MASTER

Time-predictability of a computer system

van der Put, J.B.W.

Award date:
2010

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
Time-predictability of a computer system

Wouter van der Put

April 2010
Abstract

This master thesis describes the research that has been done for the master project that has been performed at Prodrive B.V..
This research investigates the influences on the running time of an application. These influences are separated into five levels: processor core, memory hierarchy, system architecture, operating system and application. The influences on the running time are described and formulated in abstract formulas using literature studies in combination with results from experiments. These tools give insight into a complete computer system which can be used to upgrade current systems and/or design future (hard) real-time embedded systems.

Author

J.B.W. (Wouter) van der Put
ID: 0539553
J.B.W.v.d.Put@student.TUe.nl
WoutPut@Hotmail.com

Supervisors

Prodrive B.V.
André Boon (Andre.Boon@Prodrive.nl)
Dion Cornelissen (Dion.Cornelissen@Prodrive.nl)

TU/e - Department of Electrical Engineering - Electronic Systems
Bart Mesman (B.Mesman@TUe.nl)

Assessment committee

Prodrive B.V.
Advisory member: André Boon (Andre.Boon@Prodrive.nl)

TU/e - Department of Electrical Engineering - Electronic Systems
Voting member: Henk Corporaal (H.Corporaal@TUe.nl)
Voting member: Bart Mesman (B.Mesman@TUe.nl)

TU/e - Department of Mathematics and Computer Science - System Architecture and Networking
Voting member: Reinder Bril (R.J.Bril@TUe.nl)

Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>29-01-10</td>
<td>First version</td>
</tr>
<tr>
<td>V2</td>
<td>25-03-10</td>
<td>Improved most chapters; changed to top-down structure; included chapters 7-10</td>
</tr>
<tr>
<td>V3</td>
<td>14-04-10</td>
<td>Improved most chapters; changed to bottom-up structure; expanded conclusion</td>
</tr>
<tr>
<td>V4</td>
<td>28-04-10</td>
<td>Improved most chapters; expanded conclusion and future work</td>
</tr>
</tbody>
</table>

Review List

<table>
<thead>
<tr>
<th>Organisation</th>
<th>Reviewer</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prodrive B.V.</td>
<td>André Boon</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Prodrive B.V.</td>
<td>Dion Cornelissen</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Prodrive B.V.</td>
<td>Glen de Brujinje</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TU Eindhoven</td>
<td>Gert-Jan van den Braak</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TU Eindhoven</td>
<td>Bart Mesman</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
## Contents

1. **INTRODUCTION** .......................................................................................................................... 6  
   1.1. CONTEXT .................................................................................................................................. 6  
   1.2. PROBLEM STATEMENT ............................................................................................................ 6  
   1.3. GOAL ..................................................................................................................................... 7  
   1.4. APPROACH ............................................................................................................................. 8  
   1.5. DEFINITIONS AND ABBREVIATIONS ..................................................................................... 8  

2. **PROCESSOR CORE** .................................................................................................................... 11  
   2.1. INTRODUCTION ....................................................................................................................... 11  
   2.2. HISTORICAL OVERVIEW OF CPUs ....................................................................................... 11  
   2.3. PROCESSOR PIPELINES ........................................................................................................ 13  
   2.4. HAZARDS IN PIPELINED PROCESSORS ................................................................................ 14  
   2.5. IMPROVEMENTS .................................................................................................................. 15  
   2.6. MEASUREMENTS .................................................................................................................... 16  
   2.7. CONCLUSION ......................................................................................................................... 18  

3. **MEMORY HIERARCHY** ............................................................................................................... 19  
   3.1. INTRODUCTION ....................................................................................................................... 19  
   3.2. HIERARCHY ............................................................................................................................ 19  
   3.3. CACHE DETAILS ....................................................................................................................... 20  
   3.4. MEASUREMENTS ..................................................................................................................... 22  
   3.5. CONCLUSION ......................................................................................................................... 24  

4. **SYSTEM ARCHITECTURE** ....................................................................................................... 25  
   4.1. INTRODUCTION ....................................................................................................................... 25  
   4.2. BOTTLENECKS ....................................................................................................................... 26  
   4.3. MEASUREMENTS ..................................................................................................................... 26  
   4.4. CONCLUSION ......................................................................................................................... 29  

5. **OPERATING SYSTEM** ............................................................................................................... 30  
   5.1. INTRODUCTION ....................................................................................................................... 30  
   5.2. PRIORITIES ............................................................................................................................ 30  
   5.3. INTERRUPTS ............................................................................................................................ 30  
   5.4. MEASUREMENTS ..................................................................................................................... 30  
   5.5. CONCLUSION ......................................................................................................................... 31  

6. **APPLICATION INPUT DATA** .................................................................................................. 32  
   6.1. INTRODUCTION ....................................................................................................................... 32  
   6.2. IRREGULAR DATA PROCESSING .............................................................................................. 32  
   6.3. CONCLUSION ......................................................................................................................... 33  

7. **USE-CASE** ............................................................................................................................. 34  
   7.1. INTRODUCTION ....................................................................................................................... 34  
   7.2. EXTRACTING PARAMETERS ................................................................................................... 34  
   7.3. CONCLUSION ......................................................................................................................... 37  

8. **RELATED WORK** ...................................................................................................................... 8  

9. **CONCLUSION** .......................................................................................................................... 38  
   9.1. INTRODUCTION ....................................................................................................................... 38  
   9.2. PROCESSOR CORE ................................................................................................................ 38  
   9.3. MEMORY HIERARCHY ............................................................................................................ 39  
   9.4. SYSTEM ARCHITECTURE ..................................................................................................... 39  
   9.5. OPERATING SYSTEM ............................................................................................................ 40  
   9.6. APPLICATION INPUT DATA .................................................................................................. 40  
   9.7. GENERAL CONCLUSION ...................................................................................................... 41
10. FUTURE WORK ............................................................................................................. 42
    10.1. CPU/MEMORY FREQUENCY SCALING ................................................................. 42
    10.2. GRAPHICAL USER INTERFACE .......................................................................... 42

APPENDIX A DETAILS OF INVESTIGATED COMPUTER SYSTEMS .................................. 43
    A.1 COMPUTER SYSTEM 1 .......................................................................................... 43
    A.2 COMPUTER SYSTEM 2 .......................................................................................... 44
    A.3 COMPUTER SYSTEM 3 .......................................................................................... 45

APPENDIX B SOURCE CODE OF TEST PROGRAMS ....................................................... 49
    B.1 USED AT OPERATING SYSTEM LEVEL ................................................................. 49
    B.2 USED AT SYSTEM ARCHITECTURE LEVEL .......................................................... 49
    B.3 USED AT MEMORY HIERARCHY LEVEL ............................................................... 51
    B.4 USED AT PROCESSOR CORE LEVEL .................................................................... 51

APPENDIX C EVENT BASED SAMPLING DETAILS ...................................................... 52
    C.1 RATIOS ................................................................................................................ 52
    C.2 EVENTS .............................................................................................................. 52
List of Figures

FIGURE 1.1. EXAMPLE OF WORST AND BEST CASE EXECUTION TIMES AND THEIR ESTIMATES [WEE '08] ......................................................................................................................... 7
FIGURE 2.1. COMPUTER SYSTEM OVERVIEW - PROCESSOR CORE HIGHLIGHTED ........................................ 11
FIGURE 2.2. CLASSIS RISC PIPELINE [PH97] .................................................................................................... 13
FIGURE 2.3. NEHALEM MICROARCHITECTURE PIPELINE FUNCTIONALITY [DAV09] ............................. 14
FIGURE 2.4. EFFECT OF BRANCH PREDICTION ON COMPUTER SYSTEM 1 (INTEL CORE 2 DUO E8500 3.16GHz) ........................................................................................................ 17
FIGURE 2.5. EFFECT OF BRANCH PREDICTION ON COMPUTER SYSTEM 3 (INTEL XEON X5560 3.06 GHz) ....................................................................................................................... 17
FIGURE 3.1. COMPUTER SYSTEM OVERVIEW - MEMORY HIERARCHY HIGHLIGHTED ................................. 19
FIGURE 3.2. VISUALISATION OF SIZE AND LATENCY IN A MEMORY HIERARCHY [FH01][EEKS06] .......... 20
FIGURE 3.3. L1 DATA CACHE MISS RATE AND L2 CACHE MISS RATE (SEE APPENDIX C FOR DETAILS) ...... 23
FIGURE 3.4. L1 DATA CACHE MISS PERFORMANCE IMPACT, BUS UTILIZATION AND TLB MISS PENALTY (SEE APPENDIX C FOR DETAILS) ........................................................................ 23
FIGURE 3.5. CLOCKS PER INSTRUCTION – CPI (SEE APPENDIX C FOR DETAILS) ........................................ 24
FIGURE 4.1. COMPUTER SYSTEM OVERVIEW - SYSTEM ARCHITECTURE HIGHLIGHTED ........................................ 25
FIGURE 4.2. SEVERAL SYSTEM ARCHITECTURES [INTELE5540][INTELE8500][INTELL5410][INTELX5560] ................................................................................................................................. 26
FIGURE 4.3. INTEL SERVER BOARD S5520HC FUNCTIONAL BLOCK DIAGRAM [INTELS5520HC] ............ 27
FIGURE 4.4. READING MEMORY FROM MULTIPLE THREADS ON COMPUTER SYSTEM 3 (TWO QUAD core CPUs) ................................................................................................................... 28
FIGURE 4.5. STANDARD DEVIATION WHILE READING MEMORY FROM MULTIPLE THREADS .................. 29
FIGURE 5.1. COMPUTER SYSTEM OVERVIEW - OPERATING SYSTEM HIGHLIGHTED ..................................... 30
FIGURE 5.2. RUNNING TIMES SHOWN ON A TIMELINE AND IN A HISTOGRAM ........................................... 31
FIGURE 6.1. COMPUTER SYSTEM OVERVIEW - APPLICATION HIGHLIGHTED ............................................. 32
FIGURE 7.1. THREAD-DEPENDENT RUNNING TIMES ................................................................................... 35
FIGURE 7.2. TIME-DEPENDENT RUNNING TIMES ...................................................................................... 36
FIGURE 7.3. INPUT-DEPENDENT RUNNING TIMES HISTOGRAM ................................................................. 37
FIGURE 10.1. COMPUTER SYSTEM 1 BLOCK DIAGRAM ................................................................................. 43
FIGURE 10.2. COMPUTER SYSTEM 2 BLOCK DIAGRAM ................................................................................. 45
FIGURE 10.3. COMPUTER SYSTEM 3 BLOCK DIAGRAM ................................................................................. 46

