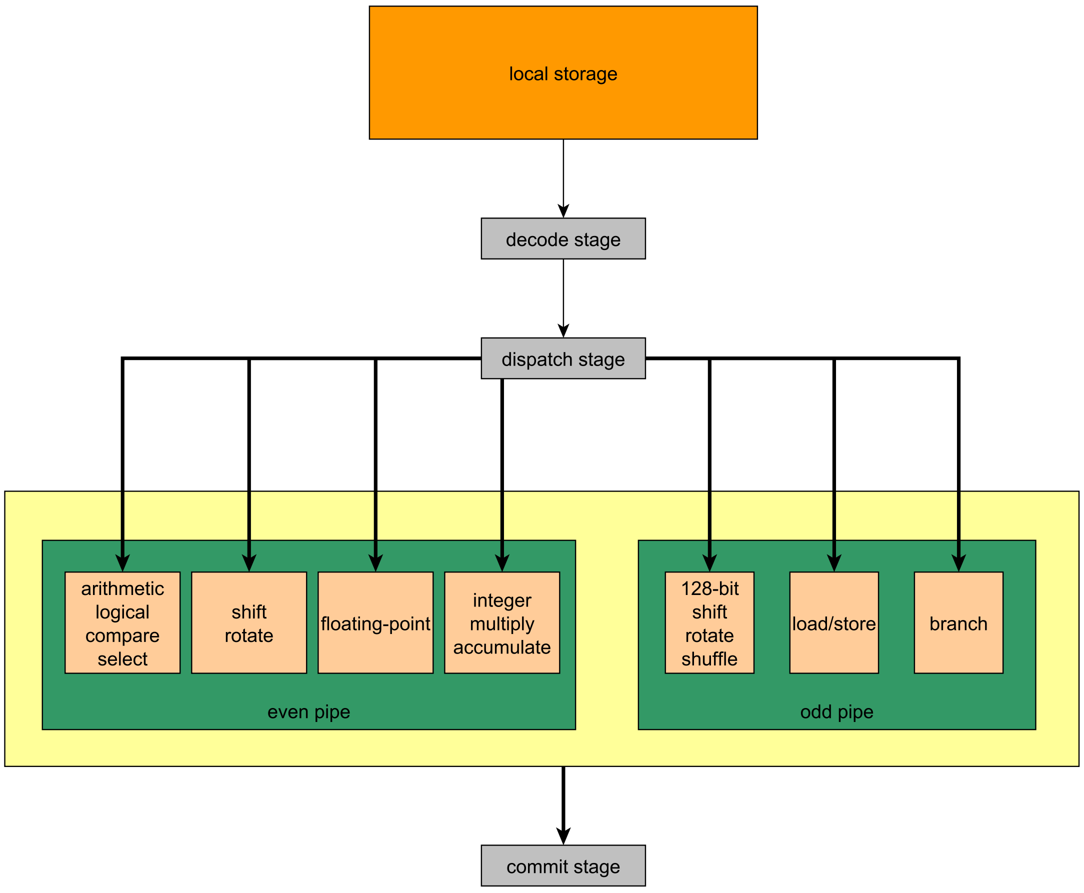


Figure 3.14: Architecture of an SPE in the Cell processor.

3.3 Architectures to be benchmarked

Due to practical availability, it was only possible to benchmark the following processors:

- Intel Atom Z530
- Intel Xeon E5540
- PowerPC MPC8548
- PowerPC P4080

Their properties are summarized in Table 3.1. We will not further discuss the other architectures that were introduced in the previous paragraphs, i.e. the P5020, DSPs and the Cell processor.

Architecture	cores	clock frequency	L1 cache		L2 cache	L3 cache
			instruction	data		
Intel Atom Z530	1	1596 MHz	32 KiB	24 KiB	512 KiB	-
Intel Xeon E5540	4	2527 MHz	32 KiB	32 KiB	256 KiB	8 MiB (shared)
MPC8548	1	1333 MHz	32 KiB	32 KiB	512 KiB	-
P4080	8	1500 MHz	32 KiB	32 KiB	128 KiB	2 MiB (shared)

Table 3.1: Important characteristics of the benchmarked architectures.

The benchmarks are presented in the following two chapters. On the Intel Atom processor, the BIOS settings could not be altered. Therefore, it should be noted that the benchmarks were run with the following settings:

- The hardware prefetchers for the cache are enabled. This is a mechanism that speculatively pre-fetches cache lines from memory before they are actually used. We do not expect this to have a big impact on the results as the experiments performed in the thesis are run from the cache levels.
- Hyper-threading (SMT on Intel platforms) is enabled. This slows down the execution of the experiments because another process can be scheduled on the same core. However, when running the experiments, we try to minimize the number of running processes in order to reduce the impact.
- The Intel SpeedStep technology is enabled. This is Intel's implementation of dynamic frequency scaling. This means that depending on the usage of the processor, the clock frequency is scaled dynamically to optimize power consumption. The consequence is that the clock frequency is not constant which makes it harder to establish expectancies for the benchmarks. Also, since we normalize the results of the benchmarks to remove the effect of the clock frequency, the normalized results will be less accurate.

3.4 A first comparison

Although the main focus of the study is performance, there are additional criteria that demand to be taken into account by the application environment:

- power consumption
- platform documentation
- duration of supply

3.4.1 Power consumption

The processors will be put upon AMC modules and will be placed in an AdvancedTCA shelf. A shelf has a limited cooling capacity and an upper bound on the power consumption of a module is defined in the AdvancedTCA specification. For this reason, the power consumption can be determinant in the choice for an architecture.

The Thermal Design Power (TDP) values of the architectures to be benchmarked are shown in Table 3.2. The following remarks apply to the table. Because the MPC8548 and the P4080 are SoCs, a single chip contains the core, the memory controllers and the controllers for the I/O peripherals. This is in contrast with the Atom and the Xeon processors. These chips only contain the processing core; an additional chipset is needed for the memory controller (called northbridge) and the peripherals (called southbridge). That is why we have to take into account the power consumption of these additional circuits in order to make a comparison.

Architecture	thermal design power (W)			
	processor	northbridge	southbridge	total
Intel Atom Z530	2.2	2.3		4.5
Intel Xeon E5540	80.0	27.1	4.5	111.6
MPC8548	6.5			6.5
P4080	19.0			19.0

Table 3.2: TDP values of the various processors.

The AdvancedTCA specification puts a limit of 60 W on the power consumption of an AMC board. The maximum power consumption can be increased by leaving some slots in a row unused such that the cooling capacity of an entire row is divided among the remaining slots. However, the resulting cooling capacity is still too low for the Intel Xeon processor which rules it out at first sight. Nonetheless, it is still worthwhile to investigate the performance of the Xeon processor because in principle, it is possible to step away from the usage of an AdvancedTCA shelf.

3.4.2 Platform documentation

The system under consideration is an industrial, real-time application. This means that

- the system must not show unpredicted behaviour
- the system must be fast so that the deadlines set by the application are met

To satisfy those requirements, it is important that the platform is well documented. This allows the software developers to thoroughly understand the architecture in order to provide a reliable application and to get the maximum performance out of the platform. In this area, the PowerPC processors manufactured by Freescale Semiconductor are much better documented than the Intel processors. The main reason for this difference is that Freescale Semiconductor essentially serves a different market than Intel: it focuses more on the industrial sector (automotive, medical) which typically have stricter requirements in terms of reliability and predictability.

3.4.3 Duration of supply

Intel ends the lifetime of a processor 5 years after its introduction. This means that neither the processor, neither support for it will be available after this time. In the case of Freescale Semiconductor, the end of life is minimum 10 years after a processor has been released. Depending on the estimated life-time of the system, this can be a determinant factor to choose for a Freescale processor instead of an Intel processor.

Chapter 4

Benchmarks for the host application

This chapter presents the comparison of the architectures presented in the previous chapter for the host application. Section 4.1 gives an overview of the characteristics of the application. Then, Section 4.2 introduces the concepts of micro- and macro-benchmarks. The benchmarking methodology is presented in Section 4.3. Section 4.4 presents the results of the memory benchmark while Section 4.5 discusses the benchmarks for the integer and floating-point performance. Finally, Section 4.6 formulates a conclusion of the benchmarks and Section 4.7 discusses future work.

4.1 Characteristics of the host application

The host application acts as a controller for the whole system and therefore its tasks are very diverse. In the software deployment, the application consists of about 30 Linux processes, divided over 3 AMC modules. Since the performed tasks are very diverse, there is not a clear dominating type of computation. Both floating-point and integer computations are performed, as well as communication over Ethernet and SRIO. Because of this computational diversity, we chose to run different types of benchmarks.

4.2 Micro- and macro-benchmarks

In the world of hardware benchmarking, there exist two types of benchmarks. Macro-benchmarks measure high-level units of work that are closely aligned with real-world workloads. Typical benchmarking suits are collections of programs that form a representative subset of the application space. Popular macro-benchmarks are Dhrystone [13], Whetstone [14] or the Linear Algebra PACKage (LAPACK) [15]. On the other hand, micro-benchmarks measure very small and specific units of work, such as the latency of a specific Linux system call or how fast a context switch occurs.

Macro-benchmarks are useful because they provide an absolute measure that can be used for comparison. The benchmarking application is run on each architecture and the latency is measured. Provided that the testing environment is the same for every architecture, the measurement results in a simple figure for performance comparison. However, this approach has limited value for us because it does not provide us insight into the reasons of why we see different results for different platforms. There are many factors and parameters that define the performance of an architecture. Therefore, we would like to refine our measurements by comparing the individual factors that make out the performance of the architecture. This knowledge enables us to make predictions on the performance of applications on the platform.

4.3 Benchmarking methodology

Our main focus is the computation ability of the processor. Therefore, we will concentrate on benchmarking the processing capabilities (integer and floating-point) and the performance of the memory hierarchy as we believe that these three aspects are determinant for the performance of the host application. The I/O performance, such as the performance of SRIO and Ethernet communication, is outside the scope of this research. Our methodology is to use macro-benchmarks for the measurement of the integer and floating-point performance and micro-benchmarks to measure the memory performance.

4.3.1 Micro-benchmark

LMBench [16] is a suite of micro-benchmarks. It consists of a collection of small programs that attempt to test various aspects of the performance of a computer system individually: memory performance, file system and disk performance, network performance... For our purpose, *LMBench* provides tests that measure the performance of the memory hierarchy. In particular, we will run the following benchmarks (the bandwidth benchmarks have been slightly modified to suit our purpose):

lat_mem_rd memory read latency

bw_mem_rd memory read bandwidth

bw_mem_wr memory write bandwidth

bw_mem_cp memory copy bandwidth

4.3.2 Macro-benchmark

The macro-benchmark set that is used is *nbench* [17]. It is composed of 10 different applications that can be categorized in three different categories: integer, floating-point and memory. These categories are not absolute: for instance, applications that fit in the floating-point category also perform integer operations and do memory transfers. However, the distinction is made based on the dominant type of operation that it performs.

We chose to only use the integer and floating-point benchmarks. There are two reasons:

- The micro-benchmarks of the *LMBench* suite allow us to gain a much more detailed view of the memory performance than the memory benchmarks of the *nbench* suite can offer us. The *nbench* applications are macro-benchmarks: they are 'polluted' with other operations such as integer computations and branches.

- In order to be able to understand the results of a memory benchmark of the *nbench* suite, its memory behaviour must be thoroughly analyzed (memory footprint, regularity of access,. . .). Given the size and complexity of the benchmarks, this is not feasible within our timespan.

The following *nbench* applications were classified as being integer-intensive:

- floating-point emulation
- Huffman compression
- International Data Encryption Algorithm (IDEA) encryption and decryption

Furthermore, the 'Huffman compression' benchmark also measures the branching performance of the processor. The following *nbench* applications were classified as being floating-point intensive:

- Fourier transform
- Neural network simulation
- LU decomposition

4.4 Performance of the memory hierarchy

With the *LMBench* benchmark suite, we are going to benchmark two aspects of the memory hierarchy: the read latency and the bandwidth for reading, writing and copying.

4.4.1 Memory read latency

The read latency benchmark is devised as follows. A list of pointers is constructed, where each pointer points to the next pointer in the list. This way, traversing the list results in the memory being 'walked'. The time to do about 1,000,000 loads (the list wraps around) is measured and reported. The actual number of loads is dynamically adjusted until a satisfying level of statistical confidence is achieved.

There are two parameters to the list:

size of the list The size of the list determines in what level of the memory hierarchy the list will be accessed from.

stride of access The stride of the access influences the results. If the stride is smaller than the cache line size, accesses to higher cache levels get more than one cache hit per cache line from lower cache levels. Also, if the stride is high, the list will be spread out over multiple memory pages. The consequence is that misses from the Translation Lookaside Buffer (TLB) will start having bigger influences on the latency results.

Figure 4.1 shows the result of the memory read latency test for the Atom Z530 for varying strides. The latency is presented as a function of the data size. The data size is the amount of contiguous memory that is needed to store the list. For varying strides, the number of elements in the list must

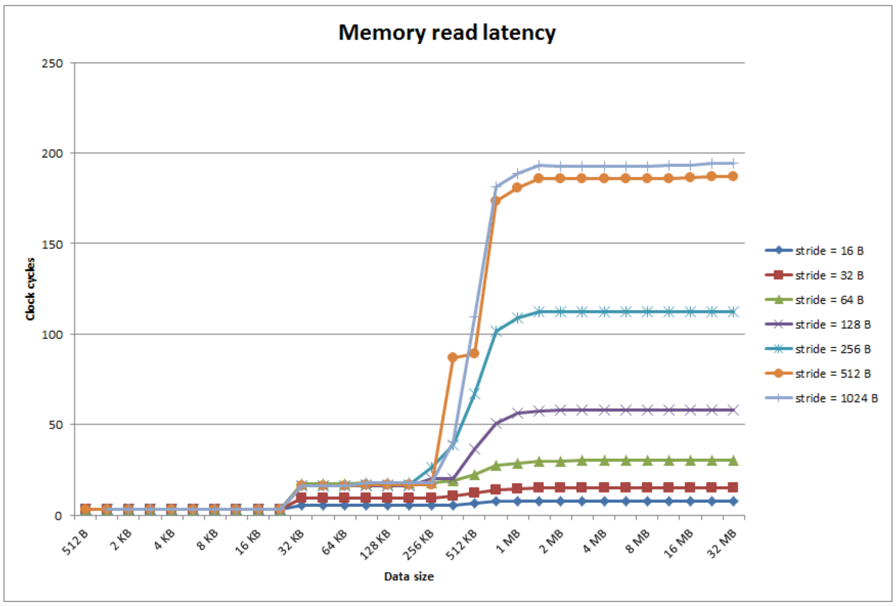


Figure 4.1: Results of the memory read latency benchmark for the Atom Z530.

be adapted to keep the data size constant. The plot tells us the following information about the architecture.

number of cache levels The course of the line as the data size increases is a sequence of plateaus. Each plateau corresponds to a level in the memory hierarchy. In this case, we can identify 3 plateaus which means that there are two levels of cache. The last plateau corresponds to main memory.

cache sizes The data size at which a plateau ends indicates the size of the cache level. Beyond that data size, a bigger portion of the list will be stored in a higher cache level until the part left in the lower cache level becomes insignificant compared to the total data size.

cache line size If we look at the plateau for the L2 cache in the plot in Figure 4.1, we see that starting from a stride of 64 bytes, the lines coincide. This means that the cache line size is 64 bytes. The reason is that a fetch from L2 cache will load an entire cache line into L1 cache. If the stride is smaller than the cache line size, the next item to be loaded from L1 instead.

prefetch mechanisms If we look at the plateau for the main memory access, we see that there are great differences in the memory read latency when varying the strides. Normally, it would be expected that, just like for the L2 cache, the lines coincide when the stride is beyond the cache line size. However, the observation is different due to prefetching mechanisms. Intel processors have a prefetch mechanism that ensures that a cache line is prefetched before it is used. For that, it is speculating on the need for a cache line. From the plot, it 'appears' that the cache line size is 512 bytes. This is in accordance with the documentation [5] because it states that the hardware prefetcher always tries to stay 256 bytes ahead. Beyond that stride, the increase in latency is due to more TLB misses.

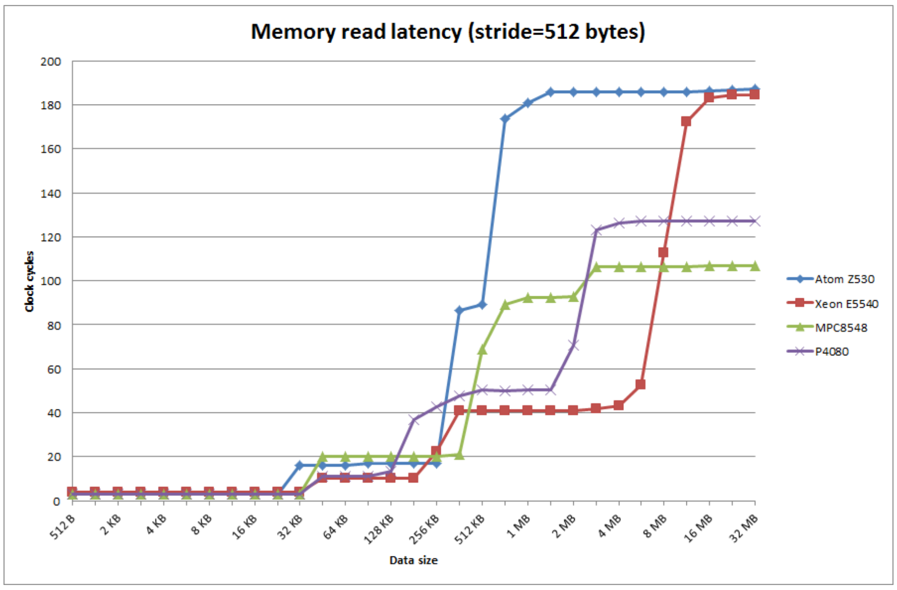


Figure 4.2: Results of the memory read latency benchmark for all architectures.

Figure 4.2 shows the memory read latency of the various processors for a stride of 512 bytes. This stride was chosen because it is the lowest stride for which there is no influence of the hardware prefetchers. To make a fair comparison of the architectures the influence of the clock frequency should be eliminated so we converted the results to clock cycles. The results are also shown in Table 4.1. In order to compare with the documentation, the latencies given in the processor datasheets are also given.

We have only extracted the latencies to the cache levels. The memory latency is dependent on both processor characteristics (memory controller) as well as on memory characteristics that are external to the processor. Ideally, we would wish to isolate the latency of the memory controller from the total latency in order to compare the performances of the memory controllers of the different processors. However, since we are working with different DRAM modules, we cannot do that. Therefore, we leave the main memory performance out of consideration.

Architecture	L1		L2		L3	
	measured	datasheet	measured	datasheet	measured	datasheet
Intel Atom Z530	3	3	16	16	-	-
Intel Xeon E5540	4	4	10	10	41	>35
MPC8548	3	3	20	20	-	-
P4080	3	3	11	11	50	N/A

Table 4.1: Cache access latencies extracted from the results in Figure 4.2 as well as the latencies given in the datasheets. The latencies are given in clock cycles.

All of the measured latencies are in accordance with the latencies given in the datasheet. For the latency to L1 cache, all processors have an access time of 3 clock cycles except the Xeon. The most significant difference is the L2 access time between the MPC8548 and the P4080. The read latency on the P4080 is almost half the read latency on the MPC8548. The reason is that each core of the P4080 has a backside L2 cache running at the clock frequency of the core. In the MPC8548, the L2 cache is connected to the global bus, which runs at a lower frequency. Also, the measurements show that there is a trade-off between cache size and cache latency. Indeed, the Atom Z530 and the MPC8548 have the highest L2 cache size (512 KB) but they also have the highest cache latency.

4.4.2 Memory bandwidth

We measure the memory bandwidth for three operations: read, write and copy. Furthermore, the benchmarks are run for two different data types: 32-bit integers and 64-bit floating-point numbers. This was chosen because we know that there the host application performs both integer and floating-point computations. Just as for the memory latency, we choose to focus on the cache levels only.

Each benchmark is implemented as shown in Listing 4.1. This loop is timed and the bandwidth is calculated as

$$\text{bandwidth} = \frac{\text{data size}}{\text{execution time}}.$$

Listing 4.1: Implementation of the bandwidth benchmarks.

```
// TYPE is either "int" (32-bit integer) or "double" (64-bit floating-point)
TYPE data[DATA_SIZE / sizeof(TYPE)];
TYPE *p = data;
int nr_iterations = DATA_SIZE / (32 * sizeof(TYPE));
for (i=0 ; i<nr_iterations ; i++)
    /*
   * perform operation on p[0], p[1], ..., p[31]
   */
  p += 32;
```

Every iteration of the loop performs the operation on 32 data elements. The loop was unrolled in order to reduce the loop overhead. To measure the L1 cache bandwidth, the data size is set to 8 KiB for all architectures. For the L2 cache, the data size is set to 128 KiB for the Intel Atom, the Intel Xeon and the MPC8548. Because the P4080 has an L2 cache size of 128 KiB, it was chosen to benchmark its bandwidth with a data size of 64 KiB.

Using the instruction latencies and throughputs from the datasheets of the processors as well as the memory latency results from Section 4.4.1, we calculated predictions for the bandwidths to be measured. There are several remarks:

- The instruction timings for the MPC8548 and the P4080 were extracted from [6] and [8] respectively.

- The Intel documentation [5] gives incomplete information about the instruction timings of its processor models. The technical reports [18] and [19] report instruction latencies for various x86 architectures that were experimentally measured. Therefore, the timing data from these documents were used instead.

- As mentioned before, the Intel Atom Z530 has its hardware prefetchers enabled. This mechanism improves the bandwidth of the memory hierarchy but in the predictions, this mechanism was not taken into account.

