Performance monitoring on high-end general processing boards using hardware performance counters

GABOR ANDAI
Performance monitoring on high-end general processing boards using hardware performance counters

Gabor Andai

2015-03-15

Master’s Thesis

Examiner
Mats Brorsson

Advisers
Bo Karlsson, Robert Thorhuus, Ericsson AB
KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)
Department of Communication Systems
SE-100 44 Stockholm, Sweden
Abstract

Most of the advanced microprocessors today incorporate on-chip hardware performance counters. These counters are capable to count various events in a non-invasive way, while executing real workloads. Events such as the number of instructions, memory accesses, cache and TLB misses are the most common ones that can be precisely measured.

The primary accomplishment of this work was to implement a performance monitoring tool, which could be used to evaluate system behaviour on high-end processing platforms. The tool is able to collect data from hardware performance counters and present them in an interpretable way. Moreover, it has support for two different platforms and two operating systems. As a secondary objective, several measurements were carried out on both supported platforms and operating systems to demonstrate the tool’s capabilities, and to solve the potential use-cases.
Sammanfattning

De flesta av dagens mikroprocessorer innehåller prestandaräknare direkt i hårvaran. Dessa räknare kan räkna olika typer av händelser på ett icke störande sätt medans hårvaran är under last. Händelser såsom instruktioner, minnesaccesser, cache och TLB missar är de vanligast förekommande räknarna som kan göras precisa.

Acknowledgements

Special thanks for my supervisors, Marcus Jägemar and Mats Broxsson for their support and guidance. I would also like to thank to Bo Karlsson, Robert Thorhuus and all the other Ericsson and Freescale employees, who helped me in any way. Special thanks for Ericsson for all the opportunities that I got.
## Contents

1 Introduction
   1.1 Background .......................... 1
   1.2 Problem Description .................. 1
   1.3 Proposed solution ..................... 2
   1.4 Purpose .............................. 3
   1.5 Goals ................................ 3
   1.6 Delimitations ......................... 4
   1.7 Structure of this Thesis .............. 4
   1.8 Methodology .......................... 5

2 Hardware overview ..................... 6
   2.1 Freescale PowerPC P4080 ............... 6
       2.1.1 Caches .......................... 6
       2.1.2 Memory management unit ............ 7
   2.2 Freescale PowerPC T4240 ............... 7
       2.2.1 Caches .......................... 8
       2.2.2 Memory management unit ............ 8

3 Performance monitoring .............. 10
   3.1 Overview ............................ 10
       3.1.1 Monitoring methods ................. 10
       3.1.2 Sampling methods .................... 11
       3.1.3 Elected method ....................... 11
   3.2 Hardware performance counters ........ 11
       3.2.1 Performance counters in P4080 and T4240 .... 12
   3.3 Selected metrics for monitoring ....... 13
       3.3.1 Cycles/Instruction ratio ............... 14
       3.3.2 L1 Instruction cache hit-rate and ratio .... 14
       3.3.3 L1 Data cache hit-rate and ratio ........ 14
       3.3.4 L2 Instruction and Data cache hit-rate and ratio .... 15
       3.3.5 Interrupt/Cycle ...................... 15
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.6</td>
<td>L1 Instruction TLB miss-rate and ratio</td>
<td>15</td>
</tr>
<tr>
<td>3.3.7</td>
<td>L1 Data TLB miss-rate and ratio</td>
<td>16</td>
</tr>
<tr>
<td>3.3.8</td>
<td>L2 TLB miss rate</td>
<td>16</td>
</tr>
<tr>
<td>3.3.9</td>
<td>Branch Target Buffer hit-rate and ratio</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Perf - software interface</td>
<td>17</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Capabilities</td>
<td>17</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Interface</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Implementation</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Requirements</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Overview</td>
<td>19</td>
</tr>
<tr>
<td>4.3</td>
<td>Charmon - characteristic monitor tool</td>
<td>20</td>
</tr>
<tr>
<td>4.4</td>
<td>Ping-pong</td>
<td>23</td>
</tr>
<tr>
<td>4.5</td>
<td>Loadsim - load simulator tool</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Measurements</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparing between an RTOS and Linux on P4080</td>
<td>25</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Workload</td>
<td>26</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Test cycle</td>
<td>26</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Observations</td>
<td>27</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparing P4080 and T4240</td>
<td>29</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Workload</td>
<td>29</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Test cycle</td>
<td>30</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Observations</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>Related and future work</td>
<td>33</td>
</tr>
<tr>
<td>6.1</td>
<td>Related work</td>
<td>33</td>
</tr>
<tr>
<td>6.2</td>
<td>Future work</td>
<td>34</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Charmon</td>
<td>34</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Workload applications</td>
<td>34</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>36</td>
</tr>
<tr>
<td>A</td>
<td>Test results of comparing RTOS and Linux</td>
<td>39</td>
</tr>
<tr>
<td>A.1</td>
<td>Data plots - comparing RTOS and Linux</td>
<td>39</td>
</tr>
<tr>
<td>B</td>
<td>Test results of comparing P4080 and T4240</td>
<td>47</td>
</tr>
<tr>
<td>B.1</td>
<td>Dhrystone results</td>
<td>47</td>
</tr>
<tr>
<td>B.1.1</td>
<td>P4080</td>
<td>47</td>
</tr>
<tr>
<td>B.1.2</td>
<td>T4240</td>
<td>47</td>
</tr>
</tbody>
</table>
B.2 Data plots - comparing P4080 and T4240 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
List of Figures

2.1 P4080 block diagram ........................................ 7
2.2 T4240 block diagram ........................................ 8
3.1 PMC block diagram ........................................ 12
4.1 Software overview ........................................ 20
4.2 Charmon sequence diagram ................................ 21
4.3 Sample plot ........................................ 22
4.4 Ping-pong diagram ........................................ 24
5.1 Test cycle for comparing operating systems .............. 27
5.2 Test cycle for comparing P4080 and T4240 ............... 31
A.1 Signal turnaround time ..................................... 40
A.2 L1 Instruction Cache - Hit ratio and Instruction accesses .... 40
A.3 L1 Instruction TLB reloads ................................ 41
A.4 L1 Data Cache - Hit ratio and number of cache reloads ........ 42
A.5 L1 Data TLB reloads ..................................... 43
A.6 Number of interrupts ..................................... 43
A.7 L2 Instruction Cache - Hit ratio and instruction accesses .... 44
A.8 L2 Data Cache - Hit ratio and data accesses ............. 45
A.9 L2 TLB reloads ........................................ 45
A.10 Cycles/Instruction ....................................... 46
A.11 Branch Target Buffer - Hit ratio and hit rate ............ 46
B.1 L1 Instruction Cache - Hit ratio and Instruction accesses .... 48
B.2 L1 Instruction 4-Kbyte TLB reloads ...................... 48
B.3 L1 Data Cache - Hit ratio and number of cache reloads .... 49
B.4 L1 Data 4-Kbyte TLB reloads ............................. 49
B.5 Number of interrupts ..................................... 50
B.6 L2 Instruction Cache - Hit ratio and instruction accesses .... 50
B.7 L2 Data Cache - Hit ratio and data accesses ............. 51
B.8 L2 TLB reloads ........................................ 51
List of Figures