List of Tables

TABLE 2.1. HISTORICAL OVERVIEW OF CPUS [FH01] .................................................................................. 12
TABLE 2.2. ISA EXTENSIONS [DAV09] .............................................................................................................. 13
TABLE 3.1. EXAMPLES OF CACHE EFFECT ................................................................................................... 22
TABLE 9.1. VARIABLES ON PROCESSOR CORE LEVEL .................................................................................. 38
TABLE 9.2. VARIABLES ON MEMORY HIERARCHY LEVEL ............................................................................. 39
TABLE 9.3. VARIABLES ON SYSTEM ARCHITECTURE LEVEL ...................................................................... 40
TABLE 9.4. VARIABLES ON OPERATING SYSTEM LEVEL ............................................................................... 40
TABLE 9.5. VARIABLES ON APPLICATION LEVEL ........................................................................................ 41
TABLE 10.1. CACHE PARAMETERS OF PROCESSORS BASED ON NEHALEM MICROARCHITECTURE [INTELNEHALEM] ............................................................................................................... 46
1. **Introduction**

1.1. **Context**

1.1.1. **Demand**

Tomorrow’s needs demand smarter devices. To fulfill these needs, more data needs to be processed in less time. Hence, getting grip on timing is crucial. Timing is not only important in classical hard real-time applications such as automotive applications (airbag), flight control systems and space missions. These applications typically only require a small amount of processing, but have stringent requirements on the minimum and, more importantly, the maximum response time.

Multimedia, entertainment and medical applications also demand strict timing behaviour. For example, both a Full-HD 100 Hz TV and a digital pathology slide scanner both need to process a lot of data in a short amount of time to be useful. In addition to a short response time, a lot of computational power is required to meet these strict needs. This combination of requirements calls for even more advanced systems.

1.1.2. **Supply**

Various state of the art embedded systems, including high speed I/O communication networks and High Definition video applications, demand stringent constraints on latency and throughput of the data processing unit(s). For this reason, time-predictable platforms like Field Programmable Gate Arrays (FPGAs) are frequently used to meet these requirements [AMV'07][LAW06][VA08]. FPGAs are chosen, because they can be configured to implement deterministic hardware guaranteeing specific latency and throughput values.

FPGA platforms also have their downsides. In most cases, they are not flexible (if not reprogrammed) and can only handle one single task. The design of an FPGA platform is often very costly. Usually, experienced Hardware Description Language (HDL) programmers, a complete special Printed Circuit Board (PCB) design and a lot of design-, qualification and test time are required.

A system composed of commercial, off-the-shelf (COTS) components is therefore favourable. A standard PC consisting of a motherboard, CPU, storage- and I/O devices is such a system. This choice has several advantages compared to an FPGA platform.

In the first place, it reduces the design time, because the hardware is already qualified. The reduction in design time reduces costs and reduces the time to market (TTM). This is a significant advantage for companies like Prodrive B.V. where a short TTM is essential. When products are sold in low volumes, product design is a relatively large part of the total costs. Reducing this cost has a major impact on the total costs. Furthermore, programming the devices is faster and cheaper. Programming a CPU is very easy, there are a lot of easy to use design tools and well-known high-level programming languages can be used. Configuring an FPGA on the contrary, requires extended knowledge of the platform, experience with complex tools and expertise. In addition, the components in a standard PC are inexpensive, because they are mass produced and there is much competition amongst manufacturers. They are available in different types and prices and are all compatible because of the wide compliance to well defined standards. All-in-all, the product cost per unit reduces and hence the market price decreases and/or the total profit increases.

1.2. **Problem statement**

Computer systems based on general purpose CPUs, in contrast to FPGA platforms, are not deterministic at all at a high level of abstraction. They are so complex that their behaviour is deterministic only at a highly detailed level. Therefore, it is hard to give timing guarantees when using CPU-based computer systems. The time-predictability in particular, is very important if (hard) real-time systems are to be designed. Currently, typically 80% of the design process consists of verification (e.g. by simulation). Reducing these costs has a major impact on the total design costs. This research investigates the possibilities to use a CPU-based computer system for time-critical applications.
1.2.1. Time-predictability

Real-time requirements impose timing constraints on a system [But04]. These requirements can additionally be divided into soft- and hard real-time requirements depending on the consequences if their deadlines are missed.

To verify whether a certain application running on a certain hardware platform meets its timing requirements, its worst-case execution time (WCET) should be known. Finding the WCET is easy if the worst case input to the application and state of the hardware would be known. Unfortunately, this is not the case in most situations. Therefore, the WCET should be predicted.

![Figure 1.1. Example of worst and best case execution times and their estimates [WEE+08]](image)

Figure 1.1 shows an example overview of execution times. Predicting the WCET by performing measurements is in most cases not helpful as Figure 1.1 shows. The reason is that the observed execution time is, strictly speaking, no more than a minimum of the real WCET. The running times of an application may be very irregular, so that the observed WCET is not even a useful indication of the real WCET.

An important issue when predicting the WCET is to not underestimate it. This could possibly result in missing a deadline. Greatly overestimating the WCET on the other hand is not very useful, because it results in an unnecessary and unusable low utilisation of the computer system.

For soft real-time systems, not only the worst case execution time is important, but also the average case execution time.

1.2.2. Influences on running time on five levels

The running time of an application is influenced by many factors. These are grouped into the following levels:

1. Processor core
2. Memory hierarchy
3. System architecture
4. Operating system
5. Application input data

These levels also define the outline for the remainder of this document.

1.3. Goal

The goal of this master project is to describe how the described hardware and software components interact at the described levels and how they influence the running time of an application. These influences can force a maximum throughput constraint, a minimum latency constraint and/or a (non-)constant delay.

These insights can then be used to predict the running time of an application on a future computer system. In addition, the results may give advice to programmers on how to minimise the running time of their application.
1.3.1. Subgoals

Six subgoals are defined to solve the main goal.

Subgoal 1) The processor core level
The properties of the processor may have an impact on the running time of an application running on it. A processor consists of microarchitectural components which influence the time-predictability of applications.
Describe the influences in an abstract formula to gain insight at this level.

Subgoal 2) The memory hierarchy level
The way how a memory hierarchy is organised and how it tries to provide minimal average memory access latency influences the time-predictability of applications.
Describe the influences in an abstract formula to gain insight at this level.

Subgoal 3) The system architecture level
The layout and connections between the main components in a computer system influence the time-predictability of applications running on it.
Describe the influences in an abstract formula to gain insight at this level.

Subgoal 4) The operating system level
The thread scheduler of the operating system in combination with other applications may have an impact on the running time of an application.
Describe the influences in an abstract formula to gain insight at this level.

Subgoal 5) The application level
The running time of an application is not only determined by it’s source code, but also by it’s input data.
Describe the influences in an abstract formula to gain insight at this level.

Subgoal 6) Use-case
The resulting formulas are verified using a use-case. In addition, their functional effectiveness is shown.

Combining the solutions of these subgoals will give the solution to the main goal.

1.4. Approach

In order to solve the subgoals, certain steps have to be carried out. The investigations of the levels as described by the subgoals are executed by performing literature studies. In addition to the theoretical background, practical experiments are carried out. The results from each literature study are then compared and related to the results from the experiments. Finally, the results are formulated in abstract formulas. These formulas are then evaluated using a use-case.

1.5. Related work

Sebestyen and Marfievici [SM09] presented similar real-time predictability issues. Their idea is to use thread affinity and co-scheduling to reduce the WCET of tasks. Chapter 6 also demonstrates and concludes that thread affinity has an important influence on the WCET of tasks.

Gheorghita et al. [GSBC05] use automatic scenario detection to improve the WCET estimation. The source code is analysed and divided into a set of scenarios. Each scenario represents the “mode” the application is in for a certain type of, manually selected, input data. An important requirement is that the set of scenarios must cover all possible input data. The WCETs of all scenarios are then estimated. The WCET of the entire application is calculated as the maximum of all these WCETs. A subsequent paper [GPH+09] improves this idea by dynamically detecting scenarios and changing system parameters during runtime.

Puschner [Pus05] suggests using WCET-oriented programming and single-path conversion for real-time systems. These techniques increase the BCET and try to reduce the WCET thereby reducing jitter (WCET-BCET). Intuitively, using sing-path conversion will increase the WCET. But in practice, it is possible that more instructions can be executed in parallel and less pipeline stalls occur, hence the WCET may even decrease. Chapter 2 demonstrates and
concludes that reducing the number of (difficult-to-predict-) branches reduces the branch misprediction impact and reduces jitter. Chapter 3 shows that data regularity is important to reduce cache misses too.

Paolieri et al. [PQC+09] propose a multi-core architecture where a request from a hard real-time task to a shared resource can only be delayed by a bounded amount of time. Furthermore, a ‘WCET-computation mode’ is available which can be used to estimate WCETs of tasks where each request to a shared resource is delayed by the maximum amount of time. It is then shown that this architecture can be used to safely and accurately predict WCETs.