- Because the loop is unrolled, the loop overhead is deemed negligible and is therefore not taken into account for the estimations.

- The MPC8548 has a 32-byte cache line while the other processors all have a 64-byte cache line. This means that the MPC8548 has twice as many data accesses to higher cache levels than the other processors.

4.4.3 Memory bandwidth: expectations

Read

For the reading operation, we only want to load the entire array without performing any operations on the data. However, when written in C, an optimizing compiler removes instructions that load data values that are not used. Therefore, the body of the loop has been coded by hand in assembly. Table 4.2 shows the expected bandwidth results for the read operation. The bandwidths are given in bytes per cycle as we want to discard the effect of the clock frequency. We want to compare the processors on an architectural level.

Architecture	L1		L2	
	int	double	int	double
Intel Atom Z530	4.00	8.00	2.21	3.37
Intel Xeon E5540	4.00	8.00	2.91	4.00
MPC8548	4.00	8.00	1.28	1.52
P4080	4.00	8.00	2.67	4.00

Table 4.2: Estimated bandwidth for the read operation in bytes per cycle.

The L1 bandwidth is expected to be the same for all architectures because a throughput of 1 instruction per cycle can be achieved. Furthermore, the bus from the L1 cache to the load-store unit is wide enough to carry 8 bytes in a single clock cycle. Therefore, the bandwidth for the 64-bit floating-point data is twice the bandwidth for the 32-bit integers.

The L2 bandwidth depends mostly on the latency to the L2 cache. When there is a cache miss in the L1 cache, an entire cache line is fetched from the L2 cache and stored in the L1 cache (64 bytes for all architectures except the MPC8548 which has a 32 byte cache line). Subsequent items are read from the L1 cache until a cache miss for the next cache line occurs. The number of items in a cache line depends on the data type and the size of the cache line.

Write

To benchmark the write operation, the loop shown in Listing 4.2 is executed. In contrast with the read operation, no assembly code needed to be written; the compiler performed a good transformation of C code into machine-dependent instructions.

Listing 4.2: Implementation of the "write" benchmark.

```
/* TYPE is either "int" (32-bit integer) or "double" (64-bit floating-point) */
TYPE data[DATA_SIZE / sizeof(TYPE)];
TYPE *p = data;
int nr_iterations = DATA_SIZE / (32 * sizeof(TYPE));
for (i=0 ; i<nr_iterations ; i++)
  p[0] = p[1] = ... = p[31] = 1;
  p += 32;
```

Architecture	L1		L2	
	int	double	int	double
Intel Atom Z530	4.00	8.00	2.21	3.20
Intel Xeon E5540	4.00	8.00	2.91	4.27
MPC8548	4.00	8.00	1.28	1.52
P4080	4.00	8.00	2.67	4.00

Table 4.3: Estimated bandwidth for the write operation in bytes per cycle.

Again, as for the read operation, the predicted bandwidths to the L1 cache are the same for all platforms and have the same value as the read operation.

The predicted L2 bandwidths are approximately the same as for the read operation. This is because we assume that a write to L2 cache is just as costly as a read from L2 cache. Indeed, when we write a new value, it is first written in L1 cache. However, a cache line will need to be evicted to L2 cache. A new cache line cannot be written before the victim cache line has been written to L2 cache in its entirety. We assume that this takes as long as a read from L2 cache. We need to make this assumption because the documentation for the platforms lacks timing details for this.

The only difference with the read operation lies in the L2 bandwidth for 64-bit floating-point data for the Atom and the Xeon. This is because the latency of the "mov" instruction is different for a load and for a store.

Copy

Just as for the write operation, the copy operation is written in plain C. Its implementation is shown in Listing 4.3.

Listing 4.3: Implementation of the "copy" benchmark.

```
/* TYPE is either "int" (32-bit integer) or "double" (64-bit floating-point) */
TYPE data_src[DATA_SIZE / sizeof(TYPE)];
TYPE data_dst[DATA_SIZE / sizeof(TYPE)];
TYPE *src = data_src;
TYPE *dst = data_dst;
int nr_iterations = DATA_SIZE / (32 * sizeof(TYPE));
for (i=0 ; i<nr_iterations ; i++)
  dst[0] = src[0];
  dst[1] = src[1];
  ...
  dst[31] = src[31];
  src += 32; dst += 32;
```

Architecture	L1		L2	
	int	double	int	double
Intel Atom Z530	1.00	0.89	0.70	1.00
Intel Xeon E5540	4.00	8.00	1.83	2.46
MPC8548	2.06	4.00	0.89	0.76
P4080	2.06	4.00	1.60	2.37

Table 4.4: Estimated bandwidth for the copy operation in bytes per cycle.

For the L1 bandwidth for the copy operation, the situation is different than for reads and writes.

- It is expected that the Xeon maintains the same bandwidth because of its separate load and store execution units while the other processors have a combined load/store unit.

- The Atom has an in-order execution flow. Therefore, it cannot achieve the same degree of pipelining as the other processors.

The L2 bandwidth for the copy operation is expected to be lower than the read and the write bandwidth. This holds for all architectures. The reason is that, on average, there are twice as many cache misses because both the load and the store operations can cause a miss.

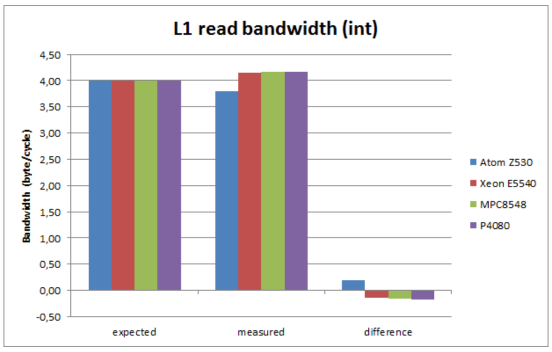
4.4.4 Memory bandwidth: assessing the expectations

Read

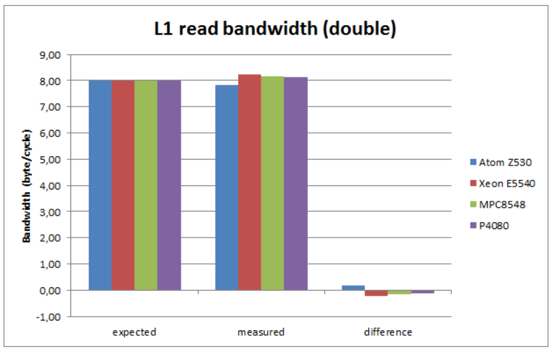
Figures 4.3a and 4.3b show the results of the L1 read bandwidth and Figures 4.4a and 4.4b show the results of the L2 read bandwidth. Each plot shows three columns: the expected bandwidth, the measured bandwidth and the absolute difference.

$$\text{difference} = \text{expected} - \text{measured}$$

This means that a positive difference indicates an overestimation of the bandwidth; similarly a negative difference indicates an underestimation of the bandwidth.

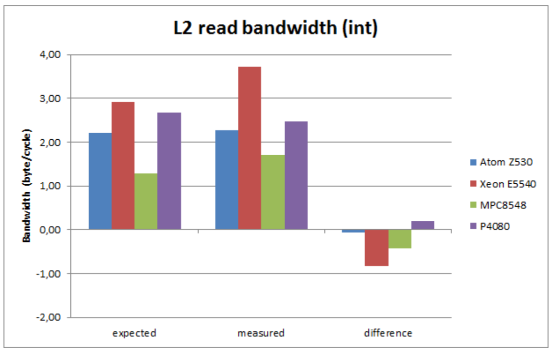


(a) L1 read bandwidth (int)

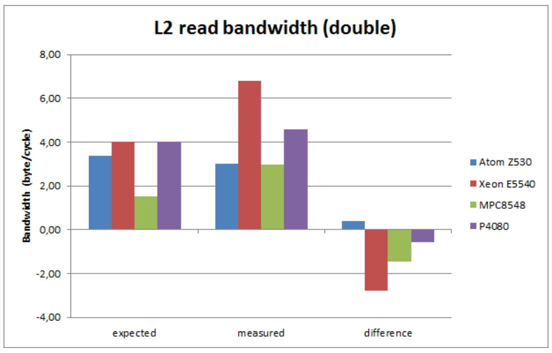


(b) L1 read bandwidth (double)

Figure 4.3: Bandwidth results of the L1 cache for the read operation.



(a) L2 read bandwidth (int)



(b) L2 read bandwidth (double)

Figure 4.4: Bandwidth results of the L2 cache for the read operation.

Architecture	measured (byte/cycle)				difference (%)			
	L1		L2		L1		L2	
	int	double	int	double	int	double	int	double
Intel Atom Z530	3.80	7.83	2.28	2.99	4.9	2.1	-3.2	11.1
Intel Xeon E5540	4.14	8.22	3.73	6.78	-3.6	-2.8	-28.2	-69.6
MPC8548	4.16	8.15	1.71	2.97	-4.1	-1.9	-33.4	-94.9
P4080	4.17	8.12	2.47	4.57	-4.2	-1.5	7.4	-14.3

Table 4.5: Comparison of the estimated bandwidths and the measured bandwidths for the read operation.

Table 4.5 shows the measured bandwidths alongside the difference with the expected values, i.e.

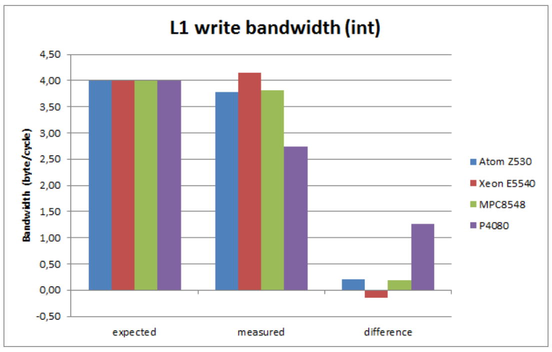
measured This column represents the measured bandwidths in bytes per cycle.

difference This column indicates how much the measured values differ from the expected values.

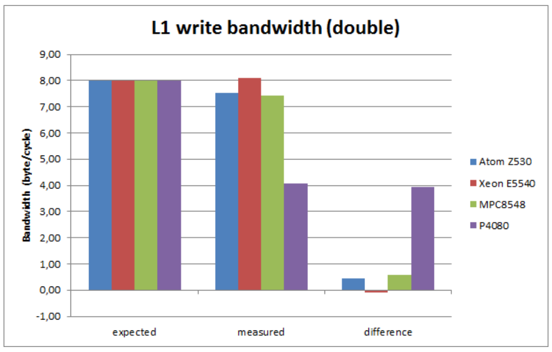
$$\text{difference} = 100\frac{\text{expected} - \text{measured}}{\text{expected}}$$

The expected bandwidths for the L1 cache are accurate as they are all within 5% of the measured bandwidths. For the L2 cache, there are some differences. We could not trace back the exact reason of the differences. There are factors that we did not take into account in our models to make the predictions and we could not manage to attribute the differences to these factors.

Write

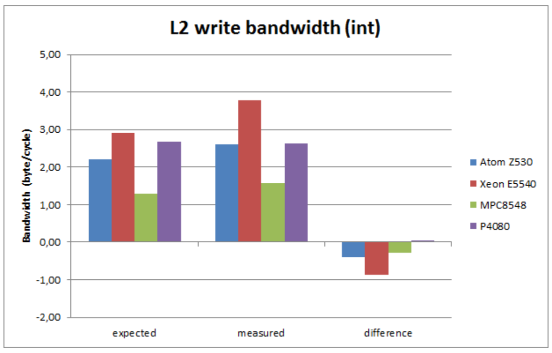


(a) L1 write bandwidth (int)

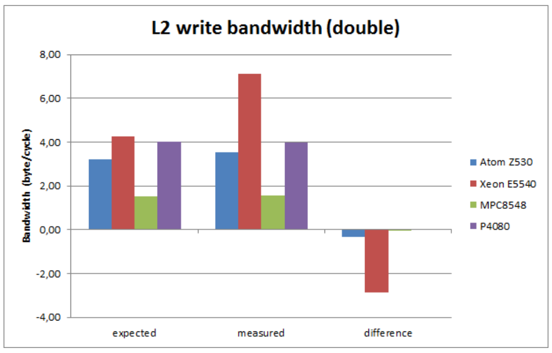


(b) L1 write bandwidth (double)

Figure 4.5: Bandwidth results of the L1 cache for the write operation.



(a) L2 write bandwidth (int)



(b) L2 write bandwidth (double)

Figure 4.6: Bandwidth results of the L2 cache for the write operation.

Presented in the same way as for the read operation, Figures 4.5 and 4.6 show the bandwidths for the write operation for respectively the L1 and the L2 cache.

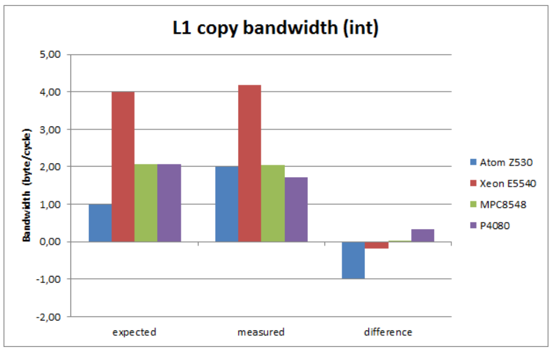
Architecture	measured (byte/cycle)				difference (%)			
	L1		L2		L1		L2	
	int	double	int	double	int	double	int	double
Intel Atom Z530	3.79	7.54	2.60	3.55	5.4	5.7	-17.8	-10.8
Intel Xeon E5540	4.15	8.09	3.77	7.12	-3.8	-1.1	-29.7	-66.9
MPC8548	3.81	7.43	1.57	1.56	4.8	7.1	-22.7	-2.6
P4080	2.73	4.08	2.62	4.00	31.7	49.0	1.6	0.1

Table 4.6: Comparison of the estimated bandwidths and the measured bandwidths for the write operation.

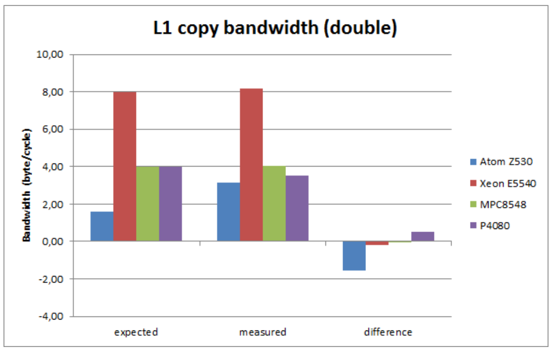
As for the read operation, Table 4.6 compares the expected values with the measured values. For the L1 cache, the expectations were correct except for the P4080 for floating-point values. We were expecting a bandwidth of 8 bytes per cycle as [8] mentions a 1 cycle throughput for the floating-point store instruction. Therefore, we cannot explain this result.

Also, the bandwidth results for the L2 cache deviate from the expected results for the Xeon E5540, the MPC8548 and the P4080. As for the read operation, we did not succeed in assigning the differences to individual factors.

Copy



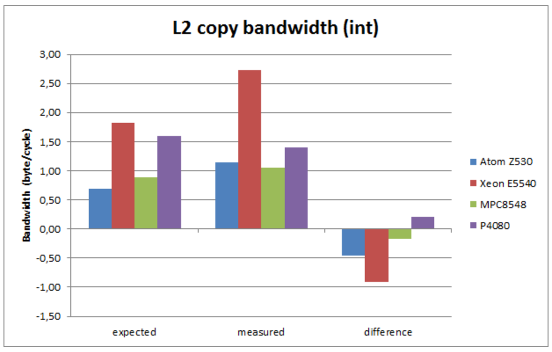
(a) L1 copy bandwidth (int)



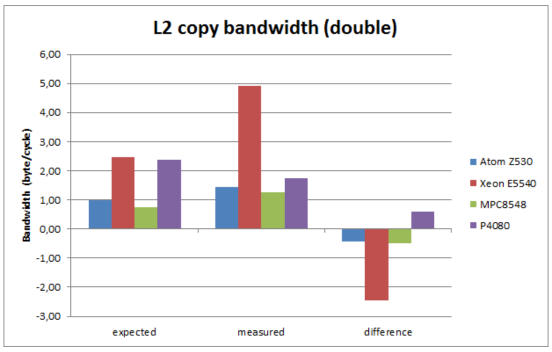
(b) L1 copy bandwidth (double)

Figure 4.7: Bandwidth results of the L1 cache for the copy operation.

As for the read and write operations, the results are shown graphically in Figures 4.7 and 4.8.



(a) L2 copy bandwidth (int)



(b) L2 copy bandwidth (double)

Figure 4.8: Bandwidth results of the L2 cache for the copy operation.

Architecture	measured (byte/cycle)				difference (%)			
	L1		L2		L1		L2	
	int	double	int	double	int	double	int	double
Intel Atom Z530	1.99	3.13	1.15	1.43	-99.2	-95.6	-65.0	-43.4
Intel Xeon E5540	4.18	8.18	2.73	4.93	-4.4	-2.3	-49.5	-100.2
MPC8548	2.06	4.03	1.05	1.25	0.4	-0.9	-18.3	-63.9
P4080	1.73	3.50	1.40	1.76	0.3	12.5	12.7	25.8

Table 4.7: Comparison of the estimated bandwidths and the measured bandwidths for the copy operation.

Table 4.7 compares the expected results with the measured bandwidths for the copy operation. For the L1 cache results, the expectations were accurate except for the Atom Z530. For that processor, it was expected that the in-order execution flow is a big disadvantage because it cannot interleave the pairs of load/store instructions. However, from the results, it appears to do so nonetheless, behaving like an out-of-order processor. More information about this could not be found in the documentation.

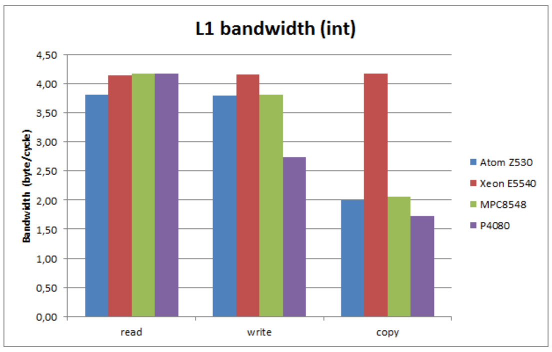
As for the read and write operations, we did not succeed in assigning the differences between the expected and measured L2 bandwidth to individual factors.

4.4.5 Memory bandwidth: conclusion

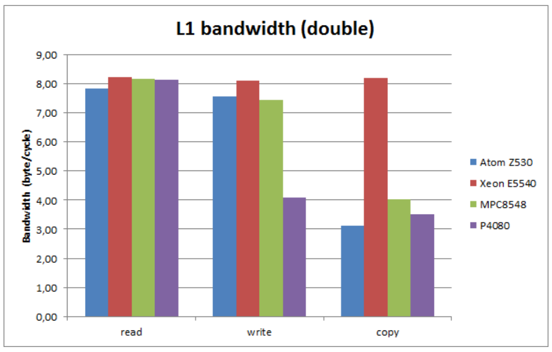
Figures 4.9 and 4.10 show the measured results for the three operations together. For the L1 bandwidth, we can state the following:

- For the read and write operations, all architectures have equal performance. There is one peculiar case for the P4080 on the writing benchmark but it is expected that this is due to a measurement error.

- For the copy operation, the Atom Z530, the MPC8548 and the P4080 have approximately the

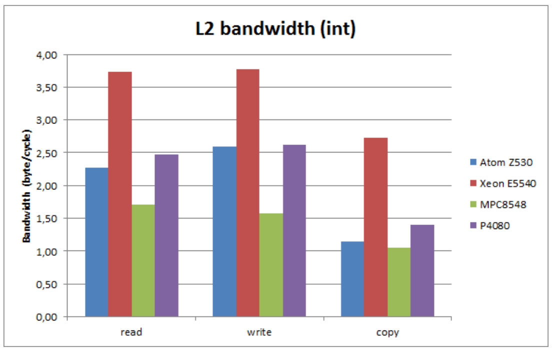


(a) L1 bandwidth (int)

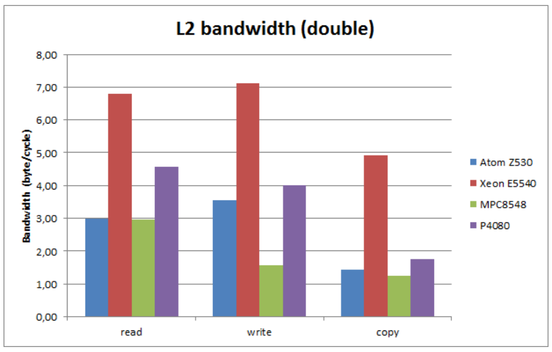


(b) L1 bandwidth (double)

Figure 4.9: L1 bandwidth results.



(a) L2 bandwidth (int)



(b) L2 bandwidth (double)

Figure 4.10: L2 bandwidth results.

same performance.

- The Xeon E5540 is the only processor that can maintain the same performance on the copy operation as for the read and write operations. The reason is that it has separate execution units for loads and stores, allowing it to perform a load and a store simultaneously.

For the L2 bandwidth, we can make the following remarks.

- We can conclude that the relative performance on the L2 bandwidth is decided by the L2 access latency. Overall, the Xeon has the best performance, followed by the P4080, the Atom and the MPC8548. This order corresponds to the L2 memory latencies measured in Section 4.4.1.

- For the integer bandwidth, the performance of the Atom Z530 is very close to the performance of the P4080, despite the relatively big difference in L2 cache latency (11 cycles for the P4080 while it is 16 cycles for the Atom Z530). We conjecture that this is due to the hardware prefetcher of the Atom. However, the tested system does not allow to disable this functionality so we cannot validate this hypothesis.