B.9 Cycles/Instruction .................................................. 52
B.10 Branch Target Buffer - Hit ratio and hit rate ................ 52
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTB</td>
<td>Branch Target Buffer</td>
</tr>
<tr>
<td>CPI</td>
<td>Cycles per Instruction</td>
</tr>
<tr>
<td>FSL</td>
<td>Freescale Semiconductor</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating Point Unit</td>
</tr>
<tr>
<td>HPC</td>
<td>Hardware Performance Counter</td>
</tr>
<tr>
<td>IPC</td>
<td>Instructions per Cycle</td>
</tr>
<tr>
<td>L1</td>
<td>Level 1</td>
</tr>
<tr>
<td>L2</td>
<td>Level 2</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PMC</td>
<td>Performance Monitor Counter</td>
</tr>
<tr>
<td>PMU</td>
<td>Performance Monitor Unit</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating system</td>
</tr>
<tr>
<td>SOC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>TLB</td>
<td>Translation Lookaside Buffer</td>
</tr>
<tr>
<td>VSP</td>
<td>Variable Size Pages</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

"State-of-the-art high performance microprocessors comprise tens of millions of transistors and operate at frequencies up to 2GHz. These processors execute many tasks at a time, employ significant amounts of speculation, out-of-order execution and other micro-architectural techniques." [3] To be able to perform in-depth analysis and debugging, these processors often incorporate on-chip hardware performance counters. These counters can be used to precisely measure various events and performance metrics in a non-invasive way, while executing real workload.

1.2 Problem Description

The following four problems were identified during software and hardware development of high-end processing boards:

1. **Investigate performance impact of switching to another operating system**
   Switching to a new operating system, is a challenge in itself. Before doing that, it is very useful to know what performance impact could be expected. Using a monitoring tool and target specific benchmarks to run tests on both old and new operating systems can help to better understand the performance impact.

2. **Investigate performance impact of switching to different platform**
   General and objective evaluation of processors is beyond complicated. When it comes to switching from one platform to another relying on results
from general benchmarks can be misleading. Simply, it is hard to forecast how a proprietary applications will perform on a new platform. By using a monitoring tool and target specific workload, both which run on the old and new platform, it is possible to objectively evaluate the performance impact of the new platform.

3. **Memory contention on multi-core systems:**

   Multi-core processors are using common shared memory among cores. Evidently, this leads to memory contention, "which is one of the largest sources of inter-core interference in statically partitioned multicore systems, and the contention reduces the overall performance of applications and causes unpredictable execution-times" [4]. In present-day processors, it is hard to protect one core from another to steal each other’s memory bandwidth. Hardware vendors are taking the first steps introducing quality of service in core interconnect systems, but solutions are far from mature [19] [18]. Until then, actions can be taken from software. A monitoring tool, which accurately measures the amount of memory bandwidth per application could be used to arbitrate accesses to the memory bus, making the system more predictable [4].

4. **Varying memory configurations and components**

   Configuration of the memory controller is continuously fine tuned through the development phase, and even after the release of a hardware board. These configuration changes can affect the performance of the memory subsystem.

   Moreover, the first version of the production board is usually followed by several versions, where hardware components are replaced, due to for example cost reduction or provisioning reasons. A previous example showed, using different DDR3-memory circuits with equal speed from another vendor caused performance degradation of the software running on top. Using the monitoring tool along with reproducible workload, tests could be conducted on new hardware versions. In this way, performance degradations could be discovered before starting mass production of new boards.

### 1.3 Proposed solution

In some way, all aforementioned problems are related directly or indirectly to the alteration of a computer system’s performance. To be able to solve any of them, performance of the computer system shall be evaluated first. Therefore, the way to
tackle these issues starts with monitoring various performance metrics. This work suggest to use a general purpose performance monitoring tool, utilizing hardware counters.

1.4 Purpose

Such tool could be beneficial for many players in a complex hardware and software development project:

- **System engineers** could experiment and acquire hard evidence to verify or falsify their assumptions.

- **Software developers** could use it for optimize and debug their applications.

- **Test engineers** could include performance measurements in automated tests and detect performance differences and odd behaviours of a system.

- **Hardware engineers** could use it as a quality assurance tool in cases, when change of settings or components can affect performance.

1.5 Goals

The main goal of this thesis is to implement a performance monitoring tool and carry out various measurements on high-end processing boards. This goal was divided into the following sub-goals:

1. Monitoring tool shall be able to continuously collect data from hardware counters and present the results in an interpretable way.

2. Monitoring tool shall monitor various performance metrics of the memory subsystem.

3. Monitoring tool shall support two different processors, namely Freescale P4080 and Freescale T4240.

4. Monitoring tool shall support two operating systems, namely Linux and a specific real-time operating system.

5. Measurements shall be carried out to compare performance of the two aforementioned operating systems.

6. Measurements shall be carried out to compare performance of the two Freescale processors.
1.6 Delimitations

The monitoring tool only supports the selected Freescale platforms and operating systems. Extra care was taken to minimize architecture specific code, but despite all the efforts it couldn’t be implemented in a pure generic way. Due to hardware and OS architectural differences, some parts of the tool include specific code that needs to be ported, in case of a new target platform or OS. Moreover the tool is constrained to measure selected performance metrics of the memory subsystem. New metrics can be added only programmatically.

Regarding the measurements, this work is not inteded to conduct a holistic benchmark of the selected platforms and operating systems. The goal was to use custom benchmarks and workloads that represent specific telecommunication applications. Therefore, the results of the measurements are biased and cannot be considered representative.

1.7 Structure of this Thesis

The thesis is organized as follows:

- **Chapter 2** presents the target hardware platforms that were used.
- **Chapter 3** gives background information performance monitoring methods and the chosen approach. Furthermore, section 3.2 describes hardware performance counters in general, then section 3.3 goes into details of performance events that were selected to monitor. Last, section 3.4 describes the selected software interface used for collecting data from the hardware counters.
- **Chapter 4** describes all implemented programs in detail.
- **Chapter 5** presents and analyses the results of all measurements.
- **Chapter 6** describes related work and suggests possible future extensions.
- **Chapter 7** summarizes the work and draws conclusion.
- **Appendix A and B** comprise the results and data plots of all conducted tests.
1.8 Methodology

The thesis started out with a literature study on the hardware architecture of the target processors, performance monitoring in general and related work. After, measurements were planned, followed by implementation of the performance monitoring tool and other applications. When all the programs were ready, the measurements were carried out and raw data was collected. Finally, plot scripts were implemented to visualize results, facilitating the analysis.

Measurements were conducted using quantitative method with a rather inductive approach [8]. There were some prior expectations on the outcome of the tests, but no strong hypothesis was defined beforehand that could have been rigorously tested. The goal was not to test theories, but to make observations based on the collected data and explain them. Measurements are described in more details in chapter 5.

The validity of collected data depends on the intrusiveness of the monitoring tool. In principal, the applied software and hardware solutions tried to mitigate this issue, more details can be found in chapter 3.4.2. Measurements can be considered reliable, as they gave consistent results in many different setup using several hardware from the same type. All information that is required to reproduce the tests described along with the measurements.
Chapter 2

Hardware overview

The performance monitoring tool was implemented for two specific processing boards, both of which are used for running large-scale telecommunication applications. One of these processing boards is equipped with two Freescale PowerPC P4080 processors, while the other board has two Freescale PowerPC T4240 processors mounted on it. Both P4080 and T4240 processors are high performance networking platforms, designed for enterprise level switching, routing and other networking applications. The architecture of these boards are very much the same, both have an extensive amount of DDR3-memory and many high speed network interfaces. Thus meaningful comparison is possible to be made between them.