This research investigates a complete, realistic computer system instead of only a single detail of it while assuming ideal conditions.

1.6. Definitions and Abbreviations

1.6.1. Definitions

- Names and symbols for prefixes for binary multiples according to [IEC60027-2]
- Letters for variables used in formulas
  - T: Time
  - N: Number
  - F: Factor
  - P: Chance
  - X(t): Time dependent variable
  - X(i): Input dependent variable
### 1.6.2. Abbreviations

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Meaning</th>
<th>Abbr.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
<td>ITLB</td>
<td>Instruction Translation Lookaside Buffer</td>
</tr>
<tr>
<td>AMD</td>
<td>Advanced Micro Devices, Inc.</td>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>AVX</td>
<td>Advanced Vector Extensions</td>
<td>LGA</td>
<td>Land Grid Array</td>
</tr>
<tr>
<td>BCET</td>
<td>Best-Case Execution Time</td>
<td>Lxy</td>
<td>Cache level $x$, function $y$: $D = \text{Data}$, $I = \text{Instruction}$, $U = \text{Unified} = D + I$</td>
</tr>
<tr>
<td>BMC</td>
<td>Baseboard Management Controller</td>
<td>MMX</td>
<td>MultiMedia eXtensions</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computer</td>
<td>NOP</td>
<td>No OPeration opcode</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal–Oxide–Semiconductor</td>
<td>opcod e</td>
<td>operation code</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial, Off-The-Shelf</td>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>CPI</td>
<td>Cycles Per Instruction</td>
<td>PCI</td>
<td>Peripheral Component Interconnect</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
<td>PCIe</td>
<td>Peripheral Component Interconnect Express</td>
</tr>
<tr>
<td>CPUID</td>
<td>CPU IDentification</td>
<td>QPI</td>
<td>QuickPath Interconnect</td>
</tr>
<tr>
<td>DDR</td>
<td>Double Data Rate</td>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>DiMM</td>
<td>Dual In-line Memory Module</td>
<td>RAW</td>
<td>Read After Write</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>DTLB</td>
<td>Data Translation Lookaside Buffer</td>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>EBS</td>
<td>Event Based Sampling</td>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>ESI</td>
<td>Enterprise Southbridge Interface</td>
<td>SPEC</td>
<td>Standard Performance Evaluation Corporation</td>
</tr>
<tr>
<td>FMA</td>
<td>Fused Multiply-Add</td>
<td>SSEx</td>
<td>Streaming SIMD Extensions version $x$</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
<td>TLB</td>
<td>Translation Lookaside Buffer</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
<td>TTM</td>
<td>Time To Market</td>
</tr>
<tr>
<td>HDD</td>
<td>Hard Disk Drive</td>
<td>VBR</td>
<td>Variable Bit Rate</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
<td>VLC</td>
<td>Variable-Length Coding</td>
</tr>
<tr>
<td>ICH</td>
<td>I/O Controller Hub</td>
<td>WAR</td>
<td>Write After Read</td>
</tr>
<tr>
<td>IOH</td>
<td>I/O Hub</td>
<td>WAW</td>
<td>Write After Write</td>
</tr>
<tr>
<td>IRQ</td>
<td>Interrupt ReQuest</td>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
2. Processor core

2.1. Introduction

This chapter describes the heart of a computer system, the processor core, in detail. Figure 2.1 shows the location of the processor core, in the heart of the system. Modern processors are very complex devices consisting of millions of transistors forming an extensive electrical circuit with many states; hence, completely describing its time-predictability is impractical. This chapter starts with an historical overview of general purpose CPUs. The third paragraph describes a modern processor pipeline. Subsequently, problems with pipelines are addressed and several improvements are shown. Finally, measurements are described and conclusions are drawn.

2.2. Historical overview of CPUs

Table 2.1 shows a list of well-known processors designed by AMD and Intel, which support the x86 (also known as IA-32) instruction set. The right side of the table shows the support of ISA extensions.
### Table 2.1. Historical overview of CPUs [FH01]

<table>
<thead>
<tr>
<th>Year</th>
<th>Comp.</th>
<th>µProcessor</th>
<th>L. (nm)</th>
<th>µArchitecture</th>
<th>x86</th>
<th>x87</th>
<th>MMX</th>
<th>3DNow!</th>
<th>SSE</th>
<th>x86-64</th>
<th>SSE2</th>
<th>SSE3</th>
<th>SSSE3</th>
<th>SSE4</th>
<th>AES</th>
<th>AVX</th>
<th>SSE5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1985</td>
<td>Intel</td>
<td>80386</td>
<td>1500</td>
<td>i386</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1989</td>
<td>Intel</td>
<td>i486DX</td>
<td>1000</td>
<td>i486</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1993</td>
<td>AMD</td>
<td>Am486</td>
<td>700</td>
<td>i486</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1993</td>
<td>Intel</td>
<td>Pentium</td>
<td>800</td>
<td>P5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1995</td>
<td>AMD</td>
<td>Am5x86</td>
<td>350</td>
<td>i486</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1995</td>
<td>Intel</td>
<td>Pentium Pro</td>
<td>350</td>
<td>P6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1996</td>
<td>AMD</td>
<td>K5</td>
<td>350</td>
<td>K5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1997</td>
<td>Intel</td>
<td>Pentium II</td>
<td>350</td>
<td>P6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1997</td>
<td>AMD</td>
<td>K6</td>
<td>350</td>
<td>K6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>AMD</td>
<td>K6-2</td>
<td>250</td>
<td>K6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>AMD</td>
<td>K6-III</td>
<td>250</td>
<td>K6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>AMD</td>
<td>Athlon</td>
<td>250</td>
<td>K7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>Intel</td>
<td>Pentium III</td>
<td>250</td>
<td>P6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>AMD</td>
<td>Duron</td>
<td>180</td>
<td>K7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>Intel</td>
<td>Pentium 4</td>
<td>180</td>
<td>NetBurst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>AMD</td>
<td>Athlon XP</td>
<td>130</td>
<td>K7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>AMD</td>
<td>Opteron</td>
<td>130</td>
<td>K8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>AMD</td>
<td>Athlon 64</td>
<td>130</td>
<td>K8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>AMD</td>
<td>Sempron</td>
<td>90</td>
<td>K7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>AMD</td>
<td>Athlon 64 X2</td>
<td>90</td>
<td>K9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td>Intel</td>
<td>Core</td>
<td>65</td>
<td>P6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td>Intel</td>
<td>Core 2 Duo</td>
<td>65</td>
<td>Core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>AMD</td>
<td>Phenom</td>
<td>65</td>
<td>K10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>AMD</td>
<td>Phenom II</td>
<td>45</td>
<td>K10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>Intel</td>
<td>Atom</td>
<td>45</td>
<td>Atom</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>Intel</td>
<td>Core i7</td>
<td>45</td>
<td>Nehalem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>AMD</td>
<td>Athlon II</td>
<td>45</td>
<td>K10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td>Intel</td>
<td>&quot;Westmere&quot;</td>
<td>32</td>
<td>Nehalem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>Intel</td>
<td>&quot;Sandy Bridge&quot;</td>
<td>32</td>
<td>Sandy Bridge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>AMD</td>
<td>&quot;Bulldozer&quot;</td>
<td>32</td>
<td>Bulldozer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>Intel</td>
<td>&quot;Ivy Bridge&quot;</td>
<td>22</td>
<td>Sandy Bridge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td>Intel</td>
<td>&quot;Haswell&quot;</td>
<td>22</td>
<td>Haswell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>Intel</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2015</td>
<td>Intel</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2.2 shows the major improvements of ISA extensions as referred to in Table 2.1.

Table 2.2. ISA extensions [Dav09]

<table>
<thead>
<tr>
<th>ISA extension</th>
<th>Year</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>1978</td>
<td>Initial instruction set</td>
</tr>
<tr>
<td>x87</td>
<td>1980</td>
<td>Mathematical functions</td>
</tr>
<tr>
<td>IA-32</td>
<td>1985</td>
<td>Address 4GiB memory; 32-bit registers</td>
</tr>
<tr>
<td>MMX</td>
<td>1997</td>
<td>8x 64-bit registers; Packed integer data types</td>
</tr>
<tr>
<td>3DNow!</td>
<td>1998</td>
<td>Floating point data types</td>
</tr>
<tr>
<td>SSE</td>
<td>1999</td>
<td>8x 128-bit registers; Single precision vectors; Streaming operations</td>
</tr>
<tr>
<td>SSE2</td>
<td>2000</td>
<td>Double precision vectors; MMX instructions on 128-bit; Cache control</td>
</tr>
<tr>
<td>x86-64</td>
<td>2003</td>
<td>16x 128-bit registers</td>
</tr>
<tr>
<td>SSE3</td>
<td>2004</td>
<td>Horizontal add/sub; Multi-threading; Complex arithmetic</td>
</tr>
<tr>
<td>SSSE3</td>
<td>2006</td>
<td>Horizontal add/sub with saturation; Decode</td>
</tr>
<tr>
<td>SSE4</td>
<td>2006</td>
<td>Dot product; Immediate rounding; Max/min; Sums of absolute differences; CRC32; XML accelerator</td>
</tr>
<tr>
<td>AES</td>
<td>2008</td>
<td>Encryption and decryption</td>
</tr>
<tr>
<td>AVX</td>
<td>2008</td>
<td>16x 256-bit registers; 3-operand instructions; Enhanced data rearrangement</td>
</tr>
<tr>
<td>SSE5: XOP, CVT16 &amp; FMA</td>
<td>2009</td>
<td>Fused multiply-accumulate for: dot product, matrix multiplication, polynomial evaluation, Newton's method</td>
</tr>
</tbody>
</table>

Table 2.1 and Table 2.2 show that processors made during the last decades support more and more ISA extensions. These extensions primarily implement increasingly wider vector instructions. This is possible due to exponentially decreasing process technologies. A side effect of the availability of all these instructions is that processors have become very complex and their performance is very state-dependent. Therefore, it has become almost impossible to predict the timing behaviour of an application running on such a processor.