4.5 Integer and floating-point performance

The *nbench* suite allows us to benchmark the integer and floating-point performance of the processors. The result of running a particular application of the *nbench* suite is the number of iterations of the application that it can run per second. In the results of the *nbench* benchmarks, we use the MPC8548 as a reference because it is the current hardware implementation of the host application. Therefore, we wish to compare the performance of each processor relative to the performance of the MPC8548. Since we want focus on architectural differences, the effect of the clock frequency should be neutralised. For the results shown in Figures 4.11 and 4.12 we therefore used the following transformation.

$$\text{result}_{\text{processor}} = 100\log\left(\frac{\frac{\text{number of iterations}_{\text{processor}}}{\text{number of iterations}_{\text{MPC8548}}}}{\frac{\text{f}_{\text{processor}}}{\text{f}_{\text{MPC8548}}}}\right) \tag{4.1}$$

Because of the $\log$ operation, a positive, negative or zero result means that the processor behaves respectively better, worse or just like a frequency-scaled MPC8548.

4.5.1 Integer performance

The results of the integer benchmarks are shown in Table 4.8 and presented visually in Figure 4.11. The original results, which represents the number of iterations per second achieved on a particular application, were transformed using the formula in (4.1). In the plot, a positive result means that the processor is faster than the MPC8548 clocked at the same frequency. A negative result means that it is slower.

	Intel Atom Z530	Intel Xeon E5540	MPC8548	P4080
FP emulation	-36.6	-3.8	0.0	7.2
Huffman	-10.3	29.6	0.0	5.7
IDEA	-16.6	9.5	0.0	1.4

Table 4.8: Results of the benchmarks for the integer performance relative to the MPC8548 and the clock frequency of the processor. A positive, negative or zero result means that the processor behaves respectively better, worse or just like a frequency-scaled MPC8548.

First of all, we can see that the Atom Z530 behaves worse than an MPC8548 running at the Atom's frequency. This indicates that the integer performance of the Atom is relatively bad compared to the integer performance of the MPC8548. We conjecture that this is due to the lack of exploitation of ILP: the MPC8548 has two integer execution units and can execute instructions out-of-order while the Atom has only one integer execution unit. Since the basic integer operations (add, subtract, shift, logical) take 1 cycle, the MPC8548 can execute 2 integer operations per cycle while the Atom can only execute 1 per cycle.

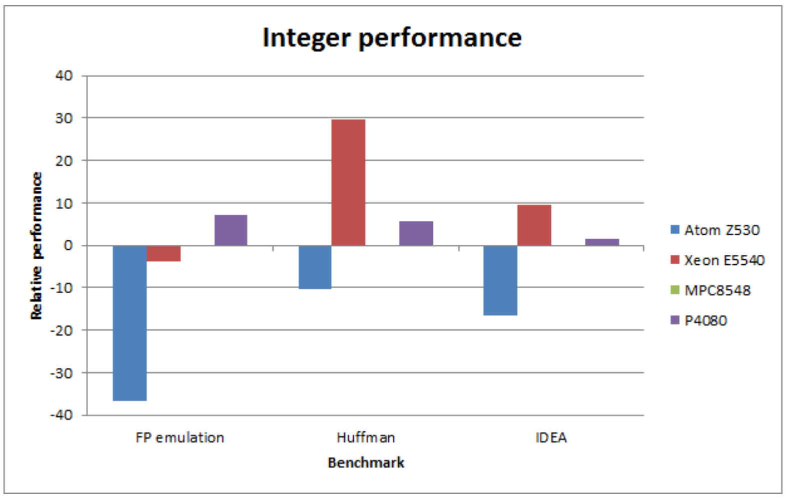


Figure 4.11: Visual representation of the results in Table 4.8.

The P4080 behaves better, meaning that there has been an improvement in the core for integer operations. However, this is probably due to the faster L2 cache (11 clock cycles for the P4080 while 20 clock cycles for the MPC8548). An indication for this hypothesis is that the IDEA benchmark operates on data sets of 4000 bytes. This means that all its data resides in L1 cache. This is also the benchmark where the improvement of the P4080 is the least. The other two benchmarks, Huffman and floating-point emulation, all use bigger data sets and thus make more accesses to L2 cache.

The performance of the Xeon E5540 is highly variable. On the floating-point emulation benchmark, it performs worse than an MPC8548 clocked at the same frequency. Meanwhile, it performs well on the Huffman and the IDEA benchmarks. The big difference on the Huffman benchmark comes from branching. The algorithm for Huffman coding and decoding contains many branches and less integer computations. The IDEA and the floating-point benchmarks contain much less branching but have more intensive integer computations. Therefore, the results primarily show that the Xeon has very good branch prediction mechanisms compared to the MPC8548.

4.5.2 Floating-point performance

The results of the floating-point benchmarks are shown in Table 4.9 and presented visually in Figure 4.12. Just as for the integer results, the transformation in (4.1) was applied to the results.

The Atom performs better than the MPC8548 for the Fourier benchmark and the LU decomposition. However, it performs worse for the neural net benchmark. A possible explanation is that the neural net test depends a lot on the computation of the sigmoid function $f(t) = \frac{1}{1+\exp -t}$. This involves a floating-point division, which is particularly costly on the Atom (see Table 4.10 for the latencies of the floating-point instructions).

For floating-point performance, the P4080 performs slightly worse than an MPC8548 clocked at the same frequency. The reason can be found in Table 4.10. In the e500mc core, the floating-point unit has changed compared to the floating-point unit in the e500v2 core. The operations take longer

	Intel Atom Z530	Intel Xeon E5540	MPC8548	P4080
Fourier	10.7	45.7	0.0	-1.7
Neural net	-19.3	54.4	0.0	-2.9
LU decomposition	4.2	61.7	0.0	-3.4

Table 4.9: Results of the benchmarks for the floating-point performance relative to the MPC8548 and the clock frequency of the processor. A positive, negative or zero result means that the processor behaves respectively better, worse or just like a frequency-scaled MPC8548.

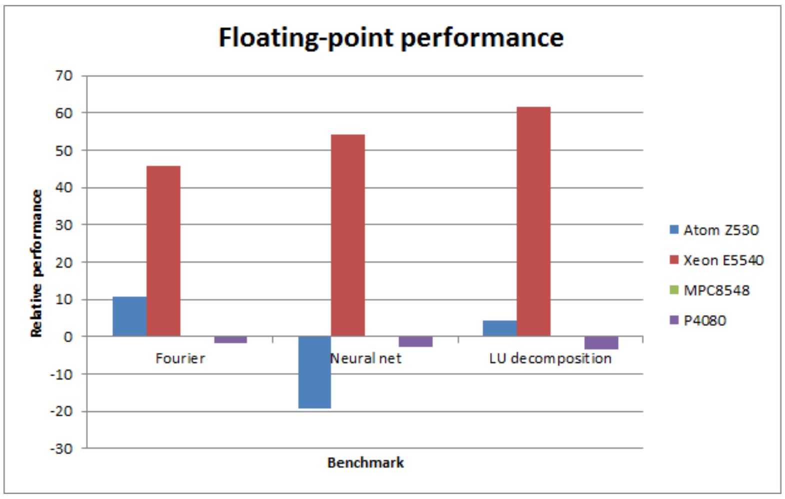


Figure 4.12: Visual representation of the results in Table 4.9.

and have a lower throughput.

The Xeon does better on all floating-point benchmarks. There are several factors that contribute.

- It has three execution units dedicated to floating-point operations: one for multiplication and division, one for addition and subtraction and one for floating-point comparisons. With ILP enabled, this allows the simultaneous execution of three floating-point operations. On the other processors, all floating-point operations are done by a single unit.
- As shown by Table 4.10, the latencies of the basic mathematical floating-point operations are lower than or equal to the instruction latencies of the other processors.
- The Xeon has dedicated floating-point registers. On the MPC8548, the floating-point data is stored in general-purpose registers, which are also used for the other data elements. The Xeon therefore suffers less from register pressure.
- Its latency to the L2 cache is the lowest of all.

Architecture	addition		subtraction		multiplication		division	
	L	T	L	T	L	T	L	T
Intel Atom Z530	5	1	5	1	5	2	65	64
Intel Xeon E5540	3	1	3	1	5	1	22	22
MPC8548	6	1	6	1	6	1	32	32
P4080	8	2	8	2	10	4	68	68

Table 4.10: Latencies (L) and throughputs (T) in clock cycles of the floating-point instructions for each processor. The throughput is the number of clock cycles necessary before the next instruction can be executed.

4.6 Conclusion

With this series of benchmarks, we have measured three aspects of the architectures.

- performance of the memory hierarchy (L1 and L2 cache levels)
- integer performance
- floating-point performance

We do not possess a clear insight in the behaviour of the host application. Furthermore, there is not a model available that is representative of the computational load of the application. Therefore, with the results of the benchmarks that were run, we can only give an indication of the performance of the application on each platform.

From the perspective of performance, the Intel Xeon architecture is the clear winner.

- It achieves high scores for memory performance and floating-point performance.
- Its result on the Huffman benchmark on Figure 4.11 shows that its branching prediction mechanism is very powerful.
- For integer computations, it performs equally well compared to the other benchmarked architectures.
- The benchmarks measured only one core but each Xeon E5540 processor has four identical cores.

However, as was described in Section 3.4, there are also non-performance related factors that have to be taken into account. Table 4.11 shows a final comparison of the architectures based on the benchmarks (for the columns "memory", "integer", "floating-point" and "branching") as well as the other factors of importance (the columns "power", "documentation" and "supply"). Based on this information, we can conclude the following.

- If it is feasible to step away from the usage of AdvancedTCA shelfs to store the modules and both the shorter supply duration (5 years for Intel) and the lack of documentation do not cause a problem, the Xeon architecture should be chosen because of its performance.

- If there is a wish to keep using the AdvancedTCA technology or either a longer supply duration is needed or the extensive documentation is desired, the choice should go to the P4080 processor. Each core is approximately equivalent to a MPC8548 processor but with a better memory hierarchy and it has eight of them. Its total power consumption is higher but that does not matter as long as it stays within the boundary set by the AdvancedTCA specification.
- The main strength of the Atom processor compared to the P4080 is its power consumption. The P4080 wins or ties on all the other aspects. Since power consumption is not important as long as it stays under the specified maximum, the P4080 should be chosen over the Atom.

	memory	integer	floating-point	branching	power	documentation	supply
Intel Atom Z530	+	-	0	0	+	-	-
Intel Xeon E5540	++	0	++	++	- -	-	-
MPC8548	0	0	0	0	0	+	+
P4080	+	0	0	0	-	+	+

Table 4.11: Final comparison of the four benchmarked architectures.

4.7 Future work

As was mentioned in the previous paragraph, we do not have a clear profile of the host application. To validate our conclusion, we would have to run the real application on each processor. During the thesis, this was not possible because the environment in which the application runs is complex and not easily reproducible. An alternative would be to profile the application and develop a model based on the profiling results that would mimic its computational behaviour. This was the approach taken for the HPPC application. Such a model has the advantage of being simple to execute yet it is representative of the real application. Also, the process of constructing the model requires investigating the behaviour of the application, thereby increasing its understanding.

We know that the application is composed of a number of Linux processes (about 30 processes running at any time). Therefore, the context switching latency can also be a factor for the choice of an architecture. Besides the benchmarks for the memory latency and bandwidth, *LMBench* also has a benchmark to compute the context switching latency. However, the results that were gained were highly variable across executions and also unrealistically low for some processors. Therefore, the results were not deemed to be significant and this was the reason that the results were not included in the thesis. However, it would be interesting to investigate into the source of the variability and to see whether a method can be found that can reliably measure the context switching latency.

Chapter 5

Benchmarks for the HPPC application

This chapter investigates the performance of the architectures on the HPPC application. Section 5.1 discusses the model of the HPPC application. It introduces the concept of arithmetic intensity to describe the behaviour of a computational block. Then, Section 5.2 describes the benchmarking methodology. Section 5.3 discusses the suitability of the architectures for the application and formulates expected results. Section 5.4 discusses the results of the benchmarks. Finally, Section 5.5 presents the conclusion of the chapter and Section 5.6 discusses future work emanating from the study.

5.1 Model of the HPPC application

5.1.1 Introduction

It was explained earlier that the HPPC application consists of two types of tasks: a sample task that performs the actual computations and a set of background tasks that serve as an interface between the sample task and the host application. This chapter investigates the performance of the sample task only; the background tasks are left out of consideration because profiling information derived from the current implementation indicates that they do not form a bottleneck in the system.

We have a model application at our disposal that simulates the behaviour of the sample task. It consists of a sequence of computational blocks that are arranged in a chain, i.e. the output of one block is the input of the next block. Each block takes 4 inputs, performs some operations on them and produces 4 outputs. The outputs are passed on to the next block using pointers. We make a distinction between data block and code blocks: a data block is a data structure while a code block is a function that performs a computation on a data block.

There are four types of blocks: the types *mul*, *div*, *matrix* and *mul-max*. In the simulation application, the different types of blocks appear consecutively as shown in Figure 5.1. To mimic the real sample task of the HPPC application, all operations are performed on double-precision floating-point (i.e. 64 bit) data items. Tables 5.1 and 5.2 describe the four types of blocks in the simulation application.

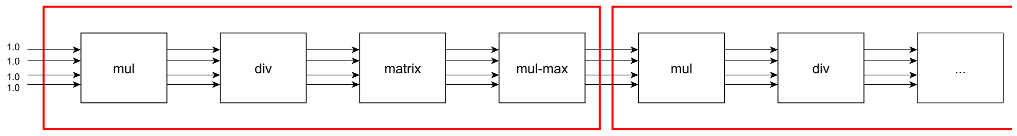


Figure 5.1: The simulation application of the sample task is a chain of blocks. Each block takes 4 inputs and produces 4 outputs. There are four different kinds of blocks and they are chained consecutively.

Type	Function	Parameters	Code

mul	Scalar multiplication	`param[4]`	`out1 = in1 * param[0]`
			`out2 = in2 * param[1]`
			`out3 = in3 * param[2]`
			`out4 = in4 * param[3]`

div	Scalar division	`param[4]`	`out1 = in1 / param[0]`
			`out2 = in2 / param[1]`
			`out3 = in3 / param[2]`
			`out4 = in4 / param[3]`

matrix	Matrix multiplication	`param[4][4]`	`out1 = in1 * param[0][0] + in2 * param[0][1] +`
			`in3 * param[0][2] + in4 * param[0][3]`
			`out2 = in1 * param[1][0] + in2 * param[1][1] +`
			`in3 * param[1][2] + in4 * param[1][3]`
			`out3 = in1 * param[2][0] + in2 * param[2][1] +`
			`in3 * param[2][2] + in4 * param[2][3]`
			`out4 = in1 * param[3][0] + in2 * param[3][1] +`
			`in3 * param[3][2] + in4 * param[3][3]`

mul-max	Scalar multiplication and max-operation	`param[4]`	`t1 = in1 * param[0]`
			`t2 = in2 * param[1]`
			`t3 = in3 * param[2]`
			`t4 = in4 * param[3]`
			`out1 = (t1 >= t2) ? t1 : t2`
			`out2 = (t2 >= t3) ? t2 : t3`
			`out3 = (t3 >= t4) ? t3 : t4`
			`out4 = (t4 >= t1) ? t4 : t1`

Table 5.1: Description of the different types of blocks in the simulation application for the sample task.

5.1.2 Arithmetic intensity

The arithmetic intensity of an application is defined as the ratio between the number of floating-point operations and the amount of transferred data [20]. In our case, we define it as

$$\text{arithmetic intensity} = \frac{\text{\#cycles spent on computations}}{\text{\#cycles spent on memory transfers}}.$$

Type	Computations	Memory accesses
mul	4 multiplications	4 integer loads 8 floating-point loads 4 floating-point stores
div	4 divisions	4 integer loads 8 floating-point loads 4 floating-point stores
matrix	16 multiplications 12 additions	4 integer loads 20 floating-point loads 4 floating-point stores
mul-max	4 multiplications 4 comparisons 4 if-then-else branches	4 integer loads 8 floating-point loads 4 floating-point stores

Table 5.2: Operations performed by the types of blocks in the simulation application for the sample task.

Depending on its arithmetic intensity, an application running on a specific architecture is either CPU-bound or memory-bound. For a value greater than 1, we define the application to be CPU-bound; else it is memory-bound. If the arithmetic intensity is below the threshold, the performance (measured in floating-point operations per seconds) is bounded by the memory bandwidth because more time is spent transferring data than doing computations on the data. Beyond that threshold, the application becomes CPU-bound and its performance becomes limited by the maximum number of floating-point operations that can be executed by the processor pipeline.

Knowing whether an application is memory-bound or CPU-bound on a specific architecture is important information because it tells us which part of the architecture should be optimized. For instance, if an application is memory-bound, it is a waste of resources to add additional execution units in the CPU execution engine. To get an idea of the behaviour of the test application, we will compute the arithmetic intensity for each of the four block types in the simulation application for the MPC8548 platform.

5.1.3 Analysis of the test application on the MPC8548

More on the MPC8548

As we know from Section 3.2.2, the MPC8548 has a two-level memory hierarchy. In the original application, it was made sure that the entire application, including the data, could be stored entirely in the L2 cache. Therefore, we assume that no memory fetch is from main memory. All accesses are from either L2- or from L1 cache. The cache line size of the MPC8548 is 32 bytes. Since we work with 64-bit data items, doing one fetch from L2 cache allows us to do the subsequent 3 fetches from L1 cache.

From the operations listed in Table 5.2, we can identify the following necessary instructions.

- double-precision floating-point load
- double-precision floating-point store
- double-precision floating-point addition
- double-precision floating-point multiplication
- double-precision floating-point division
- double-precision floating-point compare

Table 5.3 shows the latencies and the throughputs of these instructions as given by the datasheet of the processor. Since the execution units are pipelined, multiple instructions can be executing in the execution unit simultaneously (except for the division operation). The throughput is the number of cycles before a new instruction can be inserted in the pipeline of the execution unit. We make a difference between a load from L1 cache and L2 cache. Loading a data item from L1 cache takes only 3 clock cycles while it takes 20 cycles to load it from L2 cache.

Instruction	Operation	Execution unit	Latency	Throuhgput
evldd	load	load/store unit	3 (L1) 20 (L2)	1 (L1) 5 (L2)
evstdd	store	load/store unit	3 (L1) 20 (L2)	1 (L1) 5 (L2)
efdadd	addition	multiple unit	6	1
efdmul	multiplication	multiple unit	6	1
efddiv	division	multiple unit	32	32
efdcmp	comparison	multiple unit	6	1

Table 5.3: Latencies and throughputs of the instructions of interest [6]. The throughput is the number of cycles before a new instruction can be inserted in the pipeline of the execution unit.

Analysis

Figures 5.2, 5.3a and 5.3b show the analysis for the *mul* block type using the numbers from Table 5.3. In Figure 5.2, we computed how many cycles it takes to load the input and the parameters. Since the data between the blocks are passed around with pointers, we first need to load the addresses of the input data and then load the actual data. Also note that we assume that all the necessary data is stored in the L2 cache.

Figure 5.3a shows the computation of the output. In the case of *mul*, this consists of 4 independent multiplications which can therefore be pipelined. The MPC8548 has only one execution unit for floating-point operations. Therefore, the instructions can not be executed in parallel.

Finally, Figure 5.3b shows the sequence of operations when storing the output data. It is similar to the sequence of loading the data in Figure 5.2.

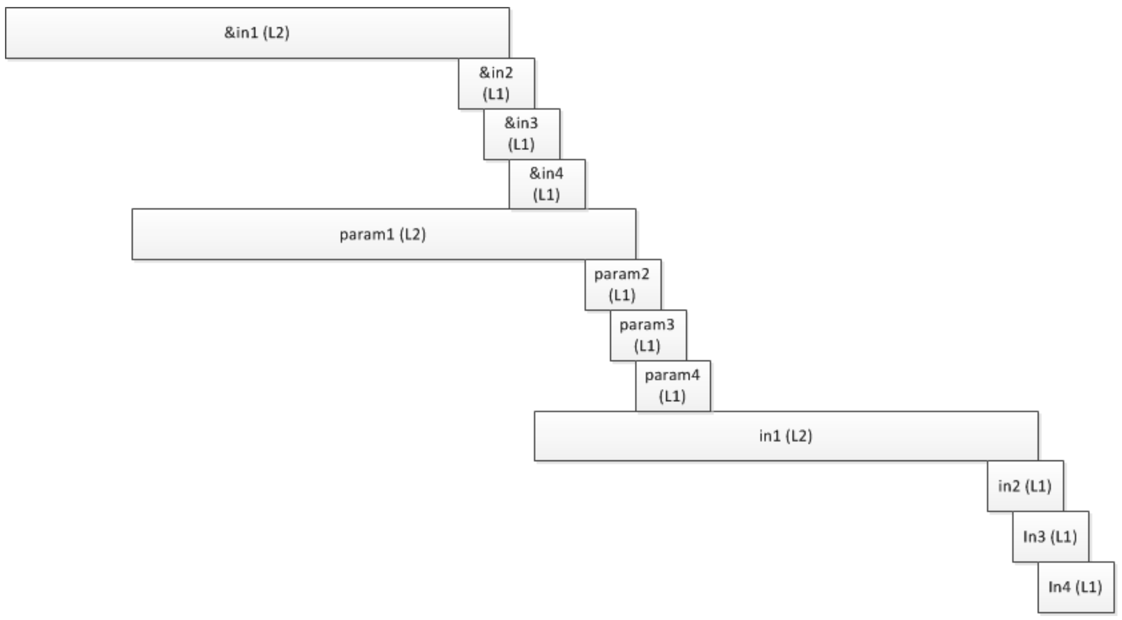
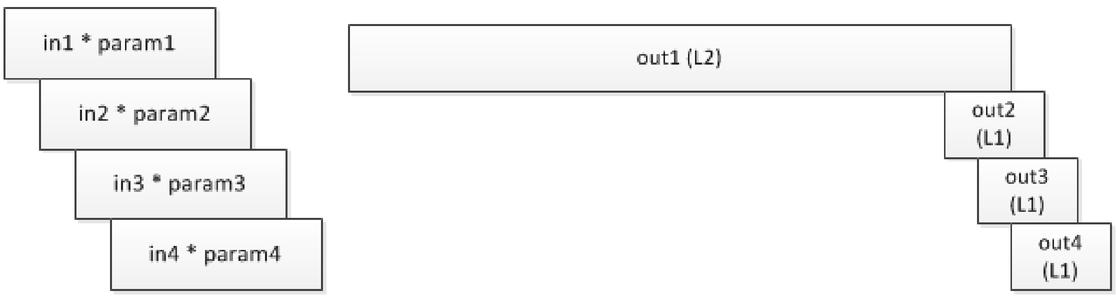


Figure 5.2: The simulation application of the sample task is a chain of blocks. Each block takes 4 inputs and produces 4 outputs. There are four different kinds of blocks and they are chained consecutively.