The following sections collect the most important attributes of the two platforms focusing on the memory subsystem including cache hierarchy, parameters of the MMUs and TLBs.

2.1 Freescale PowerPC P4080

The P4080 is most powerful member of Freescale’s P4 series manufactured with 45nm technology, from 2010. The SOC contains eight Power Architecture e500mc cores at frequencies up to 1.5 GHz.

2.1.1 Caches

Each core has 8-way set associative 32/32kB instruction and data L1 cache, and 8-way set associative 128kB unified L2 cache. The chip also has dual 1MB L3 platform caches [21]. See Figure 2.1 for the block diagram.
2.2. Freescale PowerPC T4240

Figure 2.1: P4080 block diagram

2.1.2 Memory management unit

Memory management unit (MMU) has a major impact on performance of the memory subsystem. e500mc cores employ a two-level MMU architecture with separate data and instruction L1 MMUs, and a unified L2 MMU [22]. Moreover, it has support for 4-Kbyte page entries and variable size page (VSP) entries. All in all, the two-level MMU consist of six TLBs:

- Two 8-entry, fully-associative L1 TLB arrays, one for instruction accesses and one for data accesses, supporting variable size pages.
- Two 64-entry, 4-way set-associative L1 TLB arrays (one for instruction accesses and one for data accesses) that support only 4-Kbyte pages.
- A 64-entry, fully-associative unified L2 TLB array supports variable size pages.
- A 512-entry, 4-way set-associative unified L2 TLB array supports only 4-Kbyte pages.

2.2 Freescale PowerPC T4240

T4240 is the flagship of the QoriQ T-series, a state-of-the-art embedded processor equipped with twelve multi-threaded 64-bit e6500 cores, with frequencies scaling
to 1.8 GHz. Due to the multi-threaded cores, the operating system sees 24 virtual processors in total. The processor is manufactured with 28nm technology, from 2012 [23].

### 2.2.1 Caches

T4240 has a very different cache hierarchy than P4080. Each e6500 core has 8-way set associative 32/32kB instruction and data L1 cache, however, the L2 cache is shared among core clusters. There are four cores in one cluster, which share a 16-way set associative 2048kB cache. Each 2048kB L2 cache is broken into 4 banks to support simultaneous access from all cores in the cluster, provided those accesses are to different banks. See Figure 2.2 for the block diagram.

![Figure 2.2: T4240 block diagram](QorIQ T4240 Communications Processor)

### 2.2.2 Memory management unit

e6500 core has very similar MMU as e500mc core [23]:

- Two 8-entry, fully-associative L1 TLB arrays, one for instruction accesses and one for data accesses, supporting variable size pages.
- Two 64-entry, 4-way set-associative L1 TLB arrays (one for instruction accesses and one for data accesses) that support only 4-Kbyte pages.
• A 64-entry, fully-associative unified L2 TLB array supports variable size pages.

• A 1024-entry, 8-way set-associative unified L2 TLB array supports only 4-Kbyte pages.

The L2 MMU has been improved with a 1024-entry, 8-way set-associative unified L2 TLB array for the 4-Kbyte pages. Another major improvement is the support of hardware table-walk.
Chapter 3

Performance monitoring

3.1 Overview

Performance monitoring is the process of collecting and reporting various performance related statistics from a computer system. By using a monitoring tool, system bottlenecks could be tracked down; odd behaviours of applications could be detected; optimize of an algorithm can be done, all of which increase the system’s overall performance [23].

3.1.1 Monitoring methods

Four commonly used methods for monitoring are as follows [9] [10]:

- **Trace driven simulation**: Using an emulator, such as Simics or QEMU, applications performance can be easily and precisely monitored. Moreover, it is possible to change parameters of the model, e.g. cache sizes, thereby enabling analysis of different architectural properties.

- **Hardware monitoring**: Using a logical analyser or an advanced debugger, pure hardware monitoring can be achieved on processors that has support for it. The solution is completely non-intrusive, but usually hard to set up and complicated to use.

- **Software monitoring**: In case of pure software monitoring the source code is extended with functions that capture and record events. The method can be used on any type of hardware, although, it requires the source code and considered to be highly intrusive.

- **Hybrid monitoring**: This method is a combination of software and hardware monitoring. It is mainly used to reduce the impact on the target
system done by software monitoring, and to provide a higher level of abstraction than hardware monitoring alone.

### 3.1.2 Sampling methods

A monitoring tool implements some sort of sampling method, in order to collect data on the target system’s state. There are two fundamental methods to choose from [9] [10]:

- **Time-driven method:** The sampling software uses a high resolution timer. When the timer expires the sampling software is being triggered, which reads the related data and store it as a record. Finally, the timer is re-initialized with a new value.

- **Event-driven method:** In this case, sampling is triggered by the occurrence of a certain event. This event can be anything from a register overflow to a hardware specific debug event.

### 3.1.3 Elected method

The implemented solution uses hybrid monitoring with time-driven sampling, as a small piece of software periodically collects data from hardware performance counters. Hybrid monitoring was selected due to low intrusiveness, portability among architectures, scalability, and because both T4240 and P4080 processors have extensive support for it.

### 3.2 Hardware performance counters

Current trends are driving the need for powerful and sophisticated hardware monitoring [9], as it can help developers to pinpoint problems in their software. Therefore, many of today’s advanced processor supports monitoring of low-level events. These events are counted by a set of hardware performance counters. In most cases, it’s a requirement to be able to monitor many events simultaneously. However, processors are very limited from this perspective, usually having only a handful of counters. To overcome this problem, events shall be grouped together in event sets. Then, using time-dimension multiplexing approach each and every set is sampled for the short period of time. In this way, more events can be sampled than the number of hardware counters at the expense of resolution.
3.2.1 Performance counters in P4080 and T4240

Aforementioned attributes apply for performance counters in P4080 and T4240 processors as well. Each core in these processors has a so called Performance Monitor Unit (PMU), which provides the ability to count performance events. Each PMU in P4080 has four, while in T4240 has six performance monitor counters (PMC). Figure 3.1 shows the block diagram of one PMC out of the six in e6500 core’s PMU.

Each PMC is a 32-bit counter and each counter can count up to 256 different events. The PMCs can be controlled with global (PMGC) and local (PMLC) control registers. All PMC/PMGC/PMLC registers can be read and write using `mtpmr` and `mfpmr` assembly instructions respectively.

By default a PMC counts every occurring event for all the different threads and processes. However, it is possible to count events only when a specific process or thread is running. This can be done easily by writing a bit in the Machine State Register (MSR) for a specific process, which enables the counting of events when it is running.
To get a better understanding of what type of events can be typically monitored the following list includes the most important ones on PowerPC e500mc and e6500 cores, sorted by categories [22] [24].