2.3. Processor pipelines

Most processor cores consist of a pipeline to implement the previously described functionality and execute (sub-)instructions in parallel at a high speed.

Figure 2.2. Classic RISC pipeline [PH97]

Figure 2.2 shows a classic RISC pipeline which was the basis for the first CPU implementations.
Figure 2.3. Nehalem Microarchitecture Pipeline Functionality [Dav09]

Figure 2.3 shows the (CISC) pipeline of one of the most modern CPUs currently available. There are still a lot of similarities to the classic RISC pipeline. The pipeline can still be split into the three major parts: instruction decode, execute and memory access.

2.4. Hazards in pipelined processors

Using pipelines to speed up the execution as described in the previous paragraph introduces problems called hazards. Occurrence of a hazard means that a, beforehand unknown, number of NOPs has to be inserted into the pipeline. This has a negative impact on the time-predictability. The remainder of this paragraph describes potential hazards in detail.

2.4.1. Data hazards

The following problem may occur in a pipeline.
a) Read After Write (RAW)
   If the result from a previous calculation is required for the current instruction, it has to wait for the result.
   The following hazards may occur in a concurrent execution environment.
b) Write After Read (WAR)
   The result of a calculation should not be stored in a register if previous instructions still require the old value in that register.
c) Write After Write (WAW)
   The result of a calculation should not be stored in a register if previous instructions will write their results to that same register.

2.4.2. Control hazards

There are two types of control hazards: branch hazards and exceptions & interrupts.
Branch hazards occur when the branch direction of a branch is not yet known. The next instruction can not be started until the branch direction is calculated.
An exception is a synchronous event generated by the processor itself when it detects that one or more predefined conditions are not met while executing an instruction [Inte1A32] e.g. divide by zero. Exceptions occur when program errors are detected by the processor, exceptions are generated by software or exceptions are generated by machine-checks.
An interrupt is an asynchronous event normally triggered by an I/O device. The processor is interrupted by external hardware interrupts and software-generated interrupts. In multiprocessor systems, processors can also send interrupts to one another.
Both internal/external errors/requests disrupt the normal flow and require context switches and/or pipeline stalls.
The task for the hardware is to [PH97]

1. Stop the offending instruction in midstream
2. Complete all instructions before the offending instruction
3. Remove all later instructions
4. Save the cause of the exception
5. Jump to a predefined address

The task for the OS is to

1. Find the cause of the exception
2. For an undefined function, hardware failure or overflow
   a) Kill the program
   b) Show an error message
3. For an I/O device request or OS service call
   a) Save the program state
   b) Perform requested task
   c) Restore program state
   d) Resume program execution

2.4.3. Structural hazards

Structural hazards occur when a part of the processor (e.g. ALU) is required by several instructions simultaneously. In this case, the instruction which finds the part occupied, has to wait for it to become free again.

2.5. Improvements

Giving compilers the responsibility to solve the mentioned hazards is unrealistic, because these dependencies occur too often [PH97]. Therefore, several improvements that have been implemented in modern processors try to eliminate, or at least reduce the impact of, the hazards described in the previous paragraph. This paragraph describes improvements which can be found in most modern processors.
2.5.1. Out-of-Order execution

As can be seen in the pipeline in Figure 2.3, there are two components called scheduler and retirement unit around the execution units. The scheduler buffers the incoming instructions and dispatches them to the execution units when they are available. These instructions are then executed out-of-order. After execution, they are reorganized in the retirement unit. This unit handles the retirement of the executions so that the program is still correctly executed. This improvement reduces the effect of structural hazards by utilizing the execution units more efficiently.

2.5.2. Simultaneous Multithreading

Modern processors may implement N-way simultaneous multithreading. In practice, only 2-way simultaneous multithreading is implemented or no multithreading is implemented at all in current models. This method allows two threads to run on the same core. They share all the parts in the pipeline except the registers which are doubled. This increases the utilization of the execution units when there are enough threads available to run. The result is that the sum of all the running times of all the threads is reduced. This improvement also reduces the effect of structural hazards.

2.5.3. Loop stream decoder

Several modern processors split the x86 assembly instructions internally into so-called micro-instructions. This decoding step takes time. The instructions inside a loop only need to be decoded once, even if a small loop of instructions is executed several times. This reduces the utilization of the decoder, may reduce the power consumption and may improve the execution speed.

2.5.4. Branch prediction

The branch hazards described in subparagraph 2.4.2 may have a large negative influence on the running time if the pipeline has to be stalled each time a branch needs to be resolved. A frequently used improvement is to use a branch predictor. This unit predicts the outcome of the branch so that the processor can continue filling its pipeline with instructions from the predicted address. If the prediction appeared to be wrong, the speculated instructions are flushed. This scheme is advantageous if more than 50% of the predictions are correct. Nowadays, success rates of ±90% are achieved [Tan06]. This high success rate is mainly caused by the fact that most branches are part of loops and hence are relatively easy to predict [KMG01].

2.6. Measurements

2.6.1. Setup

It is expected that the branch predictor has a major impact on the running time of an application. Additionally, other properties are hard to measure. Therefore, the effects of the branch predictor are examined in detail. These measurements are performed on two computer systems to investigate possible differences. The details of the investigated computer systems can be found in Appendix A. These measurements are performed to gain insight into the effect of the implemented branch predictors. For that reason, the program listed in paragraph B.4 is implemented. It runs a loop for many times while each time in the loop, a branch is taken depending on the random number generator. In this way, the random number generator simulates the effect of (un-)predictable input data.

2.6.2. Results

Figure 2.4 and Figure 2.5 show the running times of the measurement program run on two processors as red dots. The striped black line approximates these measurements with a normal distribution. The blue straight line is a first order approximation used in the prediction formula.
Figure 2.4. Effect of branch prediction on Computer system 1 (Intel Core 2 Duo E8500 3.16GHz)

Figure 2.5. Effect of branch prediction on Computer system 3 (Intel Xeon X5560 3.06 GHz)

The effect of the branch predictors is clearly visible and is similar on both computer systems. The running time decreases if it is easier to predict the branch, i.e., the branch probabilities
approach 0% or 100%. The running time is at its maximum when it is hardest to predict the branch outcome, i.e. at a 50% chance.

2.7. Conclusion
To model the running time of an application executing on certain hardware, it is useless to investigate each instruction separately. The model would become too detailed to be useful. Therefore certain abstractions have to be made. Abstracting means approximating in this case. The running time of an application can be split into three parts: the time to execute arithmetic instructions, control instructions and extra instruction because of a branch misprediction. The extra instructions that are executed because of branch mispredictions depend on the input of the application. Summarised

\[ T_{CPU}(i) = T_{arithmetic\_total} + T_{control\_total} + T_{branch\_misprediction}(i) \]

The time it takes to execute the arithmetic and control instructions depends on the number of instructions of that type and the average time it takes to execute such an instruction. This may be described in detail by

\[ T_{arithmetic\_total} = \frac{P_{arithmetic}}{N_{instructions}} \cdot T_{arithmetic\_instruction} \]

and

\[ T_{control\_total} = \frac{P_{control}}{N_{instructions}} \cdot T_{control\_instruction} \]

The chances that a certain instruction is of a certain type depend on the application. For the benchmarks described by the SPEC organisation [SPEC] in the benchmark set SPECint92, the values are [PH97]

\[ P_{arithmetic} = 37\% \]
\[ P_{control} = 23\% \]

The extra number of instructions that are executed because of branch mispredictions depends on the chance that branches are taken. Figure 2.4 and Figure 2.5 show first order approximations. The following formula is formulated to describe the extra time.

\[ T_{branch\_misprediction}(i) = (1 - |1 - 2 \cdot P_{branch\_taken}(i)|) \cdot T_{max\_misprediction} \]

The value for \( P_{branch\_taken}(i) \) represents the input dependent chance that a branch is taken while the value for \( T_{max\_misprediction} \) represents the maximum extra time that is needed when all branches are incorrectly predicted.
3. Memory hierarchy

3.1. Introduction

Figure 3.1. Computer system overview - Memory hierarchy highlighted

This chapter describes all memories in a computer system from Level 1 cache to main memory as shown in Figure 3.1. In addition, details of caches and their impact on average latencies are considered. Furthermore, experiments are performed to gain practical insight into the influences on the running time. Finally, the theory and experiments are summarised and time-prediction formulas are formulated.

3.2. Hierarchy

Figure 3.2 shows rough estimates of the sizes and latencies in a typical memory hierarchy.
3.3. Cache details

All programs and data may be (multi-level) cached, which results in a non-deterministic delay of memory requests and hence program execution time.

3.3.1. Locality of reference/principle of locality

A cache is effective because access patterns in typical computer applications have locality of reference [Tan06]. This principle can be split in two concepts, being temporal locality and spatial locality. Temporal locality means that specific data and/or resources are reused within relatively small time durations. Spatial locality means that data elements are frequently read/written from/to relatively close storage location. It can be split into three concepts.

1. Sequential locality
   Occurs when data elements are arranged and accessed linearly, e.g., traversing the elements in a one-dimensional array.

2. Equidistant locality
   Consider a loop accessing locations in an equidistant pattern, i.e. the path in the spatial-temporal coordinate space is a dotted line. In this case, a simple linear function can predict which location will be accessed in the near future.