Table 5.4 shows the result of the previous analysis for each block types. Based on the latencies, the arithmetic intensity of each block type has also been computed. An arithmetic intensity lower than 1 means that the block is memory-bound, else it is CPU-bound. The analysis shows that the block types *mul*, *matrix* and *mul-max* are memory-bound. This means that for these blocks, the performance of the memory hierarchy is very important. The *div* block type is CPU-bound but that is mainly because the division operation is slow and it is not pipelineable. This block will benefit from an execution unit with a fast divider.

Type	Computations	Memory accesses		Arithmetic intensity
		Loads	Stores	
mul	9	44	23	0.13
div	128	44	23	1.91
matrix	42	49	49	0.58
mul-max	16	44	23	0.24

Table 5.4: Arithmetic intensity of the different types of blocks. An arithmetic intensity lower than 1 means that the block is memory-bound, else it is CPU-bound.



(a) Data (b) Code

Figure 5.3: Data and code sizes of the benchmark applications on the different platforms.

5.1.4 Conclusion

From the analysis done for the MPC8548, we can conclude that the benchmark is mainly memory-bound. This means that the memory performance of an architecture will be the most important factor in its performance for the HPPC application. Only the *div* block is computation-bound; its performance will depend mostly on how fast the floating-point unit can perform a division.

5.2 Benchmarking methodology

First of all, the entire test application, i.e. a sequence of the four types of blocks, will be used as a benchmark. Then, in order to treat the different blocks in isolation, simulation applications consisting of solely one type of block will executed as well. Since there are 4 types of blocks, a total of 5 benchmarks will be run.

Just as for the benchmarks for the host application, we measure the performance of a single core. This means that for the P4080 and the Xeon E5540, only one core will be used.

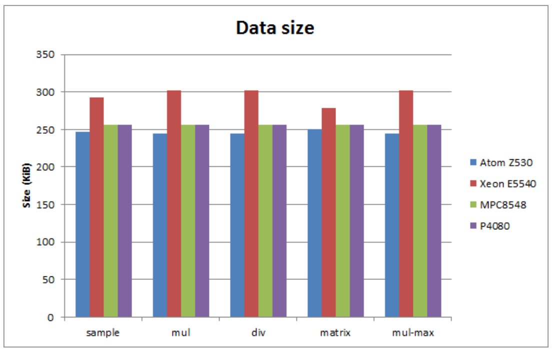
The real sample task has been optimized in such a way that it fits in the 512 KiB L2 cache of the MPC8548. Furthermore, profiling has shown that about 256 KiB of the application is code and 256 KiB is data. To mimic these conditions, we will build the test application in the exact way, i.e. 256 KiB of code and 256 KiB of data.

When building the test application, the number of data blocks does not have to be equal to the number of code blocks. As mentioned before, a data block is a C structure while a code block is a C function that accepts a data block as input and computes its outputs. Having more data blocks than code blocks means that we reuse the same code blocks for multiple data blocks, i.e. each function processes multiple data blocks. The reverse it not possible; we cannot have more code blocks than data blocks because the code blocks only exist because of the data blocks.

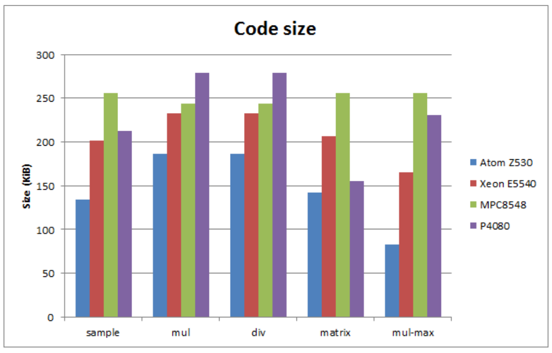
Table 5.5 gives the resulting applications that are going to be benchmarked. As mentioned before, the number of blocks has been chosen such that each application has a footprint of 512 KiB on the MPC8548. For instance, in order to have a data footprint of 256 KiB and a code footprint of 256 KiB, the *matrix* benchmark consists of 1424 data blocks and 828 code blocks. This means that a total of 1424 different C structures will be processed by 828 different C functions. Therefore, some functions will process more than one data block. To be precise, the first 596 code blocks will process two data blocks while the rest of the code blocks will process only one data block.

Type	Block count		Application size		
	Data	Code	Data	Code	Both
sample	2340	1432	255.9 KiB	255.9 KiB	511.9 KiB
mul	2978	2978	255.9 KiB	244.3 KiB	500.2 KiB
div	2978	2978	255.9 KiB	244.3 KiB	500.2 KiB
matrix	1424	828	255.9 KiB	255.5 KiB	511.4 KiB
mul-max	2978	1056	255.9 KiB	255.8 KiB	511.7 KiB

Table 5.5: Sizes of the applications to be benchmarked on the MPC8548. The first two columns represent the size in bytes of an individual data and code block. The next two columns are the number of data and code blocks of the application. The last three columns give the total application size.



(a) Data



(b) Code

Figure 5.4: Data and code sizes of the benchmark applications on the different platforms.

After this step, the same applications as in Table 5.5, i.e. the same number of blocks, were built for the other applications. Because of differences in instruction sets, each application will have a different code size on a processor. This is shown in Figures 5.4a and 5.4b where respectively the size of the data and code is shown. Based on the latter and Table 5.5, we can make the following observations.

- The floating-point instruction set on Intel processors (the x87 instruction set) is stack-based. This reduces code size because it requires less operands in the instructions. Therefore, the code size on the Atom and the Xeon is relatively small compared to the data size.
- Since the Xeon is a 64-bit architecture, its pointers are 8 bytes instead of 4 bytes on the other architectures. Therefore, each data block has a bigger size than on the other architectures.
- On the *matrix* benchmark, which performs a matrix multiplication, the code size of the P4080 is relatively small compared to the other benchmarks. This is because it has a dedicated

multiply-add instruction.

5.3 Expected results

Using the clock cycle counts of the different processors, we calculated an estimation of the number of clock cycles needed to execute each block. For simplification, we divided each block in three parts.

- Loading the input and the parameters
- Doing the computation
- Storing the output

For each part, we calculated the number of cycles to execute it. Then, we defined the latency for that block to be the sum of the three parts. This is a simplification because it does not take into account overlapping of the parts (e.g. overlap loading of input and parameters while already performing some computations). Based on this total, the relative performance of each processor compared to the MPC8548 is shown in Table 5.6. For the Atom Z530, the Xeon E5540 and the P4080, we do not have exact figures for the L2 cache throughput. We cannot use the results of the *LMBench* benchmarks as the data to be accessed is not sequential: the input data and the parameters are not stored consecutively in memory. Therefore, two extremes were taken. First, one result was calculated where we assume that the L2 cache is not pipelined; its throughput is its latency. Secondly, another result was calculated where a new access to L2 cache can be made within 1 clock cycle.

Type	Atom Z530	Xeon E5540	MPC8548	P4080
mul	94 - 110	141 - 173	100	106 - 125
div	60 - 62	146 - 157	100	61 - 63
matrix	70 - 111	112 - 163	100	67 - 87
mul-max	100 - 117	148 - 180	100	101 - 117

Table 5.6: Expected performance on the four different types of blocks.

By analyzing Table 5.6, we can learn about the performance of each architecture for the benchmark compared to the MPC8548:

Atom Z530 Overall, it is expected that it will perform better than the MPC8548 except on the *div* benchmark because its division instruction is relatively slow (see Table 4.10). Since the other block types are memory-bound (as we know from Table 5.4), the Atom will be faster due to its better memory performance.

Xeon E5540 On all benchmarks, it is expected to perform better than the MPC8548 because of its better performance on both computations and memory.

P4080 Its performance is expected to be very similar to the performance of the Atom processor.

There are some limits to the way we calculated the expected results:

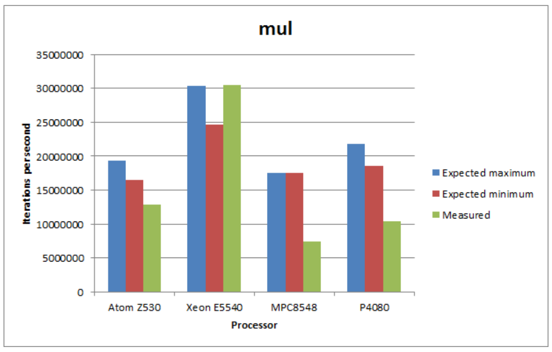
- For all architectures, we assume the whole application fits in L2 cache. This is a correct assumption for the Atom Z530 and the MPC8548 but it is not correct for the Xeon E5540 and the P4080. For those processors, around respectively 40% and 60% of the application resides in L3 cache.
- Each code block is implemented in its own function but we do not take into account the overhead caused by the function calls (i.e. return from the function, retrieve the address of the next function, jump to that address, etc. . .).
- We make the assumption that we never have to wait for an instruction to load. In reality, just like the data, not all the instructions will be in L1 cache and therefore will cause cache misses. It is expected that this assumption will have a relatively big impact on the accuracy of the estimations.

5.4 Results

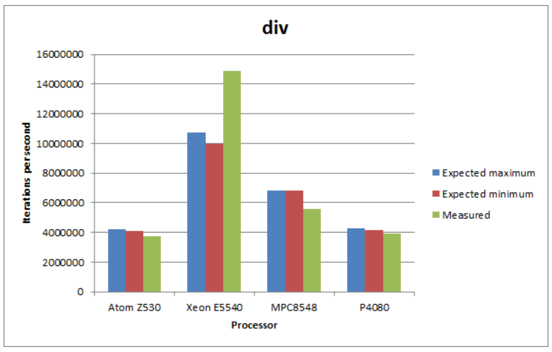
We benchmarked each application by executing the chain of blocks repeatedly for 1 second. We define the performance as the number of block executions during that timespan.

Because a processor with a higher clock frequency is inherently faster than a comparable processor with a lower clock frequency, the results were scaled to the same clock frequency. This way we can isolate the architectural features of a processor from its clock frequency. We took the clock frequency of the MPC8548 as a reference, i.e. we performed the following transformation:

$$\text{result}_{\text{new}} = \frac{f_{\text{MPC8548}}}{f_{\text{processor}}} \text{result}_{\text{old}}$$



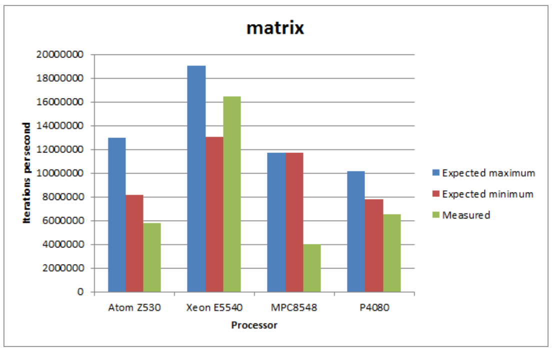
(a) *mul* benchmark



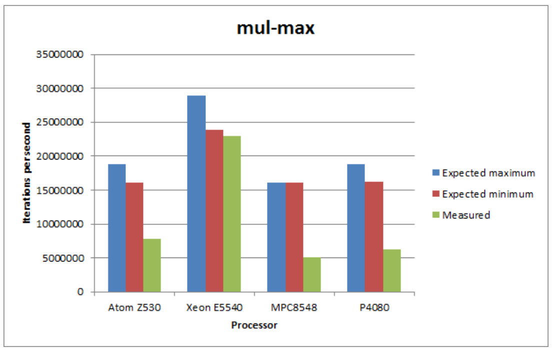
(b) *div* benchmark

Figure 5.5: Comparison of the expected results with the measured results for the mul and div benchmarks.

In Figures 5.5a, 5.5b, 5.5c and 5.5d, we compare the expected results with the measured results. The overall remark is that we overestimated the memory performance for the Atom, the MPC8548



(c) *matrix* benchmark



(d) *mul-max* benchmark

Figure 5.5: Comparison of the expected results with the measured results for the matrix and mul-max benchmarks.

and the P4080. For every benchmark except for *div*, which are the benchmarks that are memory-bound, the measured performance is lower than the expected performance. Besides the reasons mentioned above, there are additional hypotheses as to why this is the case:

- The benchmarks were run in a Linux environment. Care was taken to make sure that no other processes were running concurrently but on any Linux system there are always daemons and kernel processes running in the background. Each of these processes pollutes the cache which causes additional misses.
- The data is not aligned according to the size of a cache line. Therefore, there may be more L1 cache misses than was assumed.

Type	Atom Z530	Xeon E5540	MPC8548	P4080
sample	6.2	23.9	5.2	6.3
mul	12.8	30.4	7.4	10.3
div	3.8	14.9	5.6	3.9
matrix	5.8	16.5	4.0	6.5
mul-max	7.8	23.0	5.0	6.2

Table 5.7: Results of the benchmarks for the sample task. The numbers were scaled by a factor 1,000,000 in order to make them more readable as the absolute values of the results have no significance.

To make a better comparison between the architectures, the measured results are shown again in Table 5.7 and depicted visually in Figure 5.6. We can conclude the following:

Atom Z530 As we predicted, the Atom Z530 performs better than the MPC8548 except on the *div* benchmark. On overall, its performance is comparable to the performance of the P4080. On the benchmarks where there are not many computations (i.e. all except *matrix*), it does not

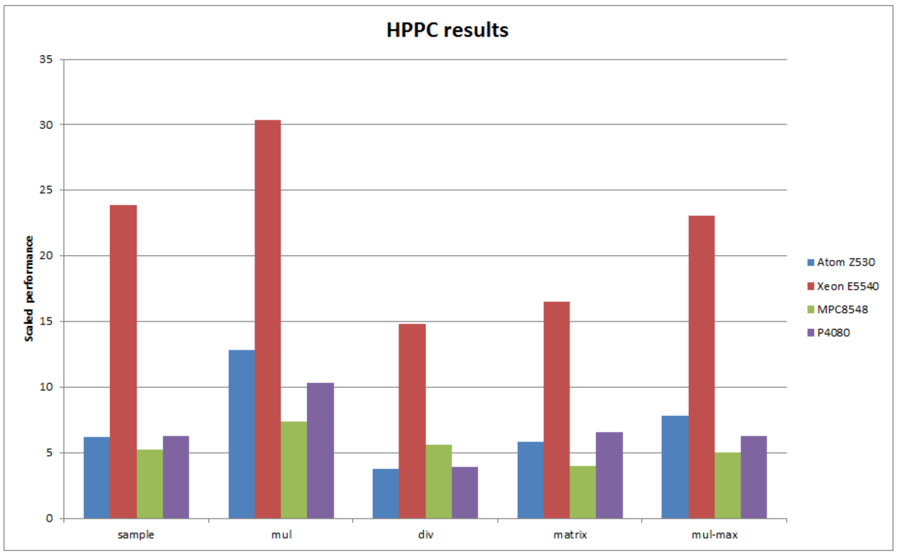


Figure 5.6: Results of the benchmark applications in Table 5.7.

suffer from its in-order execution flow. However, on *matrix*, the P4080 has a clear advantage and it is expected that this is because of its out-of-order execution.

Xeon E5540 The Xeon E5540 performs the best by far. On all applications, it performs between 2.5 and 4 times better than the MPC8548. On the memory-bound applications, i.e. *mul*, *matrix* and *mul-max*, it is interesting to see that the difference in performance increases as the arithmetic intensity decreases. For instance, the *mul* application has the lowest arithmetic intensity, meaning that it is the most memory-bound, but the Xeon is about 4 times faster than the MPC8548 on that test. This indicates that the big difference between the MPC8548 and the Xeon lies more in the performance of the memory hierarchy than in the computation power. This is somehow surprising since the application does not fit entirely in the L2 cache (the Xeon L2 cache size is 256 KiB). However, there is an L3 cache of 8 MiB that amortizes the costs to main memory. The *div* application is faster both because of the better memory performance and because it has a faster division operation (22 cycles for the Xeon and 32 cycles for the MPC8548).

P4080 Its behavior is very similar to the behavior of the Atom Z530 as we had predicted. On *mul* and *mul-max*, it suffers from its smaller L2 cache size (128 KiB) by more L2 cache misses, even though its bandwidth higher than the Atom processor. On *matrix*, it takes advantage of its out-of-order execution flow.

5.5 Conclusion

All in all, the conclusion is the same as for the host application (see Section 4.6). The Xeon is the preferred architecture if we can step away from the AdvancedTCA specification and if the poverty of the Intel documentation and the shorter duration of supply are not an issue. In the other case, the P4080 is the preferred solution. The Atom Z530 is equivalent to a core of the P4080 but since space is an important constraint, the P4080 has the advantage because it has 8 cores on a single chip.

5.6 Future work

In this chapter, we have used a model of the HPPC application as our benchmark. The model approximates its behaviour in terms of data accesses and computations but it is not an exact copy of the application. Also, we have executed the benchmarks in a Linux environment while the real application runs bare-board. Therefore, to validate the results, it would be necessary to perform similar measurements with a real HPPC application in a bare-board environment.

In the benchmarks, we have only focused on the performance of the application on the different processors. However, we did not take into account the real-time requirements of the application. In particular, previous studies performed by the owner of the system revealed that the interrupt latency of the platform, i.e. the time between the arrival of doorbell signal indicating the start of a new period of execution and the execution of the sample task, is significant for the performance. There must be a guarantee on an upper bound that is sufficiently low. The interrupt latency is very dependent on the architecture and therefore future work should perform measurements of this latency on the processors that were discussed.

Chapter 6

Porting the HPPC application to a multi-core platform

There is an important difference between the host and the HPPC application. The host is very easily portable to a multi-core platform: the Linux scheduler can automatically spread out the different application processes over the available cores. On the other hand, the HPPC application has real-time constraints and runs bare-board, i.e. it is supported by a minimal library that composes the operating system. Therefore, efficiently porting it to a multi-core platform is not trivial and it will be the topic of this chapter.

Section 6.1 presents two alternatives for a mapping to a multi-core platform. Each alternative introduces different overheads. These are analyzed and measured in Section 6.2. Using the results, the two alternatives are compared in Section 6.3. Section 6.4 analyzes the scalability of the sample task and Section 6.5 discusses the topic of synchronization for our application. Finally, Section 6.6 presents a conclusion of the chapter and Section 6.7 discusses future work arising from the analysis.

6.1 Two alternatives for a mapping

6.1.1 Introduction

On a software level, the HPPC application consists of two types of tasks:

sample task This task performs the actual computations. Referring back to Figure 2.1, this task executes the *post-processing*, *set parameters* and *pre-processing* steps. In reality, the *set parameters* step consists of several background tasks but since they execute in the sample context, we consider them as part of the sample task.

background tasks These tasks serve as an interface between the sample task and the host application. Each task is the implementation of an RPC call. They are used by the host application to either retrieve information about the sample task or to pass information to the sample task. In

Figure 2.1, the background tasks execute at the end of the execution period in the *background processing* step.

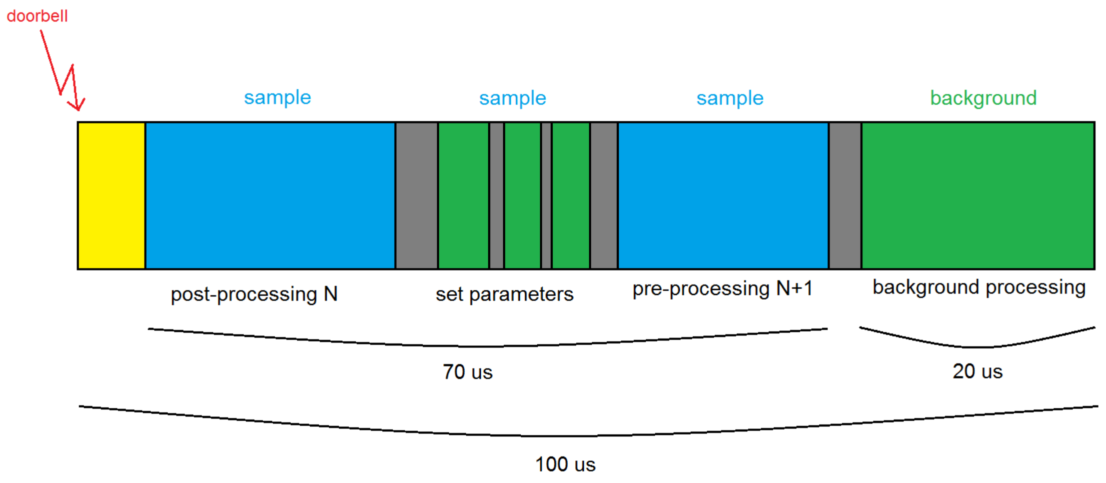


Figure 6.1: Timing details of the current implementation of the HPPC application. The sample task is shown in blue, the background tasks in green, the overhead due to the interrupt latency is shown in yellow and the context switching latencies are shown in grey.