- **General events**, e.g. processor cycles, completed instructions
- **Instruction types completed**, e.g. completed branch, load, store
- **Branch prediction and execution events**, e.g. branches finished, taken branches, BTB hits
- **Pipeline stalls**, e.g. cycles the instruction queue is not empty but 0 instructions decoded
- **Execution unit idle events**, e.g. cycles load-store/branch/floating-point unit is idle
- **Load/Store and Data cache events**, e.g. Data L1 cache misses, TLB miss cycles, write-through stores translated
- **Fetch and Instruction cache events**, e.g. Instruction L1 cache misses, number of fetches
- **Instruction MMU, Data MMU and L2 MMU events**, e.g. L1 and L2 MMU misses
- **Interrupt events**, e.g. external, critical, system call and TRAP interrupts taken
- **Thread events**, e.g. number of times the Load-Store unit thread priority switched based on resource collisions
- **FPU events**, e.g. FPU finished, FPU stall

### 3.3 Selected metrics for monitoring

To tackle the challenges described in 1.2 most of the selected events are related to the memory subsystem. In the following sections, all event sets are described in details that were implemented in the monitoring tool.
3.3. Selected metrics for monitoring

3.3.1 Cycles/Instruction ratio

**Counted events**

- Completed instructions
- Completed processor cycles

A machine instruction is comprised of a number of micro operations that can be performed in one processor cycle. Depending on the instruction, it can take several cycles to complete. The average value of the Cycle per Instruction (CPI) largely depends on the organization and instruction set architecture [1], still the relative increase of the CPI is a good indicator of performance degradation.

3.3.2 L1 Instruction cache hit-rate and ratio

**Counted events**

- L1 Instruction cache reloads due to demand fetch
- Instructions completed

Both processors are using out-of-order execution. This means, after an instruction is fetched, it is splitted into micro-operations. These micro-operations are dispatched and queued to different functional units in the CPU [1]. Consequently, an instruction cache-miss inhibits the processor to fill up these queues, which can lead to stall cycles. Low instruction cache hit-ratio indicates poor execution flow, likely due to jumpy code or poor BTB hit-ratio. The L1 Instruction cache miss-ratio can be calculated by dividing the number of instruction cache reloads with the number of instructions completed. By subtracting the miss-ratio from 1, we get the hit-ratio. Typical value for these advanced processors is above 0.9 [1].

3.3.3 L1 Data cache hit-rate and ratio

**Counted events**

- Data L1 cache reloads for any reason (read and write)
- Load micro operations completed
- Store micro operations completed
Data cache misses considered to cause less delay than instruction cache misses. In case of cache read miss other instructions can be queued that are not dependent on the data. In almost all cases when a write cache miss happens the operation is queued and subsequent instructions can be executed. The L1 Data cache miss-ratio can be calculated by dividing the number of data cache reloads with the sum of the Load and Store operations. By subtracting the miss-ratio from 1, we get the hit-ratio. Typical value is above for these advanced processors 0.9 [1].

### 3.3.4 L2 Instruction and Data cache hit-rate and ratio
**Counted events**
- L2 Instruction hits
- L2 Instruction accesses
- L2 Data hits
- L2 Data accesses

If L1 cache misses, the L2 cache is checked for the instruction/data. There is a dedicated event for L2 hits and L2 accesses, which makes it easy to calculate the hit ratio. The second and fourth counter counts accesses only that reached the L2 cache.

### 3.3.5 Interrupt/Cycle
**Counted events**
- Processor cycles
- Interrupts taken

Low interrupt/cycle indicates smooth execution. A high value means the core is constantly being interrupted, which results in poor performance and high CPU utilization. Possible causes are endless, starting from buggy code which does alignment corrections, to traffic overload. The number of taken interrupts includes external interrupts, critical input interrupt, system call and TRAP interrupts.

### 3.3.6 L1 Instruction TLB miss-rate and ratio
**Counted events**
- Instructions completed
3.3. SELECTED METRICS FOR MONITORING

- L1 ITLB 4-Kbyte reloads
- L1 ITLB VSP reloads

In case of a TLB miss the address translation proceeds with a page walk, when a process goes through the page table looking for the accessed page and its physical address. Miss penalty typically takes 10-100 cycles [2], while a hit takes 0.5-1 clock-cycle. Worth to mention though, that a TLB miss does not necessarily lead to a cache reload. Typical hit ratio is 99%, which is mainly affected by code size and locality. Both P4080 and T4240 have support for 4-Kbyte and VSP entries, thus both events shall be monitored. The first event counts the total number of instructions, while the second and third counters count the number of reloads. Divide the sum of reloads with number of instructions, we get the miss per instruction.

### 3.3.7 L1 Data TLB miss-rate and ratio

**Counted events**

- Load micro operations completed
- Store micro operations completed
- L1 DTLB 4-Kbyte reloads
- L1 DTLB VSP reloads

Just like the previous event, but counting the data TLB reloads and the number of Load and Store operations completed. Divide the sum of the reloads with the sum of micro-operations, we get the miss per operation.

### 3.3.8 L2 TLB miss rate

**Counted events**

- Instructions completed
- L2 TLB misses

If an entry cannot be found in L1 TLB, the request for translation is forwarded to the L2 MMU which looks up for the entry in the unified L2 TLB. The first event counts the total number of instructions, while the second counter the reloads. Divide number of reloads with number of instructions, we get the miss per instruction.
3.3.9 Branch Target Buffer hit-rate and ratio

Counted events

- Branch instructions, that hit or miss in the BTB and not-taken (a pseudo-hit)
- All finished branch instructions

Branch Target Buffer can dramatically reduce the penalty for the processor’s pipeline. BTB stores the predicted address for the next instruction after a branch is called [1]. If the branch is taken then the next instruction is known, eliminating the penalty to zero cycle. Low BTB hit-ratio means poor branch prediction, which results in poor instruction execution due to penalty cycles of miss-predictions. The first counted event characterizes upper bound of the hit-rate. Dividing it with all the branch instructions gives the hit-ratio of the BTB.

3.4 Perf - software interface

It is fairly easy to configure and collect data from performance counters on an embedded OS. By using privileged assembly instructions described in 3.2, counters can be directly accessed. On the other hand, from Linux user-space these assembly instructions are not available. Instead, the `perf` interface was used, which became part of the kernel from version 2.6.31 in 2009. There were plenty interfaces to choose from, i.e., Oprofile, Perfctr or perfmon2, however, unlike `perf` none of them are officially part of the Linux kernel. Another argument was that `perf`’s interface is quite simple, and no service daemons are needed. Last, according to V. M. Weaver and R. Vitillo profiling using `perf` involves very low overhead [14] [15]. Even though, it is well supported by many Linux distributions, it is not well documented nor well characterized.

3.4.1 Capabilities

First, `perf` offers generalized events. Commonly used events, such as processor cycles, instructions completed, are common among various architectures. Therefore, a piece of code stays portable to another architecture, until it monitors generalized events. Events described in chapter 3.2 are so called raw events, meaning that they are mainly architecture specific.

Another useful capability is that `perf` can be used to count software events, not handled by hardware. Kernel events such as context switches or page faults are exposed to the same interface.
3.4. Perf - software interface

Perf also supports binding the counters to a thread or a process, even among cores. Therefore, counters can be configured in three different ways:

- Count an event in a specific process/thread on any CPU-core
- Count an event in a specific process/thread only when running on the specified CPU-core
- Count an event on a specific CPU-core including all threads/processes

The first two case is useful for application developers in code debugging. However, the performance monitor tool implements the third way, where all processes and threads are monitored on each and every core, in order to see the global system behaviour.