3. Branch locality
   If there are only a few possible alternatives for the potential part of the path in the spatial-temporal coordinate space.

The effect of the principle on timing may be large. It means that data accesses to data patterns that are easy to predict will be much faster than accesses to irregular data patterns.

3.3.2. Write policy

Caches write their data according to a write policy. These policies can be split into two main categories, being
• Write-back/-behind
  A read miss in a write-back cache (which requires a block to be replaced by another) may require two memory accesses to service: one to retrieve the needed datum and one to write the replaced data from the cache to the store (e.g. main memory) if that datum needed to be written
• Write-through
  Every write to the cache causes a synchronous write to the backing store

The choice for a certain write policy will have an effect on timing. A write-back policy means fast writes, but potentially slow reads. A write-through policy means medium speed writes and reads.

3.3.3. Cache types
Caches appear in several types. Instruction cache can speed up the fetching of executable instructions. Data cache can speed up data fetch and store instructions. A special type of cache, the Translation Lookaside Buffer (TLB) is used to speed up virtual-to-physical address translation for executable instructions and/or data. These components are called ITLB and DTLB for instruction or data TLB respectively.

3.3.4. (Non-)Blocking caches
Blocking caches wait until the main memory request completes in case of a cache miss. Non-blocking caches continue serving following requests while the main memory request completes in case of a cache miss. Non-blocking caches handle requests faster, but special precautions may be taken to avoid potential coherency problems.

3.3.5. Replacement policy
The replacement policy decides where a certain block of data from main memory can be stored in the cache [PH97][Tan06]. This may be fully associative, direct mapped or a version in between.

a) Fully associative
  The data can be stored at any location in the cache
  • Pro
    Lowest miss rate compared to direct mapped/set associative caches
  • Con
    Slowest/largest hit times compared to direct mapped and set associative caches, because finding the data in the cache takes a lot of chip area and time, since all entries have to be checked

b) N-way set associative
  The data can only be stored at N certain locations in the cache
  • Pro
    Lower miss rate than direct mapped caches
    Shorter hit times than fully associative caches
  • Con
    Higher miss rate than fully associative caches
    Slower/larger hit times than direct mapped caches

c) Direct mapped (1-way associative)
  The data can only be stored at a certain location in the cache
  • Pro
    Shortest hit times compared to fully- or set associative caches, because finding out if the data is cached is easy
  • Con
    Highest miss rate compared to associative caches

Choosing the replacement policy is a typical trade-off between hit rate and hit time. Both influence the average memory access time.
### 3.3.6. Cache hits and misses

The ratio of the number of hits to the total number of memory request is called hit rate. The miss rate is defined similarly.

The average memory access time with three levels of caches can be calculated using the following formula [HP06].

\[
T_{\text{memory accesses}}(i,t) = N_{\text{memory accesses}} \cdot P_{L1 \text{ hit}}(i,t) \cdot T_{L1 \text{ access}} + T_{\text{higher levels}}(i,t)
\]

\[
= P_{L1 \text{ hit}}(i,t) \cdot T_{L1 \text{ access}} + (1 - P_{L1 \text{ hit}}(i,t)) \cdot P_{L2 \text{ hit}}(i,t) \cdot T_{L2 \text{ access}} + (1 - P_{L1 \text{ hit}}(i,t)) \cdot (1 - P_{L2 \text{ hit}}(i,t)) \cdot T_{\text{main access}}
\]

\[
= P_{L1 \text{ hit}}(i,t) \cdot T_{L1 \text{ access}} + (1 - P_{L1 \text{ hit}}(i,t)) \cdot (1 - P_{L2 \text{ hit}}(i,t)) \cdot T_{\text{main access}} + (1 - P_{L1 \text{ hit}}(i,t)) \cdot (1 - P_{L2 \text{ hit}}(i,t)) \cdot T_{\text{main access}}
\]

\[
= \sum_{n=0}^{k} \left( \prod_{m=0}^{n-1} (1 - P_{Lm \text{ hit}}(i,t)) \right) \cdot P_{Ln \text{ hit}}(i,t) \cdot T_{Ln \text{ access}}
\]

where

\[
P_{L0 \text{ hit}} = 0 \text{ and } P_{Lk \text{ hit}} = 1 \text{ for } k \geq 2
\]

Here, \( P_{Lm \text{ hit}} \) is the hit rate at cache level \( m \), \( T_{Lm \text{ hit}} \) is the access time of cache level \( m \), and \( T_{Lk \text{ hit}} \) is the access time of the main memory i.e. \( T_{\text{main access}} \).

#### Table 3.1. Examples of cache effect

<table>
<thead>
<tr>
<th></th>
<th>High hit rates</th>
<th>Low hit rates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Level</strong></td>
<td><strong>Hit rate</strong></td>
<td><strong>Access time</strong></td>
</tr>
<tr>
<td>L1</td>
<td>95%</td>
<td>4.0 clock cycles</td>
</tr>
<tr>
<td>L2</td>
<td>95%</td>
<td>10.0 clock cycles</td>
</tr>
<tr>
<td>L3</td>
<td>95%</td>
<td>40.0 clock cycles</td>
</tr>
<tr>
<td>Mem</td>
<td>100.0 clock cycles</td>
<td>100.0 clock cycles</td>
</tr>
<tr>
<td>Minimum</td>
<td>4.0 clock cycles</td>
<td>4.0 clock cycles</td>
</tr>
<tr>
<td>Average</td>
<td>4.4 clock cycles</td>
<td>50.7 clock cycles</td>
</tr>
<tr>
<td>Maximum</td>
<td>100.0 clock cycles</td>
<td>100.0 clock cycles</td>
</tr>
</tbody>
</table>

Table 3.1 shows examples of the effect of the cache levels. Without caches, the minimum, average and maximum latency to main memory would be 100 clock cycles. The caches lower the minimum and average latency to only 4 and 4.4 clock cycles at high hit rates of 95%. On the other hand, if the hit rates decrease to 25% (due to non-regular memory access), the average latency increases to 50.7 clock cycles.

### 3.4. Measurements

#### 3.4.1. Setup

The goal of these measurements is to gain insight into the effects of the caches on the running time of an application. The theory describes a large dependence on the hit rate. When reading from memory, this hit rate depends on the information that is requested and on the information that is in the cache. Therefore, test programs are written that read a certain amount of memory in random order. The size of the memory block that is read is varied. The test programs for these measurements are written directly in x86 assembly to bypass the effects of a compiler. The code is supplied in paragraph B.3. The programs were analysed using a profiler called Intel VTune [IntelVTune]. This program can read out the performance counters on the CPU. This method is called Event Based Sampling (EBS) [DMM*04]. The measurements where performed on Computer system 2.
3.4.2. Results

Figure 3.3 shows the effect of the increasing size of the read region. Above 32 KiB, i.e. the size of level 1 cache, the level 1 cache miss rate increases. The same effect occurs with level 2 cache which has a size of 6 MiB. The values do not increase much further, because the miss rate is depicted. This is the ratio of cache misses compared to the number of memory requests.

![Graph showing L1 Data Cache Miss Rate and L2 Cache Miss Rate](image)

Figure 3.3. L1 Data Cache Miss Rate and L2 Cache Miss Rate (see Appendix C for details)

The effect of the increasing level 1 cache miss rate is shown in Figure 3.4. The impact of the cache misses increases too. This effect reduces at higher memory sizes, because the relative impact of the level 1 cache miss decreases. The effect of the increasing level 2 cache miss rate gives rise to an increasing bus utilisation. It is clear that the bus forms the bottleneck in memory intensive situations.

![Graph showing L1 Data Cache Miss Performance Impact, Bus Utilization and TLB miss penalty](image)

Figure 3.4. L1 Data Cache Miss Performance Impact, Bus Utilization and TLB miss penalty (see Appendix C for details)

The result of all previous effects is shown in Figure 3.5 where the clocks per instruction are shown. This figure defines, in combination with the number of instructions, the total running time of the application. Both in increase in the clocks per instruction and an increase in the number of instructions increase the total running time. The effects of the level 1 cache miss ratio, the level 2 cache miss ratio and the overloading of the bus are clearly identifiable in the graph.

The result of all previous effects is shown in Figure 3.5 where the clocks per instruction are shown. This figure defines, in combination with the number of instructions, the total running time
of the application. Both in increase in the clocks per instruction and an increase in the number of instructions increase the total running time. The effects of the level 1 cache miss ratio, the level 2 cache miss ratio and the overloading of the bus are clearly identifiable in the graph.

![Clocks per Instructions - CPI](image)

**Figure 3.5. Clocks per Instruction – CPI (see Appendix C for details)**

The clocks per instruction ranges from 1.78 to 109 meaning a possible speed difference of a factor of 61.