Figure 6.1 refines the illustration shown in Figure 2.1 and depicts the timing properties of the HPPC application as it is currently implemented. It shows that the *set parameters* step consists of several background tasks executing in the sample context. After the *post-processing* step, the sample task executes these background tasks. Each task updates the parameters in the computation blocks in order to start processing the next sample. Because they execute in the sample context, they cannot be interrupted during execution. The tasks perform operations that need to happen atomically. When all of them have finished their execution, the sample task resumes with the *pre-processing* step.

The timing properties are as follows.

- The doorbell frequency is 10 KHz. This means that one execution cycle takes 100 μs.
- A sample period of 70 μs is defined. This period contains the *post-processing*, *set parameters* and *pre-processing* stages.
- A period of 20 μs is reserved for the background tasks running at the end of the execution period. They execute after the sample task until the next doorbell notification.
- The remaining 10 μs are assigned to the various overheads. These include:
    - The interrupt latency of the doorbell notification, i.e. the time between the arrival of the interrupt and the start of the execution of the sample task.
    - The context switching latencies between the various tasks.

6.1.2 Parallelizing the application

In Section 2.1 we explained that the sample task is implemented as a network of computational blocks. Such a network has a graph-like structure, meaning that it has parallel paths and synchronization points. In the current implementation, the graph is serialized in order to execute on the single-core MPC8548 processor. However, on a multi-core platform, it becomes possible to execute parallel paths in the graph simultaneously on separate cores. How this is done exactly for each computation network is application-dependent and is outside the scope of this thesis. Instead, we make the assumption that, given any number of cores, we can divide the sample task (i.e. the *pre-processing*, *set parameters* and *post-processing* steps) evenly among the available cores. Also, synchronization and communication aspects between the cores will be discussed later.

After having divided the sample task over the cores, we have to map the *background processing* step to the cores. We chose to investigate two alternatives:

i. Assign a set of background tasks to each core (see Figure 6.2).

ii. Reserve a separate core for the background tasks that will serve the sample tasks on all cores (see Figure 6.3).

In the next paragraphs, we will compare both solutions in terms of performance. Our definition of performance is the meaningful computation time that is available for the sample task. Thus, referring to Figure 6.1, we are interested in maximizing the time that is available for the execution of the *pre-processing* and *post-processing* steps. Each solution introduces different components of overhead. These are necessary but they prevent the sample task from running. In the next paragraph, we will analyse and experimentally measure these overheads in order to compare the performance of the two solutions.

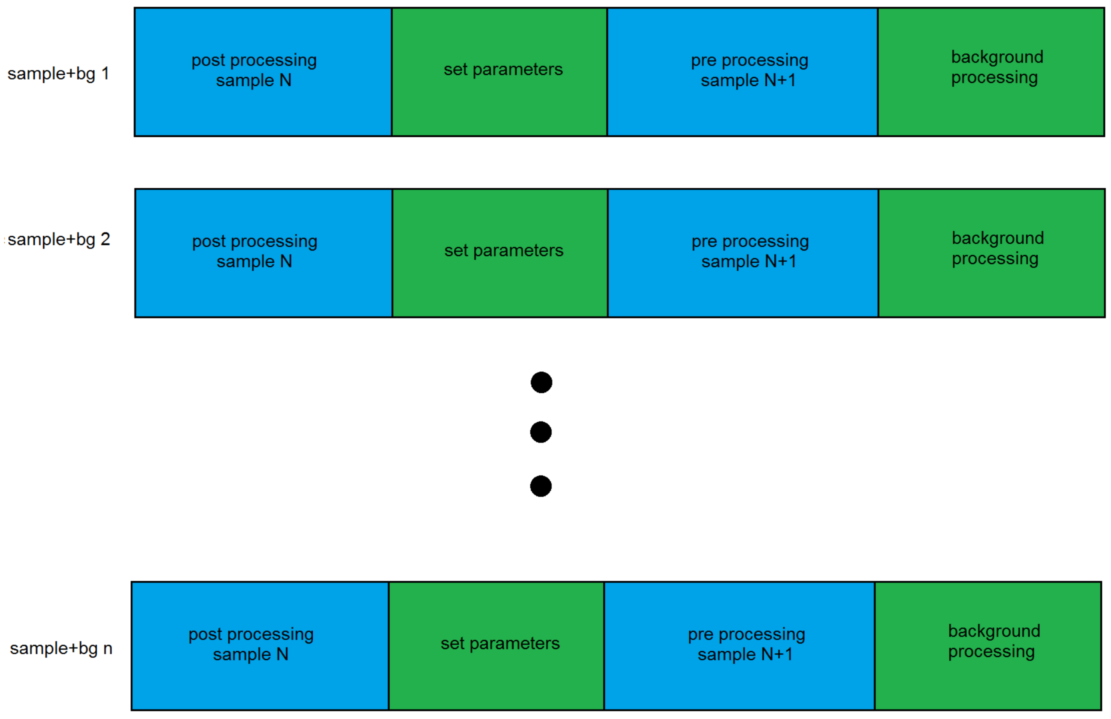


Figure 6.2: Illustration of the first solution, where we assign a set of background tasks to serve each sample task.

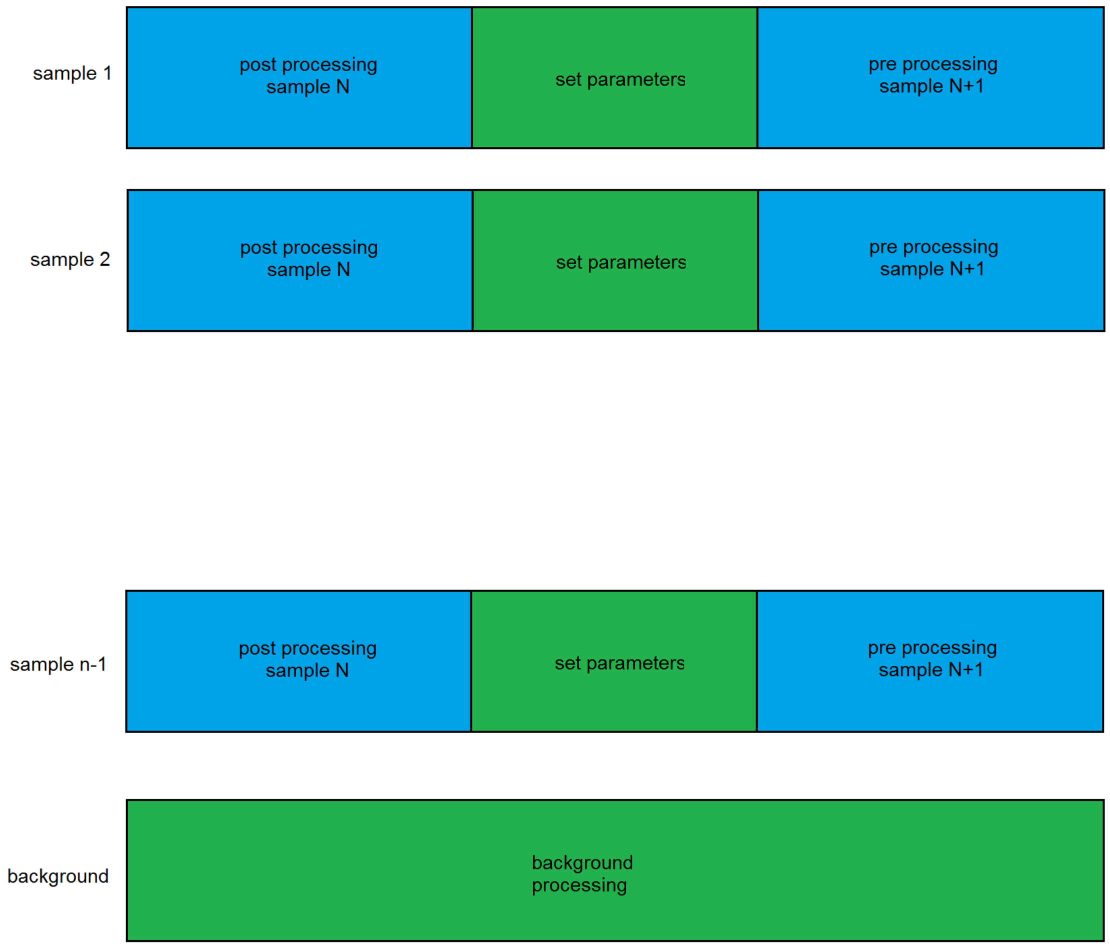


Figure 6.3: Illustration of the second solution, where we perform the whole background processing step on a separate core.

6.2 Measurements of the overheads

6.2.1 Introduction

Figures 6.4 and 6.5 refine respectively Figures 6.2 and 6.3 by showing the different overheads involved in the execution. Contrary to Figure 6.1, we lump the different background tasks executing during the *set parameters* step into one task. This is done for simplicity of our model as we assume that this step always takes the same time, independent of the sample task running on the same core.

For both solutions, we can compute the total sample time as we have defined above.

$$\text{solution 1 sample time} = n(100\ \mu\text{s} - \text{overhead} - \text{set parameters} - \text{background processing}) \quad (6.1)$$

$$\text{solution 2 sample time} = (n-1)(100\ \mu\text{s} - \text{overhead} - \text{set parameters}) \quad (6.2)$$

where n is the number of cores.

For solution 1, the overheads are as following:

- The interrupt latency (orange), which is the average time between the arrivals of the doorbell interrupt and the execution of the sample task.
- After the *post-processing* step, a context switch to the background task that will set the parameters (red).
- Then, the parameters will be loaded (yellow). As the background task and the sample task execute on the same core, it is assumed that the parameters will be available in L2 cache.
- At the end of the *set parameters* step, a context switch to return to the sample task (red).
- At the end of the sample task, a context switch to execute the *background processing* step (red).

For solution 2, the overheads are as follows:

- The interrupt latency (orange), which is the same as for the first solution.
- After the *post-processing* step, a context switch to the background task that will set the parameters (red).
- Then, the parameters will be loaded (yellow). As the background task and the sample task execute on different cores, the parameters will have to be loaded from the L3 cache. They will have to be moved from the background core to the core on which the sample task executes.
- At the end of the *set parameters* step, a context switch to return to the sample task (red).

As a conclusion, in order to calculate the different overheads, we need to know the following:

- Interrupt latency
- Context switching latency

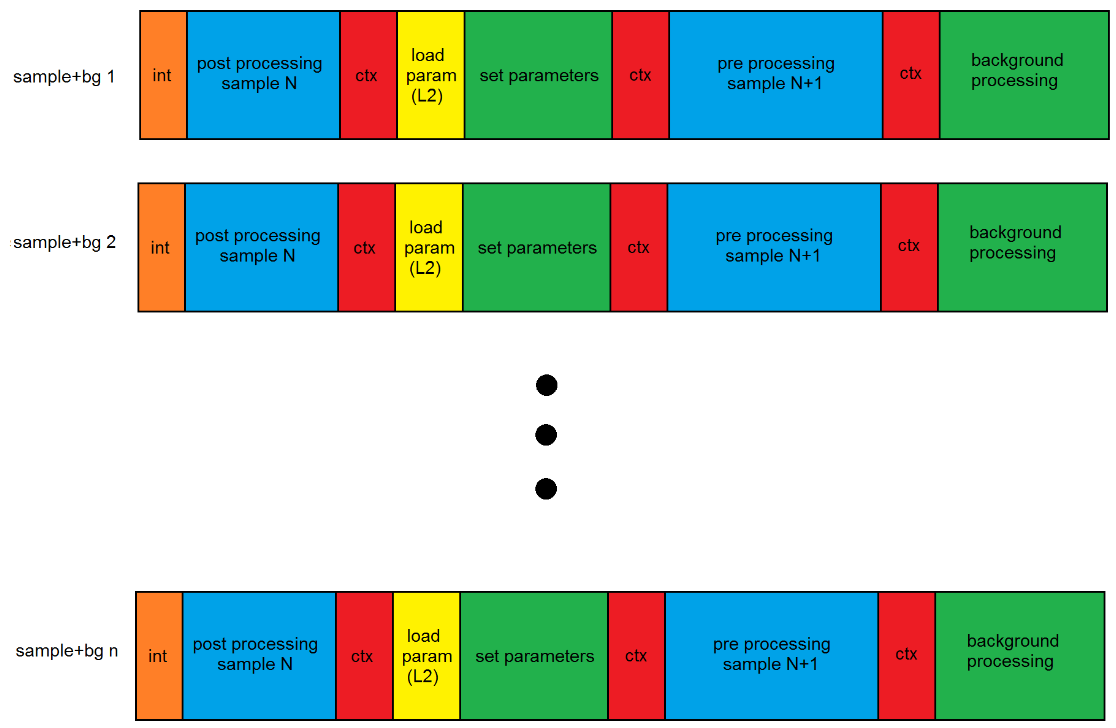


Figure 6.4: Overheads involved in solution 1.

- L2- and L3-cache bandwidth and size of the parameters to calculate the time to load the parameters
- Time to execute the *set parameters* step.

The first three will be measured experimentally. The time to execute the *set parameters* step will be estimated based on the current implementation on the MPC8548 processor.

6.2.2 Environment of the measurements

To perform the measurements, we use the P4080 octo-core processor. During the thesis, HPPC-OS, i.e. the operating system that is currently used for the HPPC modules, was being ported to the P4080. By the time that we were going to perform the measurements, a functional version of the OS was ready so we used that kernel instead of a Linux kernel. The advantages of this approach are the following:

- The environment of the measurements matches the environment into which the application will be deployed. This is especially important as the context switching latency and the interrupt latency are dependent on the operating system.
- As the operating system is non-preemptive, it does not cause any overhead. When performing measurements in a Linux environment, there are many external factors that influence the

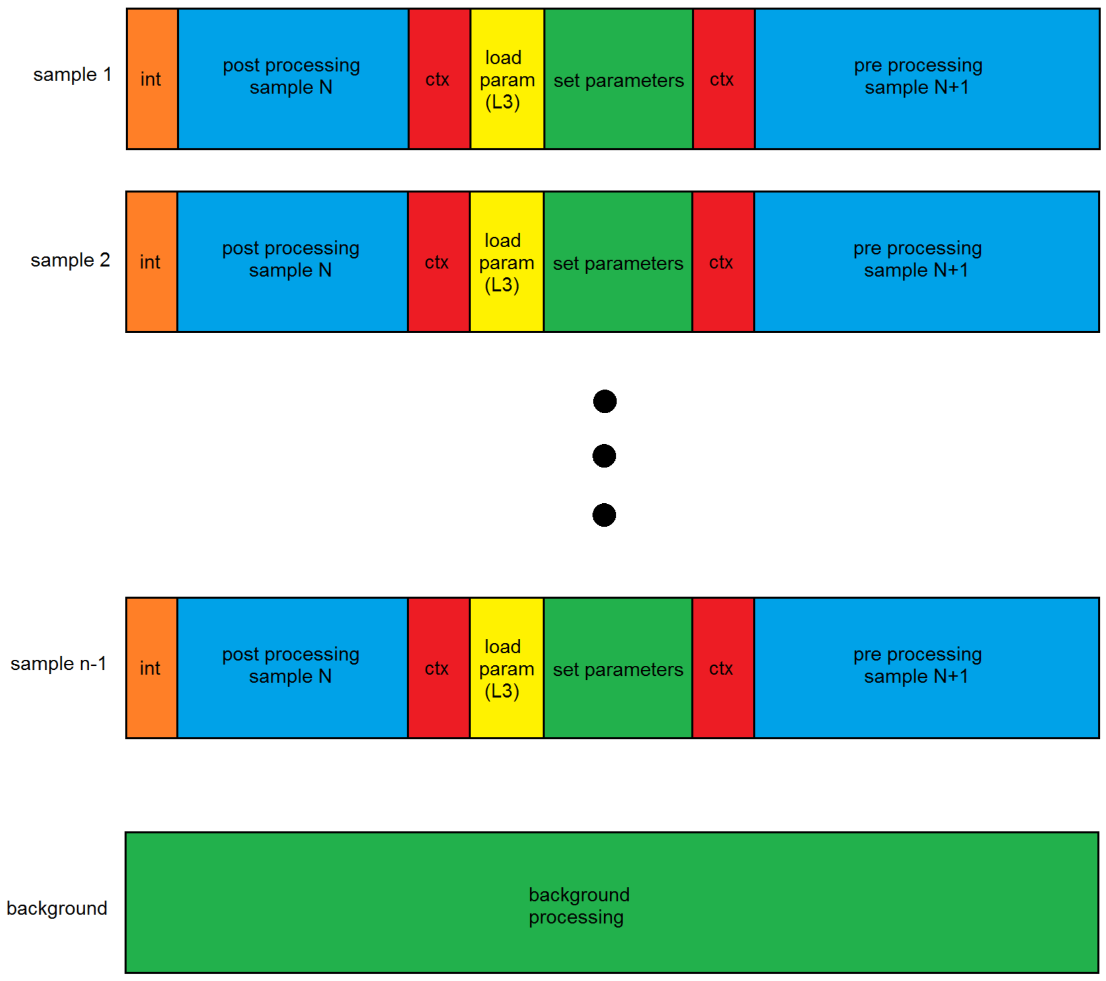


Figure 6.5: Overheads involved in solution 2.

measurement: background processes and kernel threads, interrupts, cache pollution,. . . In our setup, these influences do not exist. It is almost as if we run the application directly on the hardware.

To perform the timings, the processor has timers that run at the same frequency as the processor itself. This allows us to perform timings with a high degree of precision.

6.2.3 Interrupt latency

The first measurement concerns the interrupt latency. Inside the processor, interrupts are handled by a special unit called the Multicore Programmable Interrupt Controller (MPIC). This unit receives interrupts and forwards them to the appropriate core. In the HPPC application, the doorbell notification comes as an external interrupt to the MPIC. We define the interrupt latency as the time between the arrival of the interrupt signal at the MPIC and the start of the execution of the sample task. Because of practical reasons, we could not trigger the processor with an external interrupt. Instead, we use a mechanism called Interprocessor Interrupts (IPIs) which allows cores to send interrupts to each other. The path of such an interrupt is shown as the red line in Figure 6.6. Its path is longer than the intended path with an external interrupt (blue line): the signal has to travel from the source core over the interconnect to the MPIC. However, although we cannot measure it, we expect this difference in latency to be negligible compared to the total interrupt latency. The reason is that the interrupt is an electrical signal propagating through the interconnect, therefore causing a relatively low latency.

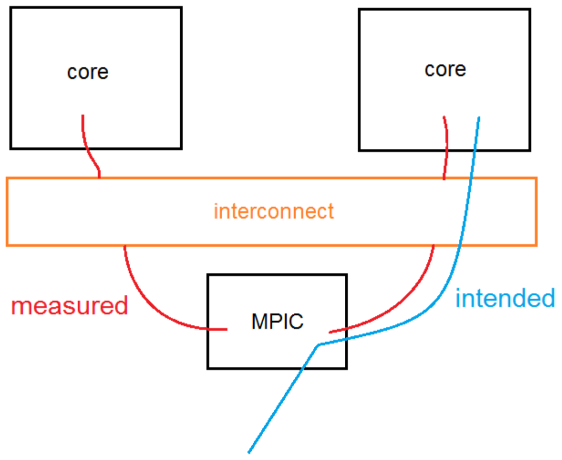


Figure 6.6: Illustration of the interrupt mechanism. The blue line is the latency that we are trying to measure. The red line is the latency that we measured.

Starting from the moment that an interrupt arrives at the MPIC, the following events happen:

i. The interrupt signal travels from the MPIC to the destination core. This time is expected to be small compared to the total latency as it is an electric signal travelling with a very high speed. Also, the interconnect is not expected to cause much delay. However, the datasheet does not give details about the routing of interrupts so we can not be sure of its exact delay.

ii. The destination core executes an initial routine that first saves the current context on the stack (saving the registers) and then fetches the interrupt vector from the MPIC.

iii. It then jumps to the address given by the interrupt vector and executes the service routine. We have chosen to implement an empty service routine: it returns immediately. The reason is that we want to measure the latency of a minimal interrupt, i.e. one that does not execute additional interrupt handling code.

iv. After the execution of the service routine, the context is restored and the End-Of-Interrupt (EOI) register in the MPIC is written to indicate that the interrupt has been handled.

v. The execution goes back to the task that was running before the interrupt happened.

To perform the measurement, we want to record the time right before the source core sends an IPI. Then, we want to record the time after the destination core has received and processed the interrupt, i.e. after the steps enumerated above have been executed. For that, we would need a global clock accessible to all cores. However, the P4080 does not have such a clock. Instead, each core has its own private high-resolution clock.

To overcome this problem, we ran a "ping-pong" experiment. This means that the core that receives the IPI immediately sends back an IPI to the source core. The source core records the time upon sending the IPI and upon receiving the interrupt sent by the destination core. This way, the latency can be measured on one core only. The measured latency is then divided by two to get the latency for a single interrupt.

Even though we are running the experiment almost directly on the hardware, there is still statistical variation in the results. It is expected that the variation comes mainly the synchronization between the cores and the MPIC as the input clock frequency of the MPIC is half the clock frequency of the cores. Another source of variation can be the different states of the cache across runs. To get an idea of the resulting distribution, we chose to perform 1000 subsequent measurements. The single interrupt latency was calculated with the following formula:

$$\text{single interrupt latency} = \frac{\text{measured latency} - \text{timing overhead}}{2}$$

The timing overhead corresponds to the time to read the timer registers and was measured to be 20 clock cycles. Figure 6.7 shows the histogram of the single interrupt latency. It can be seen that the distribution is approximately normal. The average is 1066 clock cycles while the minimum and maximum are respectively 1059 and 1076 clock cycles. Assuming a normal distribution, the standard deviation is 2.49. From that, we can deduce that for more than 99% of the time, the interrupt latency will fall between 1059 and 1073 clock cycles.