3.4.2 Interface

As it was mentioned before, the interface has only one system call: `sys_perf_event_open(2)` [25]. Performance counters can be configured in one step, by calling this system call. It returns a standard file descriptor, which shall be used in subsequent calls to start, stop and read the counters. The file descriptor corresponds to one event, but events can be grouped together in one file descriptor, thus they can be started/stopped at the same time. Events can be started or stopped via an `ioctl(2)` call or `prctl(2)`. Values can be read out with standard `read(2)` syscall or the file descriptor can also be mapped with `mmap(2)`.

According V. Weaver, the overhead of reading one event varies between 2000-5000 cycles [14], depending on processor architecture and the kernel version of the Linux. Starting and stopping the counters adds a little more overhead, as it takes approximately 4000-30000 cycles to start, read and stop a counter [14].
Chapter 4

Implementation

4.1 Requirements

The following requirements were applicable for all the developed programs:

- **Portability**: All code had to be written in a way that is possible to port to another architecture or another POSIX compliant operating system. Practically, it means exclusive usage of POSIX standard system calls, where it’s possible; strive to minimize architecture-specific code; make the necessary architecture-specific parts easily portable.

- **Modular**: Along with portability, the coding shall be done in a modular way, to be able to add new architectures easily.

- **Small footprint**: Monitoring tool’s code shall be as least intrusive as possible.

- **Robustness**: The code have to be robust and stable, being able to run continuously without crashing.

4.2 Overview

The implementation part mainly consisted of implementing the monitoring tool. Furthermore, to be able to conduct relevant measurements, synthetic workload applications were developed. The following three programs were implemented:

1. **charmon**: The first tool is called *charmon*, stands for characteristic monitor. This program is the performance monitoring tool itself, which sole purpose is to collect the data from the performance counters.
2. **ping-pong**: Ping-pong serves as a workload model for a high-level telecommunication applications. It consist of two processes with the only task to send signals back and forth between each other.

3. **loadsim**: The program loadsim, stands for load-simulator is able to generate static workload, based on a few input parameters.

Since charmon and loadsim are daemon processes, the only way to control and give them commands is using two auxiliary programs, called util-charmon and util-loadsim, which send commands using a POSIX message queue to the charmon and loadsim respectively. Implementation of all the developed programs is summarized on figure 4.1. In the following sections, each program is described in details.

![Software overview - disposition of applications](image)

**Figure 4.1**: Software overview - disposition of applications. Core 0 is used for running charmon and loadsim daemons. Both daemon processes are controlled through a POSIX message queue. All workload, including ping-pong, loadsim or other applications are bind to other cores.

### 4.3 Charmon - characteristic monitor tool

The charmon tool is a general purpose performance monitoring tool. It configures and periodically collects data from the performance counters on each core. It runs as a daemon process, bind to core 0. The daemon can be controlled with util-charmon program by the following commands:

- Start/stop sampling on all cores
4.3. **Charmon - Characteristic Monitor Tool**

- Sample period: set the number of seconds of each sample period, which is one second by default
- Eviction limit: maximum number of samples
- Print samples: dump all collected raw data samples for each event

By starting the sampling, **charmon** starts to execute on core-0 with high scheduling priority. First, it configures the performance counters on all cores with the same set of events. Then starts the counters at the same time on every core and go idle for a sample-period of time. When the period is over it stops the counters on all cores and stores the data in the internal database. Then start all over again by configuring the next event set. Figure 4.2 illustrates the operation of **charmon**.

![Charmon sequence diagram](image_url)

Figure 4.2: **Charmon** sequence diagram.
Printing the samples produce the following output for each measured event set:

<table>
<thead>
<tr>
<th>Core</th>
<th>Interval</th>
<th>CacheReloads</th>
<th>LoadMOpsCompl</th>
<th>StoreMOpsCompl</th>
<th>Hit-ratio</th>
<th>Miss-ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00:00:01</td>
<td>13554</td>
<td>618519</td>
<td>363029</td>
<td>0.9862</td>
<td>0.0138</td>
</tr>
<tr>
<td>1</td>
<td>00:00:01</td>
<td>4694962</td>
<td>269824869</td>
<td>280776914</td>
<td>0.9915</td>
<td>0.0085</td>
</tr>
<tr>
<td>2</td>
<td>00:00:01</td>
<td>4611581</td>
<td>270236996</td>
<td>281174754</td>
<td>0.9916</td>
<td>0.0084</td>
</tr>
<tr>
<td>3</td>
<td>00:00:01</td>
<td>5008557</td>
<td>269765419</td>
<td>280764080</td>
<td>0.9909</td>
<td>0.0091</td>
</tr>
<tr>
<td>4</td>
<td>00:00:01</td>
<td>4672829</td>
<td>269883001</td>
<td>280585162</td>
<td>0.9915</td>
<td>0.0085</td>
</tr>
<tr>
<td>5</td>
<td>00:00:01</td>
<td>4935537</td>
<td>270045705</td>
<td>281029316</td>
<td>0.9912</td>
<td>0.0088</td>
</tr>
<tr>
<td>6</td>
<td>00:00:01</td>
<td>4811531</td>
<td>270572444</td>
<td>281444849</td>
<td>0.9913</td>
<td>0.0087</td>
</tr>
<tr>
<td>7</td>
<td>00:00:01</td>
<td>5038306</td>
<td>270202028</td>
<td>28145053</td>
<td>0.9909</td>
<td>0.0091</td>
</tr>
<tr>
<td>0</td>
<td>00:00:01</td>
<td>8328</td>
<td>399605</td>
<td>230017</td>
<td>0.9868</td>
<td>0.0132</td>
</tr>
<tr>
<td>1</td>
<td>00:00:01</td>
<td>6657942</td>
<td>246195554</td>
<td>195404340</td>
<td>0.9849</td>
<td>0.0151</td>
</tr>
<tr>
<td>2</td>
<td>00:00:01</td>
<td>6389578</td>
<td>247652817</td>
<td>196878598</td>
<td>0.9856</td>
<td>0.0144</td>
</tr>
<tr>
<td>3</td>
<td>00:00:01</td>
<td>6694344</td>
<td>246961453</td>
<td>196098300</td>
<td>0.9849</td>
<td>0.0151</td>
</tr>
<tr>
<td>4</td>
<td>00:00:01</td>
<td>6598593</td>
<td>246319632</td>
<td>195587885</td>
<td>0.9851</td>
<td>0.0149</td>
</tr>
<tr>
<td>5</td>
<td>00:00:01</td>
<td>6491375</td>
<td>247738367</td>
<td>197019855</td>
<td>0.9854</td>
<td>0.0146</td>
</tr>
<tr>
<td>6</td>
<td>00:00:01</td>
<td>6618439</td>
<td>247418414</td>
<td>196478120</td>
<td>0.9851</td>
<td>0.0149</td>
</tr>
<tr>
<td>7</td>
<td>00:00:01</td>
<td>6625481</td>
<td>247266110</td>
<td>196443528</td>
<td>0.9851</td>
<td>0.0149</td>
</tr>
</tbody>
</table>

This data set represents two one-second long measurements of the number of cache reload, load and store operations on all cores. Naturally, the internal database can store a large number of such data sets. Furthermore, this data can be converted to a plot, like on figure 4.3, which makes analysis easier.