### 3.5. Conclusion

The total running time of an application may be split into CPU time and memory access time.

$$T_{\text{total}\_\text{running}}(i,t) = T_{\text{CPU}} + T_{\text{memory\_accesses}}(i,t)$$

The theory clearly describes the reasons for the use of caches. The influence of the caches on timing is described by

$$T_{\text{memory\_accesses}}(i,t) = \sum_{n=0}^{k} \left( \prod_{m=0}^{n-1} (1 - P_{L_{m\_hit}}(i,t)) \right) \cdot P_{L_{n\_hit}}(i,t) \cdot T_{L_{n}}\_\text{access}$$

where

$$P_{L_{0}\_hit} = 0 \& P_{L_{k}\_hit} = 1 \& k \geq 2$$

In practice, this means that the simpler the memory access pattern, the higher the hit rate and hence the faster the memory accesses. The measurements showed that the effects may be very large, a speed difference factor of 61 is measured. This result is also very useful in practice. It is important to know the cache sizes when programming a (streaming) data processing unit. The choice to split the data into smaller parts that fit into the caches may lead to a large increase in processing speed.
4. System architecture

4.1. Introduction

Figure 4.1. Computer system overview - System architecture highlighted

This chapter describes the way the main hardware components are connected in a computer system. These include main memory, I/O communication and the processor(s). The type, number and relative location of these parts may have a large effect on the running time. Computer systems are built in various ways. Several system architectures are depicted in Figure 4.2.
Over the last years, a trend can be seen from one processing core towards more processing cores. Sometimes in one package, i.e. multi-core CPUs, sometimes in multiple packages, i.e. multi-CPU systems. In addition, functionalities such as memory controller and I/O communication are moved from specific chips to the CPU. In 2003, AMD’s Athlon 64 CPUs were the first to include a memory controller; Intel followed in 2008 with its Core i7 CPUs. Integrating these functionalities into the CPU has the benefit of lower latency communications because of the shorter, hence less capacitive, copper tracks.

4.2. Bottlenecks

The diagrams in Figure 4.2 show several system architectures. The cores inside the processors and the connections in these diagrams may be used by several threads to communicate their data. The total running time of an application may be split into CPU time and memory access time as shown in paragraph 3.5. This means that at every moment, the application is limited by both the calculation capacity of the CPU(s) and the memory access latencies of the memory hierarchy. In practice however, frequently only one of both has a major impact on the running time and is called the bottleneck. Gaining insight into the bottleneck for (a certain part of) an application is important, because this might help in solving timing issues.

4.3. Measurements

4.3.1. Setup

The following measurements are performed to gain insight into the influences of multiple interacting threads on the running time. A two-CPU computer system is chosen where each CPU has four cores.

The test program starts a number of threads in a number of cores. Each threads reads a large memory block several times. The resulting running times are measured.
4.3.2. Results
Several memory throughput measurements were performed on Computer system 3 (see paragraph A.3). A block diagram of the implementation of the system architecture is shown in Figure 4.3.

![Figure 4.3. Intel Server Board S5520HC Functional Block Diagram [IntelS5520HC]
Figure 4.4 shows the measurement results from running the test program shown in paragraph B.2. The program runs a given number of threads each reading 128 MiB of memory for 200 times.

\[
\text{Throughput} = \text{BusClockFrequency} \cdot 2(\text{DDR}) \cdot 64(\text{BitWidth}) / 8(\text{bits / Byte})
\]

\[
= 667[\text{MHz}] \cdot 2 \cdot 64[b] / 8[b / B] = 10.7E6[B / s] = 9.9[\text{GiB / s}]
\]

The dark red line in the graph shows a trend line for the running time of the program when 1 to 8 threads are executed on a single CPU. The running time increases linearly with the number of threads. An average throughput of 8.7 GiB/s is achieved. This is 88% of the maximum 9.9 GiB/s which is shown in pink. The maximum throughput per channel is calculated as

The dark blue line in the graph shows the running time of the program when the threads are executed on two CPUs. The running time again increases linearly with the number of threads, but the speed is doubled compared to the single-CPU version. This result is achieved, because the computer system has two independent connections between a CPU and a memory bank. On average, a throughput of 18.4 GiB/s is achieved, 92% of the maximum 19.9 GiB/s shown in light blue. This is only achievable if the application can be split into two or more threads. In general the application should be split into the number of memory-CPU connections to achieve the maximum memory bandwidth.

The theoretical maxima are not achieved because of disturbances by interrupts and other software that is part of the operating system and because of communication overhead. Chapter 3 describes how caches are used to improve average throughput considerably. In real-time systems, not only the average case is important, but also the deviation from this average case. The yellow vertical bars in Figure 4.4 show the actual measured running times. The standard deviations from these measurements are shown in Figure 4.5.
The dark red dots show a linear increase in standard deviation as approximated by the pink trend line. This increase in standard deviation comes from the fact that the scheduler needs to schedule all threads on the same core. Furthermore, all the threads use the same caches and hence thread A may fill the caches with unwanted data for thread B.

The dark blue dots with the light blue trend line show a relatively constant line. This is explained by the facts that the thread scheduler is almost inactive, only operating system threads need some execution time. In addition, the threads influence each other only slightly, because each core has its own levels 1 and 2 cache, only level 3 cache is shared between the four cores on a CPU.

### 4.4. Conclusion

The theory describes possible bottlenecks, an application is either limited by calculation power or by CPU-memory bandwidth. The theory and the measurement results show that a solution to these problems is to split the application into multiple threads and either run these threads on multiple processor cores to gain calculation or run the threads on multiple CPUs to gain CPU-memory bandwidth. Splitting an application into multiple threads is not always possible, this adds a constraint to the possibilities.

This conclusion may be formulated as

**Constraint by maximum calculation power:**

\[
T_{\text{private \_runtime}} = \frac{T_{\text{total \_runtime}}}{\min(N_{\text{max \_threads}}, N_{\text{cores}})}
\]

**Constraint by maximum CPU-memory bandwidth:**

\[
T_{\text{private \_runtime}} = \frac{T_{\text{total \_runtime}}}{\min(N_{\text{max \_threads}}, N_{\text{CPU \_memory \_connections}})}
\]
5. Operating system

5.1. Introduction

This chapter describes the influence of the operating system on the running time of an application running on that operating system. The operating system offers an interface between the hardware and the applications.

5.2. Priorities

Most advanced OSs can run multiple threads which can interrupt each other, at timeslots which are unknown beforehand, and run for an unknown number of cycles. These interruptions of unbounded length are unacceptable for real-time systems. In most cases, this is also unwanted for interactive applications. Therefore, a scheduling algorithm is implemented. This scheduler may run high priority tasks more frequently and/or suspend low priority tasks.

5.3. Interrupts

Tasks are not only directly disrupted by other tasks through scheduling, but also indirectly by interrupts. These interrupts may be generated by external I/O devices or indirectly by (other) tasks. Paragraph 2.4.2 considers interrupts in more detail.

5.4. Measurements

5.4.1. Setup

Measurements have been performed in order to investigate the influence of these interruptions caused by other tasks and interrupts. A simple program which writes a large amount of data to main memory is run multiple times (see paragraph B.1). This program is run multiple times to gain insight into the time-dependent influences.

5.4.2. Results

The resulting running times are plotted in Figure 5.2.
The graphs show a median\(^1\) running time of 11.950 s, with a maximum of 12.015 s (+0.542\%) and a minimum of 11.916 s (-0.285\%) and a standard deviation of 0.015 s. The estimate of the running time (for this input data) is chosen as the smallest observed value. Using this method for WCET estimation of the application alone has the advantage that the WCET is never underestimated (assuming worst-case input data if any). This means that the WCET \((T_{\text{private_running_time}})\) is estimated to be 11.916 s. The observed jitter is the result of interrupting other tasks being part of the operating system and hard- and software interrupts. These are then attributed the extra mean running time of 0.034 s (11.950-11.916) with a standard deviation of 0.015 s \((T_{\text{other_running_time}})\).

5.5. Conclusion

Both the theory and the measurements show the running time of an application is time-dependent when the application is run on a computer system which runs multiple threads (including operating system threads) and handles interrupts. Hence, the running time may be split into two parts. One part being the application itself and the other being the additional running time.

\[
T_{\text{regular}}(t) = T_{\text{private_runtime}} + T_{\text{other_runtime}}(t)
\]

The other running time itself may also be split into two parts. One part being the effect of the thread scheduler which halts the application if higher-priority tasks need to be executed. The other part being the running time needed to handle hard- and software interrupts. I.e.

\[
T_{\text{other_runtime}}(t) = T_{\text{higher_priority}}(t) + T_{\text{interrupts}}(t)
\]

\(^1\)The median is chosen instead of the mean value because the median is less sensitive to outliers which may be caused by measurement errors
6. Application input data

6.1. Introduction

Figure 6.1. Computer system overview - Application highlighted

This chapter describes the effect of input data on the running time of higher-level software running on top of an operating system. Figure 6.1 shows an abstraction of the entire computer system in which the application level is highlighted. Measurement results can be found in Chapter 7.

6.2. (Ir)regular data processing

6.2.1. Sources

More and more data processing is irregular. One reason for this is that information is nowadays transferred in compressed form to reduce transfer time and storage usage. Current compression techniques, such as the VBR MP3 audio codec, JPEG image codec, XviD and DivX video codecs and general purpose ZIP compression, all use Variable-Length Coding (VLC) techniques to compress the transferred data. This means that the number of bits to represent an audio/video frame fluctuates. The result is that the encoder and the decoder, as well as the connection between them, have to deal with irregular data (-processing) and possibly varying amounts of data.

Another source of irregularity may be found during content analysis of signals with a high variance. For example, the analysis of a video sequence from a busy road crossing, evidently leads to irregular data processing as well.

6.2.2. Consequences

The regularity of the input data of an application can have a high impact on the running time of that application. The reason is that the execution path may be dependent on the input data. Certain data patterns may then result in very short or on the other hand very long running times. In addition, non-linear memory reads may have a negative effect on cache performance. This effect is described in detail in Chapter 3.

Additionally, the branch predictor inside the CPU might perform poorly when the outcomes of the branches change frequently. Measurements concerning this effect are analysed in Chapter 2.
Measurements that demonstrate the total effect of input data dependence are shown in paragraph 7.2.5.