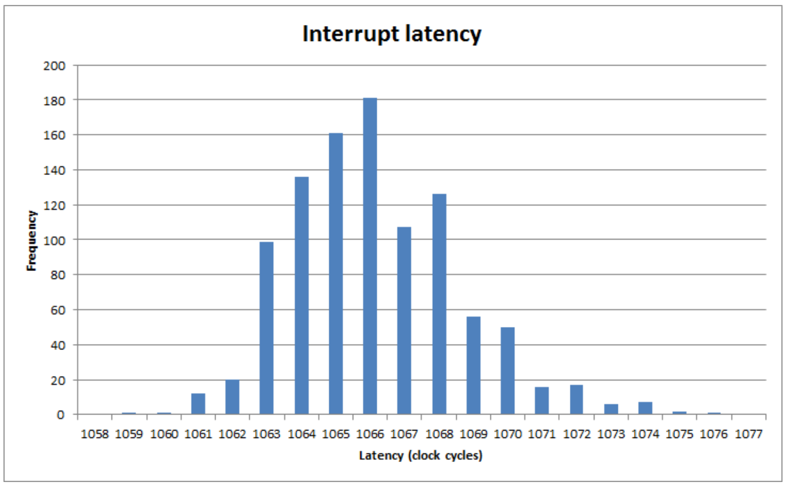


Figure 6.7: Histogram for 1000 measurements of the latency of an IPI interupt. The latency is given in clock cycles. The average is 1066 cycles.

6.2.4 Context switching latency

The next measurement is the context switching latency. For that, we use the task management library of the operating system. During a context switch, the following actions must happen:

- Save the context of the old task into its stack.
    - Save the link register and the condition register.
    - Save the general-purpose registers.
    - Save the floating-point registers.
- Save the stack pointer of the old task.
- Load the stack pointer of the new task.
- Restore the context of the new task in the appropriate registers.

In order to perform the measurement, we created two tasks that repeatedly call the function `HP_task_yield()`. This system call yields the processor to another task, thus forcing a context switch. In the implementation of the system call, code to perform the timing was inserted. Just as for the interrupt latency, the context switching latency was measured 1000 times. The histogram is shown in Figure 6.8. Just like for the measurement of the interrupt latency, the timing overhead of 20 clock cycles has been subtracted. The histogram shows that the average context switching latency is about 222 cycles. However, it shows much less spreading than the interrupt latency: only three different latencies were measured, all within a range of 5 clock cycles.

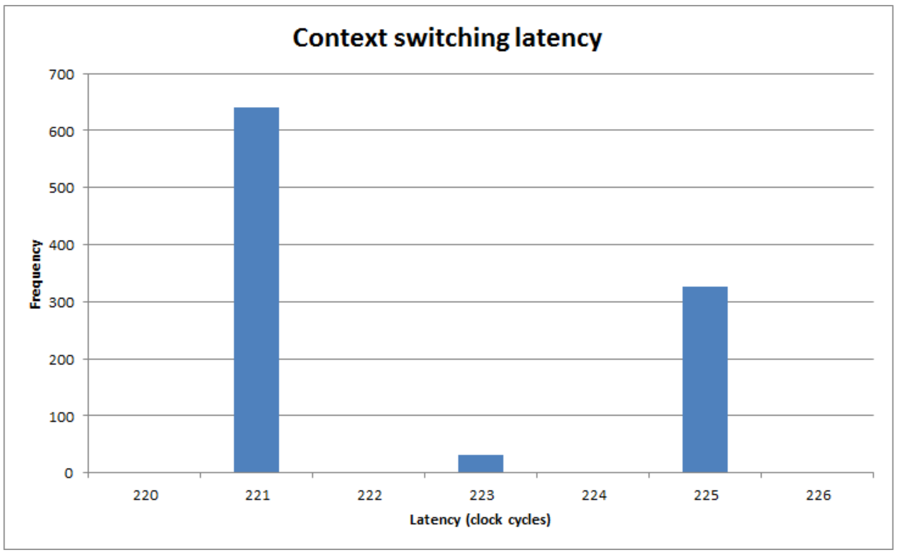


Figure 6.8: Histogram of the context switching latency.

6.2.5 Cache bandwidth

In the first solution, the parameters for the *set parameters* step are loaded from the local L2 cache while the L3 cache needs to be accessed for the second solution. With this measurement, we wish to compare the bandwidth of the L2 cache and the L3 cache. As the parameters are floating-point values, we will measure the bandwidth for 64-bit floating-point data. Note that we already measured the bandwidth for the P4080 in Section 4.4.4. However, that measurement did not include the inter-core bandwidth because we only measured the bandwidth on a single core. Also, we believe that we can make a more precise measurement in our current environment as we run the experiment bare-board.

To perform the measurement, we simulate a producer-consumer situation. We assume that the producer and the consumer tasks share a piece of memory that will be used to pass the data. The producer task writes this data, the consumer task subsequently reads this data. Depending on the assignment of the tasks to the cores, the communicated data either goes via the L2 cache or the L3 cache. This is shown in Figure 6.9.

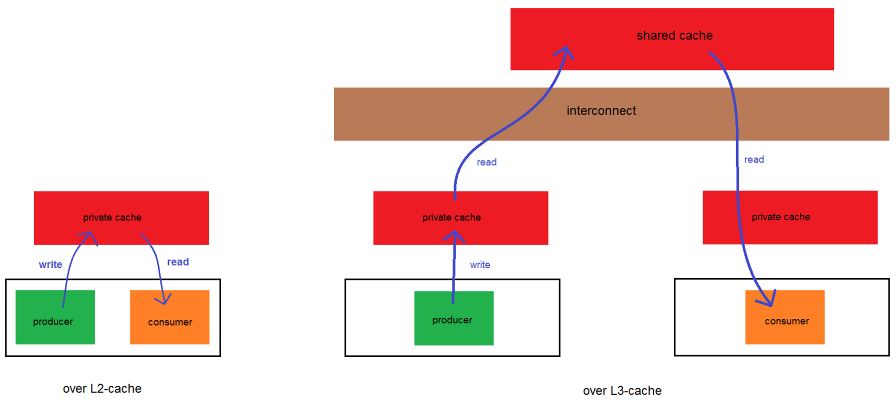


Figure 6.9: Setup for the measurement of the cache bandwidth.

The measurement procedure is as follows.

i. The background task allocates a block of data.

ii. It writes the data in order to force it to the cache.

iii. The background task and the sample task synchronize.

iv. The sample task reads the data in sequential order and measures the duration.

From this duration, the bandwidth is calculated as

$$\text{bandwidth} = \frac{\text{data size}}{\text{duration}}.$$

The results are shown in Figure 6.10. We can see that the inter-core bandwidth, i.e. the bandwidth for the L3 cache, is independent of the data size. This is what you would expect because no matter what the data size is, the data always has to follow the same path.

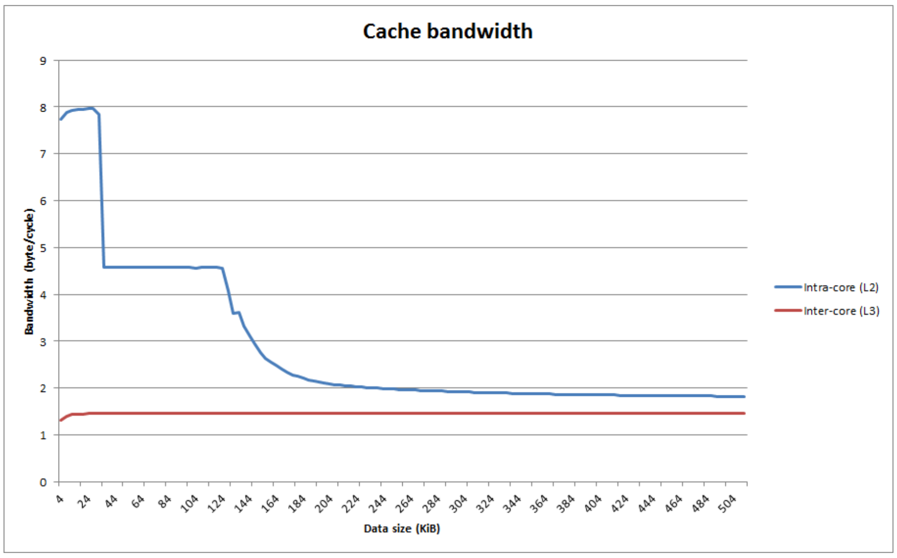


Figure 6.10: Measurement results for the cache bandwidth for data sizes of 8 Kib until 512 KiB. The results are given in bytes per cycle.

The course of the intra-core bandwidth contains the plateaus that you would expect. The first plateau (8 KiB until 32 KiB) is the bandwidth of the L1 cache. Then, from 40 KiB until 128 KiB, the bandwidth for the L2 cache is shown. From then on, the bandwidth decreases due to increasing fetches from L3 cache. It is expected that the line for the intra-core bandwidth will reach the inter-core bandwidth at 2 MiB as that is the size of the L3 cache.

From the results, we can deduce the following numbers.

$$\text{bandwidth}_{\text{L2}} = 4.575 \text{ byte/cycle}$$

$$\text{bandwidth}_{\text{L3}} = 1.457 \text{ byte/cycle}$$

We can see that the L2 bandwidth is comparable to the bandwidth measured with the *LMBench* benchmark (see Table 4.5) where we measured a bandwidth of 4.57 bytes per cycle.

6.3 Comparison of the two solutions

Using the results of the measurements, we can calculate the overhead in both solutions. We make the following assumptions:

- During the *load parameters* stage, 64 KiB need to be loaded. In reality, the amount of data to be loaded depends per execution but 64 KiB is the worst-case scenario that is specified by the application.
- The *set parameters* step takes 5 μs. This estimation was based on the timing behaviour of the application on the MPC8548 processor.

With these assumptions, the total overhead for both solutions is calculated as follows.

$$\text{solution 1 overhead} = 1066 + 222 + \frac{64 \times 1024}{4.575} + 222 + 222 = 16058 \text{ cycles}$$

$$\text{solution 2 overhead} = 1066 + 222 + \frac{64 \times 1024}{1.457} + 222 = 46483 \text{ cycles}$$

We can now compute the sample execution time of both solutions by replacing the parameters in the equations (6.1) and (6.2) by their values. We assume a clock frequency of 1500 MHz to convert the clock cycles to time values. There are still two unknown parameters in the equations:

number of cores The number of cores that will be used for a particular HPPC application mainly depends on how well the sample task can be parallelized.

background processing time for solution 1. This is also dependent on the way the sample task will be divided among the cores because only then it will be known what background tasks are necessary on each core.

Figure 6.11 shows the sample execution time of both solutions as given by equations (6.1) and (6.2) in function of the number of cores and the background processing time. In Table 6.1, the intersection points between the two solutions are shown. We can conclude that, if the background processing time is lower than 28 μs, then solution 1 gives the most time to execute the sample task, independently of the number of cores. As we have seen in Figure 6.1, the background processing time is currently 20 μs. When dividing the application over multiple cores, it is expected to get lower. Therefore, we can conclude that it is very probable that solution 1 is the better solution in terms of performance.

On an absolute level, assuming a background processing time of 20 μs, the performance of solution 1 is 64%. This means that 64% of the total available time on all cores can be spent on computations for the sample task. Under the same circumstances, this is between 32% (when using 1 core) and 56% (when using 8 cores) for solution 2.

Number of cores	Intersection for background processing (μs)
2	52
3	41
4	36
5	33
6	31
7	29
8	28

Table 6.1: Intersection points between solution 1 and solution 2 in Figure 6.11.

However, as was noted before, the 64 KiB data size for the parameters is an upper bound. In Figure 6.12, the sample execution time is plotted in function of the number of cores and the data size of the parameters while the background processing time was taken to be 20 μs. It shows that solution

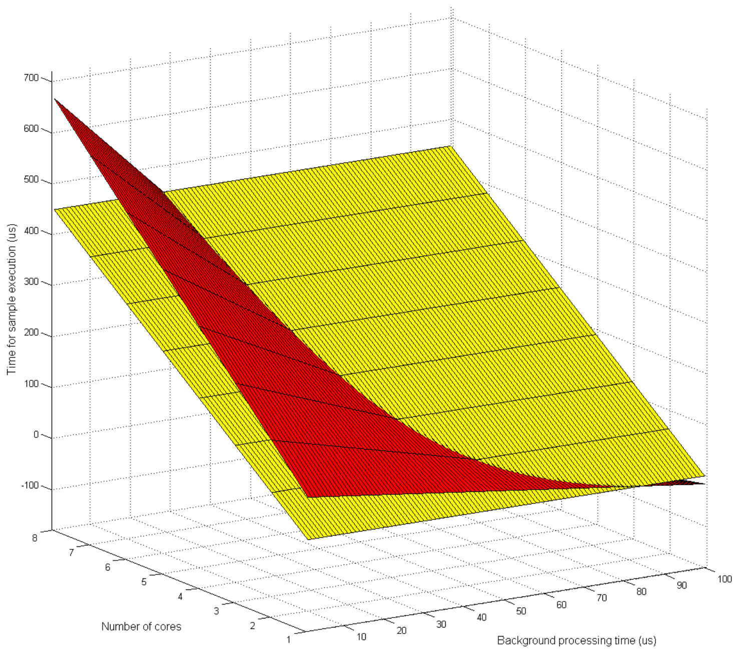


Figure 6.11: Sample execution time in function of the number of cores and the background processing time. The red surface is the sample execution time for solution 1; the yellow surface is the sample execution time for solution 2.

1 offers more performance for four cores or less, independently of the data size of the parameters. The second solution provides more time for sample processing when there are more than four cores and when the size of the parameters is less than 32 KiB.

As a conclusion, we can say that solution 1 offers more sample execution time for an execution scenario that is expected to be implemented: a background processing period of 20 μs and an upper bound of 64 KiB on the size of the parameters. However, the second solution offers two advantages that we did not take into account during the comparison:

- RPC calls will be serviced faster in solution 2. In solution 1, if an RPC call arrives just at the start of a new period of execution, it will have to wait for the execution of the sample task to be finished before it will be handled. In the worst-case, this delay can be as high as 70 μs. Servicing the RPC calls directly improves the throughput of the entire system a process on the host application is blocked while its RPC request is being serviced.

- In solution 1, during the *background processing* step, the background tasks pollute the cache for the sample task. Solution 2 does not have this disadvantage as they run in their own core. However, it is difficult to measure the impact that this has on the hit rate for the sample task. Therefore, we did not take it into account during the analysis. Furthermore, since the performance of the background tasks is less important than the performance of the sample

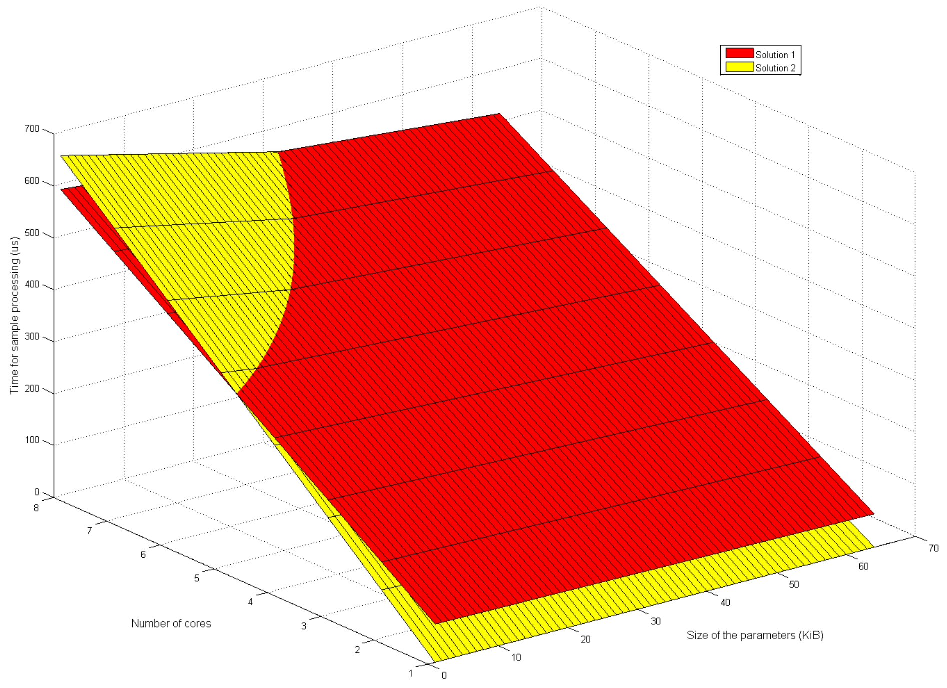


Figure 6.12: Sample execution time in function of the number of cores and the data size for the parameters (1 KiB to 64 KiB). The background processing time was fixed at 20 μs. The red surface is the sample execution time for solution 1; the yellow surface is the sample execution time for solution 2.

task, it is possible to allocate all the memory for the background tasks from a non-cacheable section of memory. This way, cache pollution by the background tasks is eliminated.

6.4 Scalability of the sample task

In the previous paragraphs, we have compared two solutions for mapping the HPPC application to the P4080 processor. In this paragraph, we will measure the scalability of the sample task as we divide it over multiple cores. We will use the model of the application that was introduced in Chapter 5 as our object of measurement.

As we know, the model of the HPPC application is a sequence of computational blocks. Code blocks operate on data blocks. Furthermore, the HPPC application provides opportunities for mainly data parallelism: when dividing the application over the cores, the total code size should stay approximately the same but the data blocks are spread out over the cores.

As we did in Chapter 5, we start with an application of 256 KiB of code and 256 KiB of data. On the P4080, this results into 3120 data blocks and 2516 code blocks. When we divide the application for 2 cores, each core will process half of the data, i.e. 1560 data blocks. This also means that each core will only access 1560 code blocks. In order to simulate the usage of the full code size, the

2516 code blocks are divided as shown in Figure 6.13. To process its data blocks, the first core uses the code blocks 1, 2, ..., 1560 while the second core reads the code blocks 957, 958,..., 2516. This method makes sure that the all code blocks are being accessed. The same method has been repeated for 3, 4, ..., 8 cores.

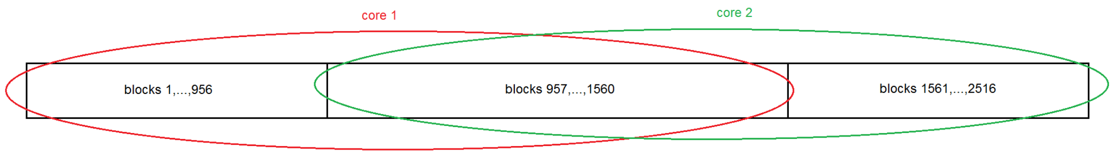


Figure 6.13: Division of the 2516 code blocks among 2 cores.

The performance was measured by running the application on all cores simultaneously. All cores ran the application for 1 second and we counted the number of blocks that were processed during this time interval. By using a barrier, all cores synchronized the start of their execution. In contrast with the previous measurements, this experiment was run in a Linux environment for practical reasons as the benchmark application uses the Portable Operating System Interface for Unix (POSIX) Threads [21] library. This library is not available in concordance with HPPC-OS. However, it is not expected that running the experiments in a Linux environment will have a big impact as the measurements are averaged over 1 second.

Figure 6.14 shows the results. The performance per core is relative to the performance when running the application on a single core, i.e. the following transformation was applied.

$$\text{result}_{\text{relative,n}} = 100\frac{\text{result}_{\text{absolute,n}}}{\text{result}_{\text{absolute,1}}}$$

with n being the number of cores. The plot also shows the division of the application (code and data) over the various cache levels. For that, the absolute usages were calculated first:

$$\begin{aligned}\text{absolute usage}_{\text{L1 data}} &= \min\left(\text{32 KiB}, \text{data size}\right)\\ \text{absolute usage}_{\text{L1 code}} &= \min\left(\text{32 KiB}, \text{code size}\right)\end{aligned}$$

$$\begin{aligned}\text{absolute usage}_{\text{L2}} = \min(&\text{128 KiB}, \text{data size} + \text{code size}\\ &- \text{absolute usage}_{\text{L1 data}}\\ &- \text{absolute usage}_{\text{L1 code}})\end{aligned}$$

$$\begin{aligned}\text{absolute usage}_{\text{L3}} = \max(&0, \text{data size} + \text{code size}\\ &- \text{absolute usage}_{\text{L1 data}}\\ &- \text{absolute usage}_{\text{L1 code}}\\ &- \text{absolute usage}_{\text{L2}})\end{aligned}$$

These were used to calculate the relative usages which are plotted in Figure 6.14.

$$\text{relative usage}_{\text{L1}} = 100\frac{\text{absolute usage}_{\text{L1 data}} + \text{absolute usage}_{\text{L1 code}}}{\text{data size} + \text{code size}}$$
$$\text{relative usage}_{\text{L2}} = 100\frac{\text{absolute usage}_{\text{L2}}}{\text{data size} + \text{code size}}$$
$$\text{relative usage}_{\text{L3}} = 100\frac{\text{absolute usage}_{\text{L3}}}{\text{data size} + \text{code size}}$$

The results of the performance indicate that the application scales more than linearly. The most important reason is that, because of the reduction in footprint, the accesses to higher cache levels decreases as the number of cores increases. Because of this behaviour, the application should be parallelised over as many cores as possible. Also, starting from 3 cores, the application on each core has become small enough to fit in the first two cache levels. This is an important threshold because from then on, the cores do not influence each other by the access to L3 cache.

It should be noted however that our analysis is based on the assumption that the application can be divided into equal parts, i.e. we assume an idealized situation. For that reason, the results should be taken as an upper bound on the maximal performance that can be achieved.