![L1 Data cache - Cache reload rate](image)

Figure 4.3: Sample plot of number of cache reloads on P4080

As it was described in chapter 3.2, both P4080 and T4240 processors have very limited number of performance counters per core. Therefore, the charmon implementation uses the time-dimension multiplexed method. This puts restriction on simultaneous data collection. All implemented event sets (chapter 3.3) can not be measured simultaneously, only in a quasi parallel, time multiplexed way. To monitor all the defined event sets, it takes the sample-period times the number of event sets to collect all the samples. In our case, with 1 second of sample-period it takes 9 seconds to measure all the 9 different event sets.
4.4 Ping-pong

Typical high-level telecommunication applications perform significant amount of signalling. The ping-pong program models this type of workload in a simple way, moreover it measures the turnaround time it takes to send a signal. Ping-pong consists of two almost identical processes. After initialization, the first process starts a time measurement and send a request signal to the second process. After receiving the signal, the second process changes its content\(^*\) then send it back. When the first process receive the reply signal, it stops the time measurement and calculate the turnaround time it took to send the signal back and forth, then sends the next signal. Figure 4.4 shows the sequence diagram of the ping-pong processes. Ping-pong was implemented for Linux and for a specific embedded RTOS.

4.5 Loadsim - load simulator tool

The load simulator tool is capable to synthetically generate L1 instruction, L1 data and L2 data cache misses on a specific core, based on a few input parameters. L1 instruction misses are generated by jumping in large switch-case statement. L1 and L2 data cache misses are generated by accessing the memory with different strides. The code was given for P4080 on the RTOS from M. Jägermar’s previous work [4] [5] and it was ported to Linux.

The goal with using loadsim was to reproduce the average number of cache misses of a real workload. Data on the average cache miss rate was already available from measurements that were conducted on a production board, which was running real applications on the field. This data was the input for loadsim to generate the same number of cache misses.

\(^*\) The content is a 2kB buffer, which is set to zero by the first process and set to one by the second process.
Figure 4.4: Ping-pong sequence diagram.
Chapter 5

Measurements

To demonstrate charmon’s capabilities, various measurements were carried out. There were no intention to set any sort of hypothesis prior to the tests. Mainly, because they are not fully comprehensive, thus they cannot be considered as a representative benchmark.

The following measurements were conducted:

1. Comparing an RTOS and Linux operating systems

2. Comparing P4080 and T4240 processors

5.1 Comparing between an RTOS and Linux on P4080

It is certainly a hard decision to change the operating system on a product, but sometimes it is inevitable. As the system evolves, it can reach a complexity that requires a more complex OS. The first test is trying to compare two operating systems. One of them is a powerful embedded real-time OS, which was fine tuned in many aspects for telecommunication applications. The other operating-system is Freescale’s Linux distribution with a 2.6 kernel.

Probably the most important advantage of Linux over a special RTOS is the enormous support provided by the community behind it. Moreover, Linux can handle highly complex systems, and also it is easier to hire developers, since more people have expertise in it. On the other hand, an RTOS is expected to be much faster and effective for a task; debugging in general and developing real-time applications can be way more easier compared to Linux.
5.1. Comparing between an RTOS and Linux on P4080

5.1.1 Workload

In this measurement the idea was to use the developed ping-pong and loadsim as a custom benchmark instead of standard ones. The reason is that a custom benchmark can better represent the applications normally running on the OS. Although, it makes the benchmark very specific, probably less portable and more complex.

Extra care was taken to make sure that both ping-pong and loadsim is comparable on both operating systems. This means, that the RTOS and the Linux version of both tools have identical code, except a few parts that OS specific. These parts were developed considering efficiency and best practices. Naturally, the same compiler flags and attributes were used for building the tools.

5.1.2 Test cycle

The following test cycle was executed on both operating systems:

1. **Phase-1**: Run only ping-pong for 10 seconds.
2. **Phase-2**: Run ping-pong and loadsim together for 10 seconds.
3. **Phase-3**: Run only loadsim for 10 seconds.

Note, that loadsim was generating the same number of L1 and L2 cache misses as measured on the field. Therefore, phase-2 was considered as a representative scenario. The test was conducted on a board with the P4080 processor, which has eight cores in total. Both ping-pong and loadsim workload was running on all cores from core-1 to core-7. Note that core-0 was used by the charmon and loadsim daemon, thus core-0 is not displayed on the plots. Figure 5.1 shows the test cycle on a result plot.
5.1 Comparing Between an RTOS and Linux on P4080

5.1.3 Observations

The following observations were made, based on the plots generated from the data. Observations are reflecting on differences, rather than similarities. All the figures can be found in appendix A.

- **Signalling time**: The most anticipated data was the difference in the signalling time, particularly in phase-2, as it represents a typical workload of the board. The result are presented on figure A.1a and A.1b. In phase-1 ping-pong being the only workload, signalling takes almost three times longer on Linux (17 micro-seconds), than on the RTOS (6 micro-seconds). However, in phase-2, when cache misses are generated with loadsim, the difference shrinks significantly, as it takes 25 micro-seconds on the RTOS, while 29 micro-seconds on Linux.

  The result shows that the RTOS has much more effective implementation for sending signals. It is way more effective than using sockets on Linux, which is considered to be effective and commonly used. However, when the CPU gets saturated by the load in phase-2, the difference shrinks between the two.

- **L1 Instruction cache**: Figure A.2 shows that in phase-1, the operating system’s working set fits in the cache on RTOS, as the hit-ratio equals to 1. While on Linux, the hit-ratio is lower as the working set does not fit in to the cache.
5.1. Comparing between an RTOS and Linux on P4080

- **L1 Instruction TLB**: Figure A.3 shows some startling results. Both the number of 4-Kbyte and VSP TLB entry reloads are roughly 500 on the RTOS, which means that the instruction working set seems to fit in very well to the TLB. On Linux however, there are almost 800,000 reloads per second in phase-1, three magnitudes more than on the RTOS. This is probably due to the combined effect of high number of context-switches and larger working set and background footprint of Linux.

- **L1 Data cache**: In phase-1 on figure A.4 when only the ping-pong is running, the number of cache reloads are almost 5x times on Linux compared to the RTOS. This shows that the background footprint of Linux is bigger and implementation of signalling is less effective. In phase-2, when the CPU gets saturated, almost the same amount of reloads can be observed.

- **L1 Data TLB**: Similar to the instruction TLB, figure A.5 shows that RTOS has way less data TLB reloads than Linux. It has to be pointed out here, that the usage of VSP TLBs were previously fine tuned on the RTOS. While Linux doesn’t make use of the variable size page TLB, which in Linux terminology is called Transparent Huge Pages (THP). In most distributions, it is turned off by default, and there are many reports on performance issues with it [16]. In this case however, RTOS performs better by using them, which shows an opportunity to boost performance on Linux for a specific application by enabling THP.