### 6.3. Conclusion

The total running time of an application with certain input data is the sum of the minimum running time added to the running time needed to execute the longer code path and to compensate for the incorrect branch predictions. In a formula

$$
T_{\text{total}} (i) = T_{\text{instruction}} \cdot (N_{\text{instructions}_{\text{min}}} + N_{\text{instructions}_{\text{irregular}}}(i))
$$

The total effect of the input data dependence may be summarised to

$$
T_{\text{total}} (i) = T_{\text{regular}} \cdot F_{\text{irregular}} (i)
$$

where

$$
F_{\text{regular}} (i) \geq 1
$$

Where $F_{\text{regular}}$ defines an input-dependent factor that compensates for the extra running time. It is important for real-time systems that their maximum $F_{\text{regular}}$ is known in order to be able to meet timing requirements in case of different input data.
7. Use-case

7.1. Introduction

The computer system that is examined is part of a design by Prodrive B.V.. The goal of this subsystem is to compress a large amount of image data using the JPEG2000 image compression algorithm [JPEG2000] under strict timing requirements. To use the formulas described in the previous chapters for prediction, a set of parameters is needed. These parameters define the properties of the algorithm and the computer system. In order to determine these parameters, measurements have to be carried out.

7.2. Extracting parameters

The previous chapters showed that the running time is influenced in several ways. Discerning all these influences from the observed running time demands several requirements from the application. The application should be configurable in several ways to extract the required parameters. The parameters are extracted and the requirements are described in the following subparagraphs.

7.2.1. Processor core

The application should be highly alterable and closely monitored to find the effect of the branch predictor. Therefore, a small application is more suitable than a large application. A larger application, like this use-case, requires a more black-box-like approach. The reason for that is clear from the following formulas.

\[ T_{total}(i) = T_{regular} \cdot F_{irregular}(i) \] (see Chapter 6)
\[ T_{CPU}(i) = T_{arithmetic \_ total} + T_{control \_ total} + T_{branch \_ misprediction}(i) \] (see Chapter 2)

where \( T_{CPU} \) is a part of \( T_{regular} \).

Both \( F_{irregular} \) and \( T_{branch \_ misprediction} \) are input dependent. A regular input, i.e. both easily branch-predictable and short code paths, results in both a low \( F_{irregular} \) and a low \( T_{branch \_ misprediction} \). On the other hand, a highly irregular input will result in both a high \( F_{irregular} \) and a high \( T_{branch \_ misprediction} \). Therefore, the effect of the branch predictor is only visible when it is explicitly possible to change the prediction property without changing the code path or the other way around. These effects are both combined, i.e. added, in the more black-box-like approach that is taken here. \( F_{irregular} \) is used to describe the effect of both influences and is analysed in subparagraph 7.2.5.

7.2.2. Memory hierarchy

The effect of the caches on the running time of an application may be large. But the effect can only be analysed when the application has a property that influences the cache usage. Most applications, especially streaming audio- or video processing applications, have some sort of property that defines a block size. This block size defines how large a block is that is processed before the next. This size then largely influences how much cache is needed to improve the performance of the application. Measuring the running time of the application for several different values of this block size gives valuable insight into the cache usage of the application. This information can in turn be used to optimise the application for a given computer system with given cache sizes. The application from this use-case could not be tuned in this way.

7.2.3. System architecture

The following formulas were formulated considering the system architecture.

Constraint by maximum calculation capacity:
\[ T_{private \_ runtime} = T_{total \_ runtime} / \min(N_{max \_ threads}, N_{cores}) \]
Constraint by maximum memory capacity:
\[ T_{private \_ runtime} = T_{total \_ runtime} / \min(N_{max \_ threads}, N_{CPU \_ memory \_ connections}) \]
In order to find an indication of $N_{\text{max\_threads}}$, first it must be decided if the application is constrained by calculation capacity or by memory capacity. This is determined by executing the application in a variable number of threads and measuring the running time.

**Figure 7.1. Thread-dependent running times**

Figure 7.1 shows the results of the measurements. The minimum running time is calculated using the running time for 1 thread divided by the number of threads. Examination of the measurement results shows that a performance gain is achieved when the number of threads is increased up to 8. Hence, it can be concluded that this application is constrained by calculation capacity. Increasing the number of threads beyond 8 does not increase the performance. Since the number of cores in Computer system 3 is 8, the number of cores ($N_{\text{cores}}$) is the limiting factor. In addition, it can be concluded that

\[ N_{\text{max\_threads}} \geq 8 \]

### 7.2.4. The operating system

The influences of other applications and interrupts are studied at the operating system level. The following formula was used to describe these influences.

\[ T_{\text{regular}}(t) = T_{\text{private\_runtime}} + T_{\text{other\_runtime}}(t) \]

In order to gain insight into both parts of the running time, the application is executed multiple times. The results are shown in Figure 7.2.
It is clear from the graph that the running time is time-dependent as shown in the formula. The private running time $T_{\text{private\_running\_time}}$ is estimated as the minimum observed running time, i.e. 16.346 s. The other running time $T_{\text{other\_running\_time}}$ is modelled by a normal distribution with an average of 66 ms and standard deviation of 76 ms. This means that the running time of the application is only slightly influenced by other applications and interrupts, only $0.066 / 16.346 = 0.40\%$ on average. This is clear from the axis of Figure 7.2 too.

### 7.2.5. Application input data

The following formula was formulated considering the application input data.

$$T_{\text{total\_input\_data}}(i) = T_{\text{regular\_input\_data}}(i) \cdot F_{\text{irregular\_input\_data}}(i)$$

The measurement is performed to gain insight into the input dependent factor $F_{\text{irregular\_input\_data}}$. This is achieved by running the algorithm with different input sets. In this case, the best- and worst case inputs are known for this algorithm. Additionally, a representative input set is available which represents the average input.
Figure 7.3. Input-dependent running times histogram

Figure 7.3 shows the effect of different input data on the running time. A simple single-coloured image, which is considered the best case input, has a running time of only 9.14 s. A representative test image has a running time of 16.41 s. A white noise image, which is considered the worst case input for this application, has a running time of 26.44 s. The conclusion is that the input dependent factor $F_{\text{irregular}}$ can be as high as $26.44 / 9.14 = 2.9$, while the average $F_{\text{regular}} = 16.41 / 9.14 = 1.8$. Additionally, it can be concluded that $T_{\text{regular}} = 9.14$ s.

7.3. Conclusion

The formulated formulas have been successfully applied to this use-case. This has gained significant benefits. It is concluded that the application is limited by calculation capacity. Furthermore, it is concluded that the application utilises the computer system most effectively if it is run in 8 threads. Additionally, it is found that the running time of the application is only slightly influenced by other applications and interrupts. Moreover, measurements showed that the algorithm is highly input-dependent. The analysis also showed how the running time of the application is affected by several system-specific properties and how the running time can be decreased if necessary.
8. Conclusion

8.1. Introduction

The influences of the five levels on the running time have been investigated and modelled. After that, the modelling formulas have been applied to a use-case. This chapter summarises the conclusions from the previous chapters and describes the modelling formulas. Finally, a general conclusion is drawn.

8.2. Processor core

8.2.1. Summary

The theory shows that the running time of an application depends on the type of instructions that are executed and the order in which they are executed. The pipeline has the goal to complete as many instructions per cycle as possible. Hazards reduce the number of instructions that are executed per cycle while the described improvements try to eliminate and/or reduce the impact of these hazards.

8.2.2. Formulas

To model the running time of an application executing on certain hardware, it is useless to investigate each instruction separately. Therefore certain abstractions have to be made. Abstracting means approximating in this case. A rough approximation of the running time of an application is given by:

\[ T_{CPU}(i) = T_{arithmetic \_total} + T_{control \_total} + T_{branch \_misprediction}(i) \]

where

\[ T_{arithmetic \_total} = P_{arithmetic} \cdot N_{instructions} \cdot T_{arithmetic \_instruction} \]

and

\[ T_{control \_total} = P_{control} \cdot N_{instructions} \cdot T_{control \_instruction} \]

and

\[ T_{branch \_misprediction}(i) = (1 - P_{branch \_taken}(i)) \cdot T_{max \_misprediction} \]

Table 8.1 describes the variables that are used at this level.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{CPU}(i) )</td>
<td>(input dependent) Time spent executing instructions on the CPU</td>
</tr>
<tr>
<td>( T_{arithmetic _total} )</td>
<td>Total time to execute all arithmetic instructions</td>
</tr>
<tr>
<td>( T_{arithmetic _instruction} )</td>
<td>Average time to execute an arithmetic instruction</td>
</tr>
<tr>
<td>( P_{arithmetic} )</td>
<td>Chance that a given instruction is an arithmetic instruction</td>
</tr>
<tr>
<td>( T_{control _total} )</td>
<td>Total time to execute all control instructions</td>
</tr>
<tr>
<td>( T_{control _instruction} )</td>
<td>Average time to execute a control instruction</td>
</tr>
<tr>
<td>( P_{control} )</td>
<td>Chance that a given instruction is a control instruction</td>
</tr>
<tr>
<td>( T_{branch _misprediction}(i) )</td>
<td>(input dependent) Additional time required to handle incorrectly predicted branches</td>
</tr>
<tr>
<td>( P_{branch _taken} )</td>
<td>Chance that a branch is taken</td>
</tr>
<tr>
<td>( T_{max _misprediction} )</td>
<td>Additional time required when all branches are incorrectly predicted</td>
</tr>
<tr>
<td>( N_{instructions} )</td>
<td>Total number of instructions in the program</td>
</tr>
</tbody>
</table>
8.3. Memory hierarchy

8.3.1. Summary

The time it takes for an application to finish can be split into two parts. In one part the CPU is executing instructions, in the other part it is waiting for memory accesses. The time it takes for the memory accesses to complete depends on the number of memory accesses. This is clear from the theory and is clearly shown in the results of the measurements. The time it takes for a memory access to finish is also dependent on the previous memory accesses. These previous memory accesses can be executed by the application itself, by another process running on the same core or by another process running on another core sharing the same cache level. The previous memory accesses executed by the application itself make the actual hit rate input dependent. The other factors also influence the actual hit rate for that cache level, so it is time dependent too.