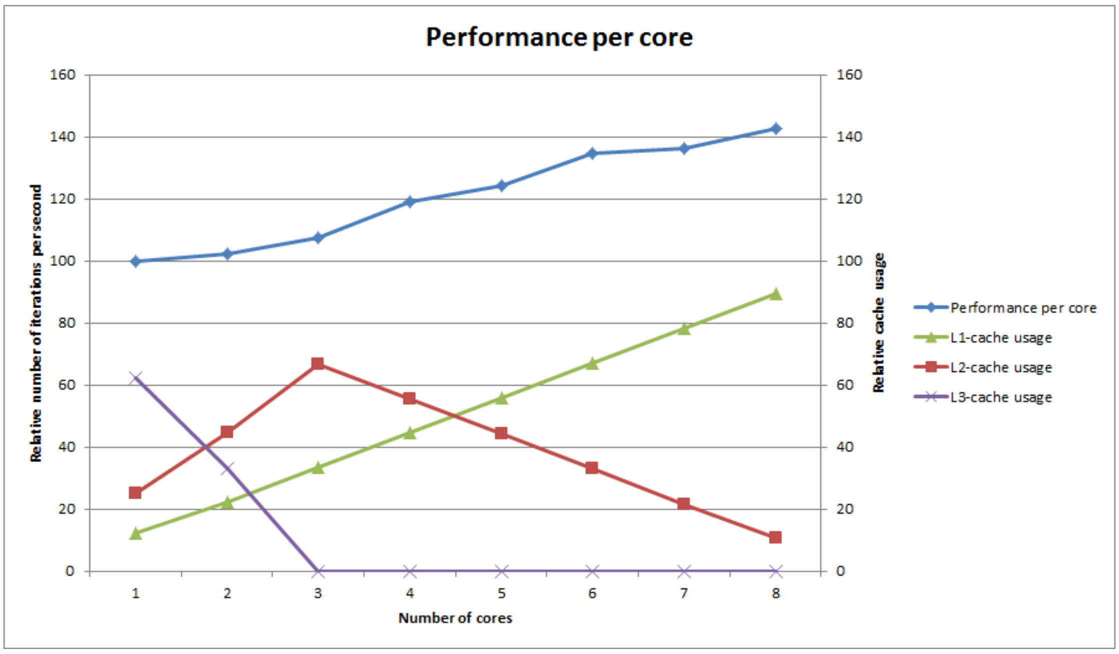


Figure 6.14: Results of the scalability of the sample task in function of the number of cores. The blue line is the performance per core relative to the single-core performance. The three other lines show the relative cache usage for each cache level. For any number of cores, the three lines add up to 100.

6.5 Synchronization on the P4080

The analyses performed in the previous paragraphs assumed that each part of the parallelised sample task ran independently of the others, i.e. that there is no communication between the parts executing on different cores. In reality, there will be locations in the application that all the cores need to reach before any of them can proceed with its execution. The location of these synchronization points is specific for each application. Also, it will depend on the way that the application is parallelised as a different division of the computational blocks making up the sample task among the cores will result in different synchronization points.

6.5.1 Barrier as a synchronization mechanism

Introduction

To provide this kind of synchronization, we introduce the well-known concept of a barrier. A barrier is a synchronization primitive that is used to synchronize different tasks. It indicates a point in the program where every task has to stop and is only allowed to continue when every other task has reached the barrier. Table 6.2 shows an example of how a barrier can be used.

task 1	task 2	...	task n
		...	
`shared BARRIER barrier;`	`shared BARRIER barrier;`		`shared BARRIER barrier;`
`/*`	`/*`		`/*`
`* Execution`	`* Execution`		`* Execution`
`*/`	`*/`		`*/`
`barrier_wait(&barrier);`	`barrier_wait(&barrier);`		`barrier_wait(&barrier);`
`/*`	`/*`		`/*`
`* This part will be executed`	`* This part will be executed`		`* This part will be executed`
`when the last task has`	`when the last task has`		`when the last task has`
`called barrier_wait.`	`called barrier_wait.`		`called barrier_wait.`
`*/`	`*/`		`*/`

Table 6.2: Example of the utilization of a barrier.

Implementation of a barrier on the P4080

We investigated two ways to implement barriers for inter-core synchronization.

counter Assume that N tasks need to synchronize at a barrier. A counter is initialized at 0. When a task arrives at the barrier, it increments the counter by 1. Then, it waits until the barrier counter reaches N. That condition indicates that all tasks have reached the barrier point and that they are allowed to continue their execution.

array of booleans On the P4080, we only have 8 cores. If we limit the usage of the barrier for inter-core synchronization, we can initialize a barrier as an array of 8 bytes. That way, we avoid the different cores to access the same memory location. Therefore, each core in the processor is mapped to a separate byte in the array. The byte of a participating core is initialized to 0. When a core reaches the barrier, it sets its byte in the barrier array to 1. It then loops over the bytes of all participating cores and waits until all are 1.

The advantage of the solution with the counter is that it is more generic: two tasks running on the same core can participate in the same barrier. This is not possible with the implementation by means of an array of booleans because each core is assigned only one byte. However, in the case of the HPPC application, it will never occur that two tasks on the same core will wait for the same barrier. Therefore, the comparison will be made solely based on the performance. The implementation for both methods is shown in Table 6.3.

counter	array of booleans

`/* Initially, counter is 0. */`
`int counter = 0;`
	`/* Initially, all booleans are false (zero-value)`
	`     . */`
	`char array_bool[N] = {0};`

`...`
`... execution`
`...`
`/* Increment has to happen atomically. */`
`atomic_inc(&counter)`
`/* Wait for the other tasks to have reached the`
`     barrier. */`
`while (counter < N);`
`/* Go on with execution. */`
`...`
`... execution`
`...`
	`...`
	`... execution`
	`...`
	`/* Each core writes a "1" to its flag. */`
	`array_bool[get_core_nr()] = 1;`
	`/* Wait for the other tasks to have reached the`
	`     barrier. */`
	`char ready = 0;`
	`while (!ready) {`
	`  ready = true;`
	`  int i;`
	`  for (i=0 ; i<N ; i++) {`
	`    if (!array_bool[i]) {`
	`      ready = false;`
	`      break;`
	`    }`
	`  }`
	`}`
	`/* Go on with execution. */`
	`...`
	`... execution`
	`...`

Table 6.3: Two ways of implementing a barrier.

Atomic operations on the P4080

In the implementation of a barrier with a counter, upon reaching the barrier, a task must perform a read-modify-write operation: it has to read the counter, increment it by 1 (modify) and then write the updated counter back to memory. This operation has to happen atomically. In particular, it is important that no other write to the counter is allowed during this sequence of operations. Table 6.4 illustrates why the increment has to happen atomically with an example.

core 1	core 2
`initially:  counter := 0`	
	`load r0, counter`
	`add r0, 1`
`load r0, counter`	
`add r0, 1`	`store r0, counter`
`store r0, counter`	

Table 6.4: Example that shows that the read-modify-write operation must happen atomically. Due to the interleaving of operations, the resulting value of the counter is 1 while it should be 2.

On the P4080, atomic operations can be implemented with load-link (*lwarx*) and store-conditional (*stwcx*) instructions. These are respectively special load and store instructions. When the *lwarx* instruction is issued, besides performing a normal load operation, it makes a reservation on the load address. The reservation is lost if between the execution of the *lwarx* and *stwcx* instructions, another core performs a write on the same address. The *stwcx* instruction first checks whether the reservation is still there and it only performs the store if this is the case. This mechanism makes sure that if the store succeeds, then it has happened atomically. Otherwise, we must retry until the store succeeds. In the example in Table 6.4, the store performed by core 1 would not have succeeded because the reservation would have been lost by the prior store by core 2. With these two instructions, we can realize a correct implementation of a barrier with a counter.

For the implementation with an array of booleans, atomic operations are not necessary. Upon reaching the barrier, a core writes the value "1" to the appropriate location, independently of the previous value. After that, while waiting, it only performs read operations on the memory locations corresponding to the other cores.

6.5.2 Overhead of a barrier

Introduction

Now we want to know the computational overhead introduced by synchronization with barriers. There are two sources of overhead:

updating the barrier This corresponds to the time to atomically increment the counter value with a barrier implemented as a counter. For an array of booleans, it is the time to write a specific value in the corresponding flag.

waiting for the barrier condition For the counter, this means reading it until it reaches the value N. For the array of booleans, the flags corresponding to all cores must be read.

Because the HPPC application has real-time constraints, it is interesting to have an upper-bound on the overhead. This is why we decided to measure the overhead for the worst-case scenario. This happens when all cores reach the barrier at exactly the same time; they try to update the barrier data structure simultaneously which means that there is maximal interference between the cores.

For the implementation with a counter, there is a possibility that one or more cores fall into an endless loop if the atomic increment always fails. However, this situation can never occur. The reason is that the atomic store operation, i.e. the *stwcx* instruction, only fails if another core has successfully performed a store on the same address. Therefore, that core will not compete with the other cores any more as it has completed its increment operation. In other words, the situation where two cores try to perform the "increment" operation and where both of them fail can not occur. Since there are a finite number of cores, it means that it will take only a finite number of "rounds" before all cores will have successfully performed the atomic increment operation. The theoretical upper bound on the number of tries before success then equals the number of cores.

Experimental setup

Figure 6.15 shows the setup of the experiment. One core is responsible for synchronizing the start of the measurement. This way, all N participating cores start updating the barrier data structure at the same time in order to trigger maximal interference between the cores.

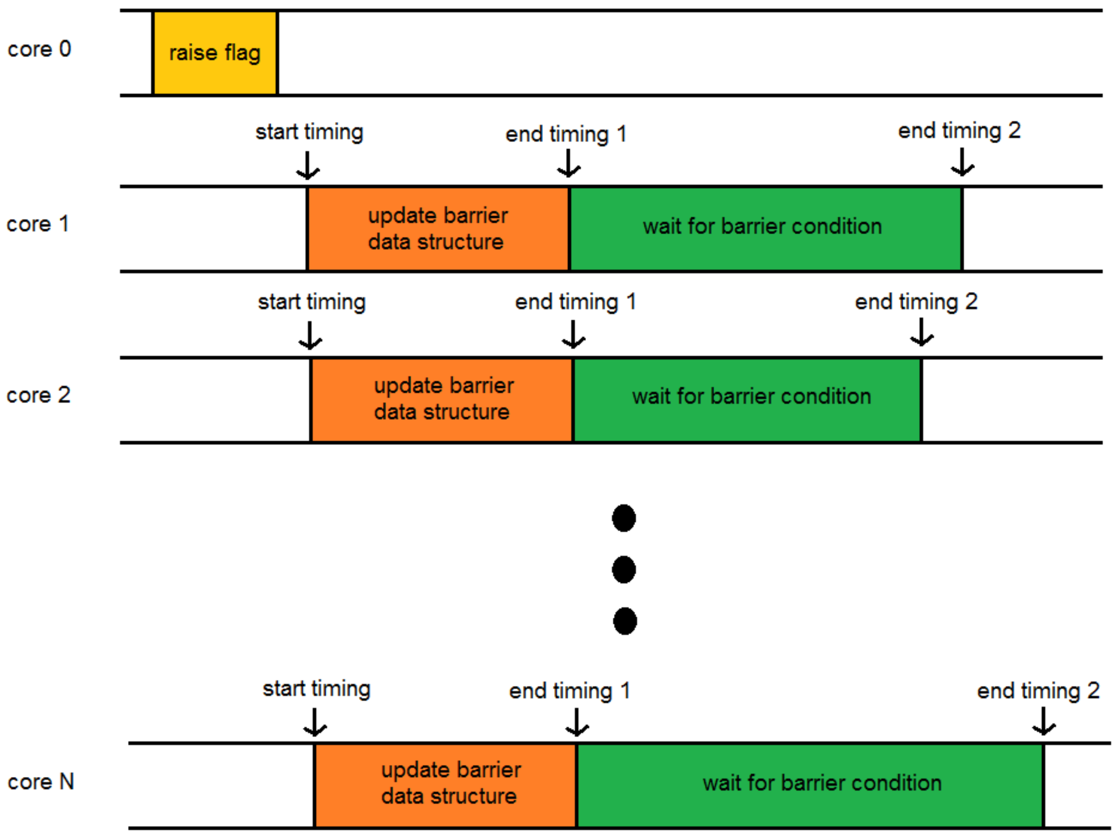


Figure 6.15: Experimental setup for the measurement of the overhead introduced by a barrier.

Counter

Figure 6.15 shows that all cores will finish updating the barrier data structure simultaneously. In practice, this is not the case because the increment operation on the counter is performed atomically; inherently, this means that updates will happen sequentially. This can be seen in Figure 6.16, where the experiment was run for a situation with 6 participating cores. The plot shows two lines: one that corresponds to the latency to update the barrier data structure (corresponding to "timing 1" in Figure 6.15) and one that corresponds to the total time ("timing 2" in Figure 6.15). The plot was constructed as follows. The experiment was run 1000 times. For each run, we recorded the time at which each core finishes the 'update' stage and the 'total' stage. We then sorted the results for both stages in ascending order according to their finishing time. The order in which the cores finish is not fixed; it varies from run to run. Each value in the plot corresponds to the median for that position. For instance, the left-most value of the 'update' stage corresponds to the median for the time that it takes before the first core has succeeded incrementing the counter.

We can see that the atomic increment operation causes sequential accesses as the 'update' line follows approximately a linear path.

To maintain cache coherency, the P4080 processor implements the MESI protocol. This is the most common protocol for write-back caches. In our case, it tries to keep the private cache, i.e. the combination of L1 and L2 cache, of each core consistent with the shared L3 cache. In the protocol, each cache line in the private cache of a core is marked with one of the four following states.

modified The cache line is present in the current cache only and it is dirty, i.e. it has been modified from the value stored in the L3 cache.

exclusive The cache line is present in the current cache only but it is clean, i.e. its value matches the value stored in the L3 cache.

shared The cache line may be present in other private caches as well and it is clean.

invalid The cache line is not valid.

When the last core has incremented the counter, i.e. it has just performed the last store, its own cache line corresponding to the *counter* variable will be in the Modified state. The copies of the *counter* variable in the other local caches are in the Invalid state. Then, it will succeed first in reading the updated *counter* variable since it has the variable in its local cache. When another core tries to read the barrier variable, the MESI protocol says that the following will happen.

- The requesting core issues a bus request to the L3 cache.
- The core that holds the cache line in the Modified state sees this and puts its value on the bus.
- The requesting core caches the value it received and puts it in the Shared state.
- The core that holds the cache line in the Modified state writes its value to the L3 cache and switches the state of its cache line to the Shared state as well.

The next core that issues the read request to the *counter* variable (the access to the bus is arbitrated with a queue) will see several caches with a copy in the Shared state. When the core puts the request on the bus, one of those caches will put its value on the bus. It then caches the copy it received and puts the cache line in the Shared state.

The course of the line corresponding to the total time can be explained as follows.

- The first one to finish is the core that was the last one to increment the counter. As mentioned before, it can directly read the value from its private cache.
- The second core to finish has to copy the value from the first core to finish. Also, that core has to write its value to the L3 cache. Both cores set their cache line to the Shared state.
- The cores that follow have to copy the counter from one of the caches that hold the cache line in the Shared state. Normally, the access to the bus is arbitrated by a queue; only one core can access the bus at the same time. Therefore, we would expect the accesses to be sequential, resulting in an ascending line in the plot. However, the line in the plot is constant. Therefore, we expect that the P4080 contains an optimization. Since the remaining cores have their copy of the counter in the Invalid state and they see a copy of that cache line on the bus, it is probable that they will use this copy on the bus to update their local copy. This can explain the flat section of the line corresponding to the core positions 3 to 6.

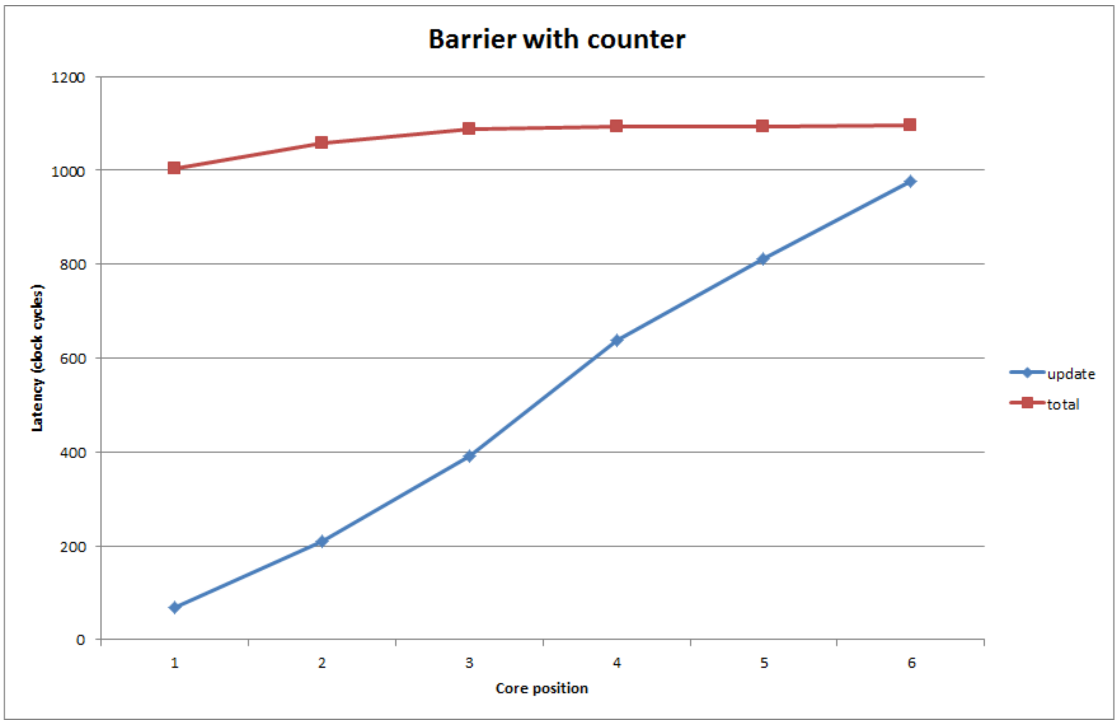


Figure 6.16: Results of the barrier overhead for the implementation with a counter.

Array of booleans

With the implementation of a barrier as an array of booleans, each core writes a 1 in its corresponding byte as a means of updating the barrier data structure. Therefore, each core writes to a different memory address. However, since the cache line size on the P4080 is 64 bytes, they all map to the same cache line.

Just before the start of the experiment, the synchronizing core, i.e. core 0 in Figure 6.15, initializes the array with 0 on each position. The effect of this write operation invalidates the cache line on every other core. When a core then performs a write, the MESI protocol prescribes the following steps to be taken.

- The core issues a bus request for a write.
- The synchronizing core sees this and blocks the request. It takes control of the bus, writes back its copy to the L3 cache and sets its copy state to Invalid.
- The core re-issues the bus request.
- It reads the value from the L3 cache, updates its local copy and sets its copy state to the Modified state.

As mentioned before, the access to the bus is arbitrated by a queue so the cores will perform their writes sequentially. When a core has finished its write, it will poll the other cores bytes in the array. The last core to write its byte will have an advantage as it will hold the most up-to-date copy of the array in its private cache. The others will have to update their local copy in the same way as for the *counter* variable. Therefore, it is expected that the total time will behave in the same way as for the counter barrier.

Figure 6.17 shows the measurement results in the same way as for the counter barrier. It matches the expected behaviour except for one remark. We expected a straight line for the update latency. However, the latencies for the first two cores are approximately equal. We can not explain the phenomenon.

Comparison

In the previous paragraphs, we have analysed the behaviour of the two implementations for a barrier separately. Now we are going to compare their performances. For that purpose, we only take into account the maximum total latency. This is the latency measured by the last core to exit the waiting section. In Figures 6.16 and 6.17, this coincides to the most-right point of the line corresponding to the total latency.

To gain insight in the scalability of the methods, we varied the number of participating cores from 1 to 7 cores. Furthermore, as we did with the previous measurements, we ran each experiment 1000 times in order to increase the statistical significance of our results.

Figures 6.18a and 6.18b respectively show the median and the maximum total latency of both implementations in function of the number of participating cores. The plots show that both implementations scale approximately linearly. The reason for their deviation from a straight line is not

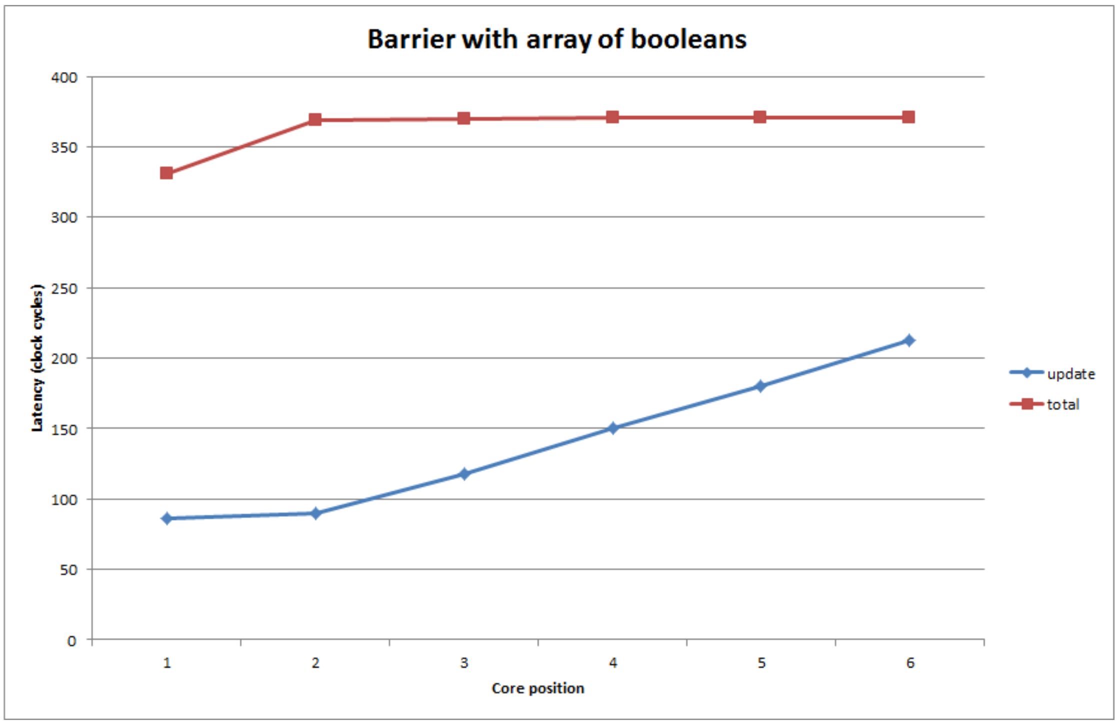


Figure 6.17: Results of the barrier overhead for the implementation with an array of booleans.