- **L2 Instruction cache**: Figure A.7 shows that RTOS has roughly half L2 instruction cache accesses than Linux in phase-2. This figure also demonstrates the importance of evaluating the hit-ratio together with the hit-rate. Just by looking at the hit-ratio in phase-1, one can deduct that Linux performs better. While checking the rate, it turns out that basically, there are no L2 instruction accesses on the RTOS, because the instruction working set fits in to the L1 cache. L2 cache is simply cold, and when there is an access it is going to be a miss with a good chance.

- **L2 TLB**: Figure A.9 shows that on Linux L2 cache performs no better than L1 TLBs. While on Linux, the number of reloads are roughly 100,000, on the RTOS it is negligible.

- **Interrupts**: Figure A.6 shows the number of interrupts on the two operating systems. Linux takes between 100-250 times more interrupts per second then the RTOS. While the number of all taken interrupts is between 2000-5000 on the RTOS, on Linux, it can go up till 250,000. Almost all
the interrupts are software (TRAP) interrupts on Linux. TRAP interrupts number is linked with the number of context switches, in which Linux is known to perform worse. It has to be pointed out though, that a context-switch on this RTOS does not TRAP the kernel, since applications are running in supervisor mode. Therefore, they does not count as an interrupt.

- **BTB hit-ratio**: On figure A.11, RTOS has a decent BTB hit-ratio, almost always above 95%. Linux on the other hand has an almost stable 85%, which is a quite bad value. Normally, the programmer cannot directly influence the BTB hit-ratio. However, GCC has a so called _builtin_expect() macro, which can be used to give a clue to the compiler on the likelihood, whether a branch is taken or not [27] [26].

- **Cycles/Instruction**: On figure A.10b, in phase-3, when only loadsim is running, there’s a massive increase in CPI on Linux. It goes up from 1.8 to 5 cycles per instruction, while RTOS keeps it below 2 in all phases. This basically shows, how much the generated cache misses can degrade performance.

To summarize the observations, almost on all figures Linux has worse values compared to the RTOS, which meets our prior conjecture. When the cores have less workload, the difference is more significant. Increasing the load makes their performance more even. Though, there is space for improvement on Linux, as it could be tailored for specific applications. For example, making use of the variable size page TLB entries.

### 5.2 Comparing P4080 and T4240

This measurement was conducted to compare two similar processors, which meant to do the same tasks and running the same applications. In our case, the older P4080 was compared to the more recent and powerful T4240.

The approach was changed from the previous measurement, where custom benchmarks were used. In this case, general purpose workloads were executed, using open-source tools.

#### 5.2.1 Workload

Three different workloads were used to exercise cores from three different aspects. These aspects were: CPU, memory subsystem and network.

- Memtest
5.2. Comparing P4080 and T4240

Memtester is a simple and effective open-source tool for stress-testing the memory subsystem [28]. Although, it is mainly used to catch memory errors, in this case, it was used as a workload to stress the caches. Memtester allocates a chunk of memory, which was set to 1GB. Then it goes through a list of tests, such as random value read and write, walking 0/1, sequential increment. Version 4.2.2 was used for the tests.

• Iperf3

Iperf3 is an open-source tool for generating network traffic and measuring the achievable bandwidth on IP networks [29]. During the test one of the 1G Ethernet interfaces was used and fully utilized with traffic. Version 2.0.5 was used for the tests.

• Dhrystone

Dhrystone is a synthetic CPU benchmark, designed to statistically imitate common CPU usage, thereby representing a general CPU performance [30]. Version 2.1 was used for the tests.

On both platforms Freescale’s Linux distribution was running with kernel version 3.8.13 on P4080 and 3.12.19 on T4240. All the tools were compiled with the same tool-chain, without any core specific flag. Since both processors are PowerPC, the same binaries were used to run the tests.

5.2.2 Test cycle

The test cycle consisted of the following workloads:

• memtester - bind to core-2 (green), going through all the different tests in an infinite loop

• dhrystone - bind to core-4 (purple), executing 300 million loops

• iperf3 - bind to core-7 (black), continuously utilizing the 1G link, effectively with 960Mbit/s bandwidth

Note that all other cores were idle, except core-0, which was used by the charmon daemon, thus, it is not displayed on the plots. Figure 5.2 shows the test cycle on a sample result plot. Also worth to mention, that this workload distribution among the cores makes it more even for P4080, because the first 8 cores on T4240 shares the same L2 cache.
5.2. Comparing P4080 and T4240

5.2.3 Observations

The following observations were made, based on the plots generated from the data. Plots can be found in appendix B. These observations are only reflecting on significant differences between the two platforms.

- **dhrystone**: The e6500 core on T4240 executed the benchmark 55% faster than the e500mc on P4080. Results can be found in appendix B.1.

- **L1 Instruction cache**: The faster execution of dhrystone is clearly visible on figure B.1, as the core-4 can execute more instructions on T4240 than on P4080. Moreover, network load on T4240 core-7 performs a little better in hit-ratio.

- **L1 Instruction TLB**: Figure B.2 shows, while on P4080, there is a steady number of L1 ITLB reloads on core-2 (memtester), basically no L1 ITLB reloads happened on the T4240.
  Some strange spikes can also be observed on figure B.2b, which cannot be explained by the executed workload. Linux probably schedules something, that flushes the TLB entries.

- **L1 Data cache**: Figure B.3 shows some unexpected results. On T4240, L1D cache hit-ratio is worse with 3% and L1D cache reload rate is three times more on core-7, which runs the network traffic. This is strange, especially because memtester and dhrystone have roughly the same
behaviour, and the number of executed instructions are the same on both processors (figure B.1). Most probably, it is caused by an inefficient, buggy network driver.

There is an interesting phenomenon that can be observed on figure B.3d, which is called cache piracy. The e6500 core is multi-threaded, but the two threads appears in Linux as two separate CPU. In reality though, these CPUs share the L1 caches. That is why the idle core-3 has the same amount of L1D cache reloads as core-2, which actually runs the memtester. This phenomenon is called cache piracy or cache bandwidth bandit [20]. An application’s performance is strongly influenced by the available cache size [17], moreover, cache stealing may result in the loss of predictability [20].

- **L2 Instruction cache**: Figure B.6 shows that T4240 has almost constant 99.9% hit-ratio, while P4080 has significantly worse values on core-2 (memtester) and on core-7 (iperf3). This can be easily explained with the significantly improved L2 cache on T4240, thereby all the instruction working set fit in.

- **L2 Data cache**: Figure B.7 clearly shows the effect of the different L2 cache architectures. On P4080, each core has a separate L2 cache, consequently there is no load on idle cores. While L2 cache on the T4240 is shared among four e6500 cores which consist of eight virtual cores. Therefore it seems that there are L2 accesses on idle cores and the hit-ratios from core-0 to core-7 are moving together as these cores belong to the same cluster.

- **L2 TLB**: Again, figure B.8 shows that there is a bug in the network driver on T4240, as there is a significant amount of L2 TLB reloads on T4240 on core-7 (iperf3), while it is almost zero on P4080.

To summarize the observations, T4240 not only has three times more cores, but each e6500 core is faster than the e500mc counterpart on P4080. Higher frequency, larger L2 caches, advanced MMU with hardware table-walk makes it more powerful than it’s predecessor, meanwhile maintaining the same level of power efficiency.