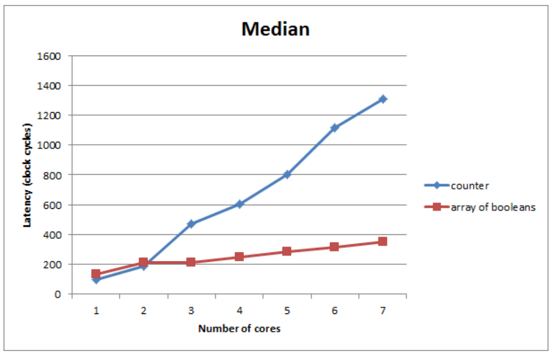
known; a possible source is the topology of the interconnect between the cores. However, there is no documentation available about the specifics of the network connecting the cores.

The results also show that the implementation of a barrier with an array of booleans is less costly in terms of overhead. The implementation with a counter introduces less overhead only in the case that there is solely one core participating in the barrier. However, this case will never be used in practice as the use of a barrier can then be avoided.

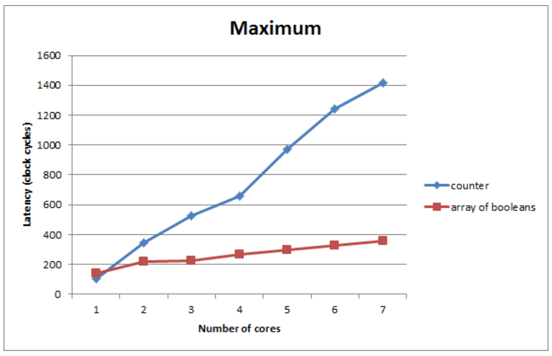
Figure 6.19 shows the same results as Figure 6.18b to the period of execution defined by the doorbell frequency (10 KHz) and assuming a clock frequency of 1500 MHz. It shows that the overhead for the barrier with an array of booleans attains a maximum of 0.30 % of the period of execution. For the implementation with a counter, this is 1.18 % which is about 4 times higher.

Conclusion

We analysed two ways of implementing a barrier: by using a counter or by using an array of bytes. The advantage of the implementation with a counter is that it can be used by more than one task on the same core. However, such a situation will never occur in the HPPC application. Therefore, the only comparison to be made is based on the performance. There, the array of booleans has less overhead than the implementation with a counter variable. In the worst-case scenario, the latter causes about 4 times more overhead. When 7 cores are participating in the barrier, the solution with a counter has an overhead of about 0.95% of a period of execution; this is 0.24% with an array of



(a) Median



(b) Maximum

Figure 6.18: Comparison of the performance of a barrier implemented by means of a counter or by means of an array of booleans in function of the number of cores. The plots are the result of 1000 experiments.

booleans. Therefore, the array of booleans is the preferred implementation of a barrier for the HPPC application.

6.5.3 Busy-waiting and interrupt-based waiting

When a task arrives at a barrier, it has to wait until the last task reaches the barrier. Waiting can be implemented in two ways.

busy-waiting The task repeatedly tests the barrier condition in a loop. This was the method used in the experiments described in Section 6.5.2.

interrupt-based waiting Upon reaching the barrier, the task puts itself in a halting state, i.e. it stops executing instructions. The last task to arrive at the barrier has then the duty to "wake-up" the waiting tasks. This can be done through an IPI, which was introduced in Section 6.2.3.

The main advantage of interrupt-based waiting is that the utilization of the core increases as no clock cycles are wasted to continuous polling. While the task is blocked, another task can be scheduled to run. Also, even if no other task is running, the core can be set in a "sleeping" mode which reduces its power consumption.

However, the main disadvantage of interrupt-based waiting is the increased response latency of a task after the condition to pursue execution has become true. The additional causes of latency are the following:

interrupt routing latency The time for the last task to dispatch the interrupt and for the interrupt to arrive at the destination core.

interrupt service routine latency The time it takes to execute the interrupt service routine.

context switching latency (optional) If another task is chosen to execute during the waiting time, an additional context switch has to occur.

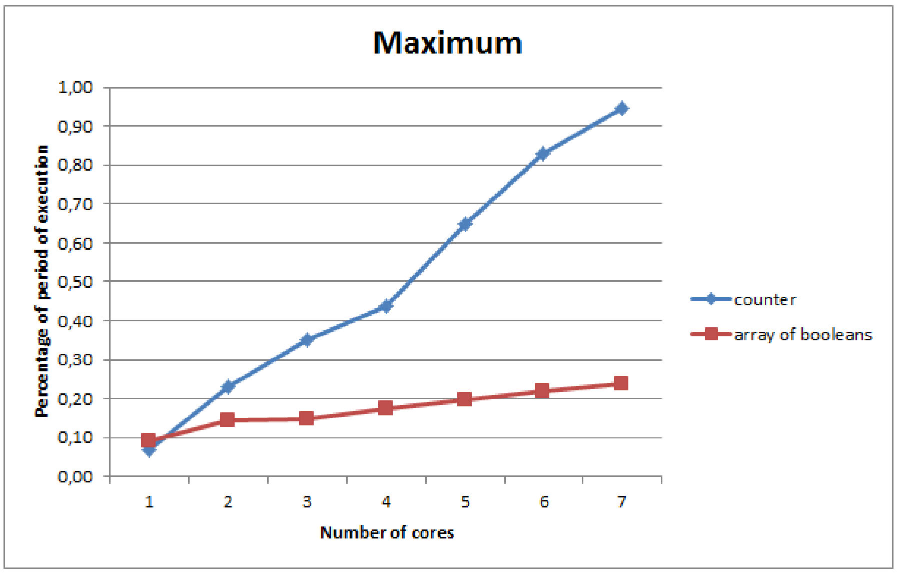


Figure 6.19: Percentage of the maximum overhead as shown in Figure 6.18b relative to the period of execution defined by the doorbell frequency (10 KHz) and assuming a clock frequency of 1500 MHz.

In Section 6.2.3, we measured the combination of the interrupt routing latency and the latency of the execution of the interrupt service routine. Figure 6.7 showed that the average latency of an IPI is 1066 clock cycles, based on 1000 experiments. The context switching latency was measured in Section 6.2.4 and it was found that its average was 222 cycles.

With these results, Figure 6.20 compares the three waiting mechanisms for a barrier: busy-waiting, interrupt-based waiting and interrupt-based waiting with a context switch. It shows the percentage of the maximum overhead relative to the period of execution defined by the doorbell frequency (10 KHz) and assuming a clock frequency of 1500 MHz. The barrier implementation of an array of booleans was used. From the plot we can conclude that interrupt-based waiting increases the barrier overhead by a factor of 4. Performing a context switch further increases the overhead.

However, the model assumes that the interrupt latency is independent of the number of cores. In Section 6.2.3, the interrupt latency was measured for a one-to-one situation, i.e. only one core that received the interrupt. We expect the latency to increase in a one-to-many situation, where there are multiple cores receiving the interrupt simultaneously. Therefore, for a higher number of cores, the latency for interrupt-based waiting is likely to be worse than is shown in the plot.

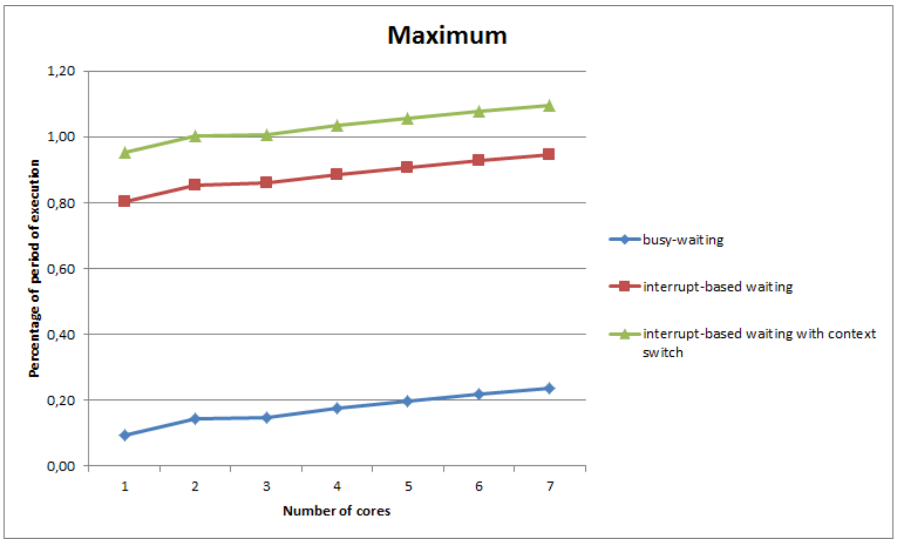


Figure 6.20: Percentage of the maximum overhead relative to the period of execution defined by the doorbell frequency (10 KHz) and assuming a clock frequency of 1500 MHz for three different waiting mechanisms.

6.6 Conclusion

This chapter has been a first investigation in porting the HPPC application to a multicore platform. Two possible mappings of tasks to cores have been compared. The scalability of the application has been measured and it has helped to identify its behaviour when dividing it among several cores. Then, two ways of implementing synchronization mechanisms were investigated and their cost has been measured.

When comparing the two solutions, we came to the conclusion that solution 1 provides more sample execution time when taking into account a realistic execution scenario. However, it should be noted that the comparison did not take into account the cache pollution introduced by the background tasks for solution 1. Furthermore, the advantages of a lower RPC servicing delay provided by solution 2 have not been included in the comparison. The effects of these should be analysed in future work.

After the analysis of the scalability of the sample task, we came to the conclusion that the application scales more than linearly. The main reason is that the memory footprint reduces when dividing the application and therefore, the code and data can be stored in lower cache levels. However, we assumed a simplistic model: the application is divided into perfectly equal parts. Therefore, the resulting performance measurements should be taken as an upper bound on the performance that can be achieved in reality.

Next, we investigated how synchronization with barriers can be implemented on the P4080 and we analysed its overhead. In particular, we compared the worst-case overhead of two different implementations: one that uses a counter and one that uses an array of booleans. The conclusion is that:

- the overhead of both implementations scales linearly with the number of cores.

- the implementation with an array of booleans introduces approximately four times less overhead.
- the maximum response latency of interrupt-based waiting compared to busy-waiting varies from about 4 times slower (7 participating cores) to 6 times slower (2 participating cores). These factors further increase when adding the cost of a context switch.

6.7 Future work

In overall, the analysis shows that the HPPC application has the potential to be parallelized in an efficient way. The comparison of the two mapping alternatives revealed a good way to divide the tasks over the cores. The next important step is to analyse how the computation networks of each sample task can be divided in an optimal way. This is application-specific and was therefore not in the scope of this thesis. The analysis of the synchronization mechanisms in Section 6.5 and the various measurements performed in Section 6.2 can be useful information for this investigation.

During the study, we only looked at a single HPPC application running on a multi-core platform. However, the system could benefit from running the host and the HPPC application on the same multi-core processor. Indeed, there is communication between the two systems in the form of RPC calls over an Ethernet bus. Therefore, if both the host application and a HPPC application co-exist on the same processor:

- Communication between the host and the HPPC will be faster as inter-core communication is faster than communication over an Ethernet bus.
- It can reduce the number of modules needed to implement the whole system as two separate modules can be merged into one.

Due to the properties of the applications, it is not practical to run the host and a HPPC application on the same operating system. A type 1 hypervisor is a layer of software that lies between the hardware and the operating system. It enables the co-existence of two different operating systems by offering a virtual CPU to each guest OS. Therefore, it can provide a solution to the execution of both the host and the HPPC application on the same processor. However, it introduces problems such as the management of shared resources, performance interference between the guest operating systems and additional overhead. Future work should reveal the advantages and drawbacks on the usage of a hypervisor.

Chapter 7

Conclusion

7.1 Comparison of processors

In the first part of the thesis, four different processors were benchmarked for their performance on two important applications in the system under consideration: the host application and the HPPC application. The choice of the processors was based on three criteria:

i. The roadmap of the company that owns the system. Currently, the system uses PowerPC processors. However, to support their decisions on the long-term roadmap, the company is interested in how well Intel processors perform on their applications compared to PowerPC processors. Therefore, processors from both Intel and PowerPC were included in the study.

ii. We wished to compare processors in a wide spectrum of the power-performance range. Therefore, we benchmarked both a low-power and low-performance processor (Intel Atom Z530) as well as a high-power and high-performance processor (Intel Xeon E5540). Furthermore, two PowerPC processors represent the middle of the range.

iii. Practical availability of the processors.

Because the exact computational behaviour of the host application is not known, we chose to investigate three general aspects of the performance of an architecture: the performance of the memory hierarchy, the integer performance and the floating-point performance. For the HPPC application, a model that simulates its behaviour was provided and this model was used as a benchmark to compare the processors. For both application, the investigation revealed the same conclusions:

- When considering performance, the Intel Nehalem architecture, implemented by the Intel Xeon E5540, achieves the best results. However, due to its high power consumption, it will be necessary to step away from the AdvancedTCA racks to store the processor boards.
- If there is a desire to stay in the current environment, i.e. a configuration where the processor boards are stored as AMC modules in an AdvancedTCA rack, then the PowerPC P4080 is the preferred platform. Its power consumption stays within the bounds set by the AdvancedTCA specification.

- Intel processors have two non-performance related disadvantages compared to the PowerPC processors. Freescale Semiconductor, which is the designer and producer of the PowerPC processors, offers a duration of supply of at least ten years while Intel offers only five years. Depending on the life time of the system, this factor can be determinant in the decision for a processor.
- The documentation provided by Freescale Semiconductor for their processors is much more extensive than the documentation provided by Intel. Documentation is important for the software engineers to achieve the highest performance for their applications. This is especially crucial for the HPPC application because of its real-time behaviour and its high demand in performance.

7.2 The HPPC application on a multi-core platform

In the second part, we investigated into ways of porting the HPPC application to a multi-core platform. We assigned the PowerPC P4080 as our target platform and this processor was used to perform the measurements. First, we analysed two alternatives for the mapping of the two types of tasks to the cores: one where the background tasks run on the same core as the sample task and one where a separate core is dedicated to the execution of the background tasks. The performance was defined as the time that is available for the execution of the sample task. The results indicated that under a realistic scenario of execution, the first alternative offers more performance.

After this, we investigated the scalability of the sample task as we divide it over the available cores. We used the model of the application as our benchmark. The measurements were made under the assumption that the application is ideally parallelizable, i.e. that it can be split up in perfectly equal parts. The results show that the application scales more than linearly. The reason is that the memory footprint reduces as the application is split up in smaller pieces. Therefore, the number of accesses to higher levels of the memory hierarchy is reduced.

The last part was dedicated to synchronization between parts of the application executing on different cores. Synchronization is implemented with barriers. Two different implementations were proposed:

- A barrier as a counter.
- A barrier as an array of booleans.

We then measured the worst-case overhead introduced by the two methods and made a comparison. The results of the comparison show that:

- the overhead of both implementations scales linearly with the number of cores.
- the implementation with an array of booleans introduces approximately four times less overhead than the implementation with a counter.

In overall, the analysis shows that the HPPC application has the potential to be parallelized in an efficient way. Also, the work done for the measurements of the interrupt latency, the context switching latency and the synchronization overheads can help finding an optimal solution for the parallelization of the computational graph that composes the sample task.

Chapter 8

Future work

The host application lacks a clear profile of its computational behaviour. During the thesis, it was not possible to use the real application as a benchmark because the environment in which it runs is complex and not easily reproducible. However, to validate the conclusion of our performance measurements, we would have to run the real application on the various processors. An alternative would be to profile the application and develop a model based on the profiling results that would mimic its computational behaviour, as was done for the HPPC application. A model of an application has the advantage of being simple to execute yet it is representative of the real application.

In the benchmarks for the HPPC application, we have focused only on the computational performance. We did not take into account the real-time requirements of the application. In particular, previous studies performed by the owner of the system revealed that the interrupt latency of the platform, i.e. the time between the arrival of doorbell signal indicating the start of a new period of execution and the execution of the sample task, is significant for the performance. There must be a guarantee on an upper bound that is sufficiently low. The interrupt latency is very dependent on the architecture and therefore future work should perform measurements of this latency on the processors that were discussed.

Porting an application from a single-core platform to a multi-core platform is complex and the studies performed in this thesis do not cover all aspects. Future work emanating from this problem comprises:

- The analysis of how the computation networks of each sample task can be divided in an optimal way. This is application-specific and was therefore not in the scope of this thesis. The analysis of the overhead introduced by different implementations of synchronization mechanisms can be useful information for this investigation.

- To reduce the number of processor boards in the system, the host application and a HPPC application can co-exist on the same multi-core processor using a hypervisor. This introduces various problems such as the management of shared resources and additional overhead. Future study should reveal the advantages and drawbacks on the usage of a hypervisor.

Glossary

arithmetic intensity ratio of the floating-point operations and the amount of data transferred.

background task task that runs as part of the HPPC application and that serves as interface between the sample task and the host application.

bare-board software application that runs without an operating system or supported only by a minimal library.

hit rate percentage of accesses that result in cache hits.

host component of the system that manages and configures the SRIO network and the other components.

HPPC component of the system that implements the control loop.

MESI protocol Cache coherency and memory coherence protocol. It gets its name from the four states in which a cache line can be: Modified, Exclusive, Shared or Invalid.

QHA component of the system that acts as an interface between the system and the outside world, i.e. the sensor and the actuator boards.

sample task periodic task that implements the control loop of the HPPC application.

SMA component of the system that synchronizes the system.

Acronyms

AdvancedTCA	Advanced Telecommunications Computing Architecture.
AMC	Advanced Mezzanine Card.
CISC	Complex Instruction Set Computing.
CMOS	Complementary Metal-Oxide-Semiconductor.
CPI	Cycles Per Instruction.
CPU	Central Processing Unit.
DRAM	Dynamic Random-Access Memory.
DSP	Digital Signal Processor.
EOI	End-Of-Interrupt.
FPGA	Field-Programmable Gate Array.
GPU	Graphical Processing Unit.
HPPC	High-Performance Process Controller.
IDEA	International Data Encryption Algorithm.
ILP	instruction-level parallelism.
IPI	Interprocessor Interrupt.
ISA	Instruction Set Architecture.
LAPACK	Linear Algebra PACKage.
MPIC	Multicore Programmable Interrupt Controller.

POSIX Portable Operating System Interface for Unix.

PPE Power Processing Element.

QHA Quad High Speed Serial Link AMC.

RISC Reduced Instruction Set Computing.

RPC Remote Procedure Call.

SIMD Single Instruction, Multiple Data.

SMA Single MoSync AMC.

SMP Symmetric Multiprocessing.

SMT Simultaneous Multithreading.

SoC System-On-a-Chip.

SPE Synergistic Processing Element.

SRAM Static Random-Access Memory.

SRIO Serial RapidIO.

SSE Streaming SIMD Extensions.

TDP Thermal Design Power.

TLB Translation Lookaside Buffer.

Bibliography

[1] Freescale Semiconductor, "MPC8548E PowerQUICC™III Integrated Processor Family Reference Manual."

[2] M. J. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, pp. 948 –960, sept. 1972.

[3] `http://www.realworldtech.com/page.cfm?ArticleID=RWT122600000000`.

[4] `http://en.wikipedia.org/wiki/Symmetric_multiprocessing`.

[5] Intel, "Intel®64 and IA-32 Architectures Optimization Reference Manual."

[6] Freescale Semiconductor, "PowerPC™e500 Core Family Reference Manual."

[7] Freescale Semiconductor, "P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual."

[8] Freescale Semiconductor, "e500mc Core Reference Manual."

[9] Freescale Semiconductor, "P5020 QorIQ Integrated Multicore Communication Processor Family Reference Manual."

[10] Freescale Semiconductor, "e5500 Core Reference Manual."

[11] M. E. Thomadakis, "The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms," tech. rep., Supercomputing Facility, Texas A&M University.

[12] `http://www.blachford.info/computer/Cell/Cell1_v2.html`.

[13] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Commun. ACM*, vol. 27, pp. 1013–1030, October 1984.

[14] H. J. Curnow, B. A. Wichmann, and T. Si, "A synthetic benchmark," *The Computer Journal*, vol. 19, pp. 43–49, 1976.

[15] `http://www.netlib.org/lapack/`.

[16] `http://www.bitmover.com/lmbench/`.

[17] `http://www.tux.org/~mayer/linux/bmark.html`.

[18] Torbjörn Granlund, "Instruction latencies and throughput for AMD and Intel x86 processors," tech. rep., Kungliga Tekniska Högskolan.

[19] A. Fog, "Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," tech. rep., Copenhagen University, College of Engineering.

[20] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[21] `http://pubs.opengroup.org/onlinepubs/9699919799/`.