The unexpected behaviour of the network workload triggered some questions, which need to be clarified. This one example shows the importance of the monitoring tool, as a buggy application or driver can easily degrade the system’s overall performance.

Last, cache piracy can adversely affect the cache capacity in multi-core processors, which leads to increased off-chip memory bandwidth. This shall be taken into consideration, when it comes to scaling of an application.
Chapter 6

Related and future work

6.1 Related work

This work is heavily influenced by the papers [4] [5] [6] [7] from M. Jägermar. It can be considered as a continuation of his work. The terminology and part of the code was reused or ported during the implementation. These parts are is pointed out in the text.

Martin Collberg and Erik Hugne has implemented a similar monitoring tool on a PowerPC platform (PowerPC 750) [9]. They are using a similar approach for the sampling, but using different software interfaces and measuring other types of events.

Anderson also implements a low intrusive system wide monitoring tool [13] using different a sampling method which is based on interrupts generated by performance counters.

Eranian [11] demonstrates on x86 platform that performance counters are the key hardware resources in order to better understand issues related to memory subsystem.

6.2 Future work

6.2.1 Charmon

The charmon performance monitoring tool will be productified and further used internally in the company. The following features could be added to improve and widen the possible use-cases of the tool.

- **Monitor performance per process basis**: Currently, the tool monitors global system behaviour as it counts events for all running processes
executed on a specific core. Using the `perf` interface, it is possible to configure counters to only count events for specific processes. This feature could be particularly useful for application developers, as they could debug specific applications and processes easier.

- **Setting performance events dynamically**: Adding or changing event types can be done only programmatically. Logical improvement could be to implement dynamic configuration of the events from command line.

### 6.2.2 Workload applications

Synthetic workload applications (*ping-pong, loadsim*) could be further optimized to better imitate real workloads. First, measurements with *charmon* shall be carried out on production hardware, running real applications on the field. Results would clearly indicate how well the synthetic workload mimics real workload. Then refine the synthetic workloads until good approximation is reached.
Chapter 7

Conclusion

The thesis was set out to implement a performance monitoring tool capable of monitoring global system behaviour, which can help to tackle the problems listed in chapter 1.2. This goal has been achieved in compliance with the requirements listed in chapter 4.1, although it took more time to complete than planned, due mainly to the complex hardware and software environment. Operation of the tool has been demonstrated by performing various measurements on different platforms and operating systems. Results of these measurements mostly met prior expectations, but also revealed some unexpected and odd behaviours, which raised questions needed to be clarified.

Expertise was gained in many areas, such as architectural understanding of memory subsystem, operating systems and multi-core programming. Moreover, insights were gained in areas not closely related to the topic, for instance, build system of the Linux kernel (yocto) and gnuplot programming.

Finally, I believe the study affirmed, that performance monitoring of a complex system is crucial. Not only it can reveal problems, and trace down bottlenecks, but it gives an ultimate understanding on the system’s performance and deliver smoking-gun evidence for questions that would be hard to answer otherwise.
Bibliography


[22] e500mc Core Reference Manual (2013)


[25] Linux Programmer’s Manual for `perf_event_open(2)`
   <http://man7.org/linux/man-pages/man2/perf_event_open.2.html>


[27] Built-in functions provided by GCC
   <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>

[28] Pyropus Technology - Memtester
   <http://pyropus.ca/software/memtester/>

[29] ESnet, Lawrence Berkeley National Laboratory - Iperf3
   <http://software.es.net/iperf/>

[30] Dhrystone wikipedia
Appendix A

Test results of comparing RTOS and Linux

A.1 Data plots - comparing RTOS and Linux
A.1. DATA PLOTS - COMPARING RTOS AND LINUX

Figure A.1: Signal turnaround time

(a) RTOS

(b) Linux

Figure A.2: L1 Instruction Cache - Hit ratio and Instruction accesses

(a) RTOS

(b) Linux

(c) RTOS

(d) Linux
Figure A.3: L1 Instruction TLB reloads
Figure A.4: L1 Data Cache - Hit ratio and number of cache reloads
A.1. DATA PLOTS - COMPARING RTOS AND LINUX

Figure A.5: L1 Data TLB reloads

(a) RTOS

(b) Linux

(c) RTOS

(d) Linux

Figure A.6: Number of interrupts

(a) RTOS

(b) Linux
A.1. DATA PLOTS - COMPARING RTOS AND LINUX

Figure A.7: L2 Instruction Cache - Hit ratio and instruction accesses

(a) RTOS

(b) Linux

(c) RTOS

(d) Linux
A.1. DATA PLOTS - COMPARING RTOS AND LINUX

Figure A.8: L2 Data Cache - Hit ratio and data accesses

- (a) RTOS
- (b) Linux
- (c) RTOS
- (d) Linux

Figure A.9: L2 TLB reloads

- (a) RTOS
- (b) Linux
A.1. DATA PLOTS - COMPARING RTOS AND LINUX

Figure A.10: Cycles/Instruction

(a) RTOS

(b) Linux

Figure A.11: Branch Target Buffer - Hit ratio and hit rate

(c) RTOS

(d) Linux
Appendix B

Test results of comparing P4080 and T4240

B.1 Dhrystone results

B.1.1 P4080
Dhrystone Benchmark, Version 2.1 (Language: C)
Program compiled without ‘register’ attribute
Please give the number of runs through the benchmark:
Execution starts, 300000000 runs through Dhrystone
Microseconds for one run through Dhrystone: 0.3
Dhrystones per Second: 3225806.5

real 1m33.278s
user 1m30.260s
sys 0m2.996s

B.1.2 T4240
Dhrystone Benchmark, Version 2.1 (Language: C)
Program compiled without ‘register’ attribute
Please give the number of runs through the benchmark:
Execution starts, 300000000 runs through Dhrystone
Microseconds for one run through Dhrystone: 0.2
Dhrystones per Second: 4687500.0

real 1m4.034s
user 1m3.595s
sys 0m0.431s

B.2 Data plots - comparing P4080 and T4240
B.2. DATA PLOTS - COMPARING P4080 AND T4240

Figure B.1: L1 Instruction Cache - Hit ratio and Instruction accesses

Figure B.2: L1 Instruction 4-Kbyte TLB reloads
B.2. DATA PLOTS - COMPARING P4080 AND T4240

Figure B.3: L1 Data Cache - Hit ratio and number of cache reloads

(a) P4080

(b) T4240

(c) P4080

(d) T4240

Figure B.4: L1 Data 4-Kbyte TLB reloads

(a) P4080

(b) T4240
B.2. DATA PLOTS - COMPARING P4080 AND T4240

Figure B.5: Number of interrupts

(a) P4080

(b) T4240

Figure B.6: L2 Instruction Cache - Hit ratio and instruction accesses

(a) P4080

(b) T4240

(c) P4080

(d) T4240
B.2. DATA PLOTS - COMPARING P4080 AND T4240

Figure B.7: L2 Data Cache - Hit ratio and data accesses

Figure B.8: L2 TLB reloads
B.2. DATA PLOTS - COMPARING P4080 AND T4240

Figure B.9: Cycles/Instruction

(a) P4080

(b) T4240

Figure B.10: Branch Target Buffer - Hit ratio and hit rate

(c) P4080

(d) T4240