8.3.2. Formulas

\[ T_{\text{total}_{\text{runtime}}}(i, t) = T_{\text{CPU}} + T_{\text{memory accesses}}(i, t) \]
\[ T_{\text{memory accesses}}(i, t) = N_{\text{memory accesses}} \cdot P_{L1}\text{hit}(i, t) \cdot T_{L1\text{access}} + T_{\text{higher levels}}(i, t) \]
\[ = P_{L1\text{hit}}(i, t) \cdot T_{L1\text{access}} + (1 - P_{L1\text{hit}}(i, t)) \cdot P_{L2\text{hit}}(i, t) \cdot T_{L2\text{access}} + (1 - P_{L1\text{hit}}(i, t)) \cdot (1 - P_{L2\text{hit}}(i, t)) \cdot T_{\text{main access}} \]
\[ = P_{L1\text{hit}}(i, t) \cdot T_{L1\text{access}} + (1 - P_{L1\text{hit}}(i, t)) \cdot (1 - P_{L2\text{hit}}(i, t)) \cdot T_{L2\text{access}} + (1 - P_{L2\text{hit}}(i, t)) \cdot T_{\text{main access}} \]
\[ = \sum_{n=0}^{k} \left( \prod_{m=0}^{n-1} (1 - P_{Lm\text{hit}}(i, t)) \right) \cdot P_{Ln}\text{hit}(i, t) \cdot T_{Ln\text{access}} \]

where

\[ P_{k_0\text{hit}} = 0 \quad \text{and} \quad P_{k_k\text{hit}} = 1 \quad \text{and} \quad k \geq 2 \]
\[ N_{\text{memory accesses}} = N_{\text{instructions}} \cdot P_{\text{memory access}} \]

SPECint92 [SPEC] had a value of [PH97]:

\[ P_{\text{memory access}} = 40\% \]

Table 8.2 describes the variables that are used at this level.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{\text{total}_{\text{runtime}}}(i, t) )</td>
<td>(input- and time dependent) Time it takes to execute the program in a single thread</td>
</tr>
<tr>
<td>( T_{\text{CPU}} )</td>
<td>Time spent executing instructions on the CPU</td>
</tr>
<tr>
<td>( T_{\text{memory accesses}}(i, t) )</td>
<td>(input- and time dependent) Time spent executing read and write memory accesses</td>
</tr>
<tr>
<td>( N_{\text{memory accesses}} )</td>
<td>Number of memory accesses in the program</td>
</tr>
<tr>
<td>( P_{Lx\text{hit}}(i, t) )</td>
<td>(input- and time dependent) Chance of a hit at cache level x</td>
</tr>
<tr>
<td>( T_{Lx\text{access}} )</td>
<td>Time it takes to read/write data from/to cache level x</td>
</tr>
<tr>
<td>( N_{\text{instructions}} )</td>
<td>Total number of instructions in the program</td>
</tr>
<tr>
<td>( P_{\text{memory access}} )</td>
<td>Chance that an instruction is a memory instruction</td>
</tr>
</tbody>
</table>

8.4. System architecture

8.4.1. Summary

The running time of an application running on a computer system can be limited by either computational power or CPU-memory bandwidth. It is shown that the system architecture has an important influence on these limitations. An application which is limited by CPU-memory
bandwidth was shown during the measurements in Chapter 4. The use-case in Chapter 7 describes a computationally limited application.

8.4.2. Formulas

Constraint by maximum calculation power:
\[ T_{\text{private runtime}} = \frac{T_{\text{total runtime}}}{\min(N_{\text{max threads}}, N_{\text{cores}})} \]

Constraint by maximum CPU-memory bandwidth:
\[ T_{\text{private runtime}} = \frac{T_{\text{total runtime}}}{\min(N_{\text{max threads}}, N_{\text{CPU memory connections}})} \]

Table 8.3 describes the variables that are used at this level.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{\text{private runtime}} )</td>
<td>Time it takes to execute the program</td>
</tr>
<tr>
<td>( T_{\text{total runtime}} )</td>
<td>Time it takes to execute the program in a single thread</td>
</tr>
<tr>
<td>( N_{\text{max threads}} )</td>
<td>Maximum number of threads into which the program can be split</td>
</tr>
<tr>
<td>( N_{\text{cores}} )</td>
<td>Number of processing cores in the computer system</td>
</tr>
<tr>
<td>( N_{\text{CPU memory connections}} )</td>
<td>Number of parallel processor-memory connections in the computer system</td>
</tr>
</tbody>
</table>

8.5. Operating system

8.5.1. Summary

The running time of an application depends on its priority relative to the priorities of other applications running on the same system. This property may depend on the moment the application is run, thus it is time-dependent. The running time is also affected by interrupts occurring during runtime. This property is again time-dependent. Both times are added to the running time of the application and form the actual running time.

8.5.2. Formulas

\[ T_{\text{regular}}(t) = T_{\text{private runtime}} + T_{\text{other runtime}}(t) \]
\[ T_{\text{other runtime}}(t) = T_{\text{higher priority}}(t) + T_{\text{interrupts}}(t) \]

Table 8.4 describes the variables that are used at this level.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{\text{regular}}(t) )</td>
<td>(time dependent) Observed total time required to execute the program</td>
</tr>
<tr>
<td>( T_{\text{private runtime}} )</td>
<td>Time it takes to execute the program</td>
</tr>
<tr>
<td>( T_{\text{other runtime}}(t) )</td>
<td>(time dependent) Time it takes to run other programs and handle IRQs</td>
</tr>
<tr>
<td>( T_{\text{higher priority}}(t) )</td>
<td>(time dependent) Time it takes to run other programs with higher priority</td>
</tr>
<tr>
<td>( T_{\text{interrupts}}(t) )</td>
<td>(time dependent) Time it takes to handle interrupt requests (IRQs)</td>
</tr>
</tbody>
</table>

8.6. Application input data

8.6.1. Summary

The running time of an application may depend on the regularity of the input data. Applications have a minimum running time \( T_{\text{min}} \). Certain input data patterns may increase the length of the code path, hence increasing the number of instructions. The regularity of the input data may increase the actual running time by a factor \( F_{\text{irregular}}(i) \) which is input dependent.

8.6.2. Formulas

The total running time of an application with certain input data can be described as
\[ T_{\text{total}}(i) = T_{\text{instruction}} \cdot (N_{\text{instructions min}} + N_{\text{instructions irregular}}(i)) \]
which can be summarised to
\[ T_{\text{total}}(i) = T_{\text{regular}} \cdot F_{\text{irregular}}(i) \]
where
\[ F_{\text{regular}}(i) \geq 1 \]

Table 8.5 describes the variables that are used at this level.

| \( T_{\text{total}}(i) \) & (input dependent) Total running time of the application \\
| \( T_{\text{instruction}} \) & Average time it takes to execute an instruction \\
| \( N_{\text{instructions min}} \) & Minimum number of instructions executed by the program \\
| \( N_{\text{instructions irregular}(i)} \) & (input dependent) Additional number of instructions executed by the program \\
| \( T_{\text{regular}} \) & Minimum time required to execute the program \\
| \( F_{\text{irregular}(i)} \) & (input dependent) Factor (\( \geq 1 \)) to compensate for additional running time |

8.7. General conclusion

This research has shown that, although computer systems may be very complex, it is possible to model the influences on the running time of the applications running on the system. This is done by means of a divide-and-conquer approach. The influences on the running time are split into five levels. These levels are then investigated one by one. Both theory and measurements are used to formulate abstract formulas.

Combining the results of these investigations provides valuable insights into the timing in a complete computer system. These insights and abstract formulas help to reduce the design time of future real-time computer system designs. This is achievable since a large part of the design time was spent on verification and simulation is no longer required since systems can be created that are correct by design. This allows the design of more complex devices while still meeting the more demanding timing requirements imposed by future system requirements. All in all, tomorrow’s needs for smarter devices will be fulfilled.
9. Future work
This work may be expanded in the future. This chapter gives possible expansion directions.

9.1. CPU/memory frequency scaling
The computer systems that were available during this research did not support adjustments of the CPU and/or memory frequencies. Performing measurements on computer systems that do support these adjustments may gain more detailed insight into processor and memory influences. For example, running a test program multiple times while each time adjusting the processor clock frequency will result in different running times. This might then be used to select the lowest clock frequency of the processor which in turn leads to possible optimisations. The processor may be swapped for a cheaper version. In addition, the power requirements for the total system will go down resulting in a more environment friendly product and longer battery life.

9.2. Graphical user interface
A graphical user interface (GUI) may be designed. This will ease the use of the formulas and will give even more insight to the user. In addition to a GUI, a database may be created from which the user may select computer system components. The program may even aid in making choices for certain components based on their properties stored in the database.
Appendix A Details of investigated computer systems

A.1 Computer system 1

Computer system 1 consists of one dual core Core microarchitecture CPU running at 3.16 GHz connected to a northbridge through a Front Side Bus (FSB) and to 2 x 2 GiB DDR2 memory in dual channel and runs Windows XP.

![Computer system 1 block diagram](image)

**Figure 9.1. Computer system 1 block diagram**

A.1.1 Processor - Intel® Core™2 Duo CPU E8500 @ 3.16GHz.

Information is available at [IntelE8500]. The following information has been extracted using the Intel(R) Processor Identification Utility [IntelPIU].

Intel(R) Processor Identification Utility
Version: 4.20.20090811
Time Stamp: 2010/01/21 10:15:10
Operating System: 5.1-2600-Service Pack 3
Number of processors in system: 1
Current processor: #1
Active cores per processor: 2
Disabled cores per processor: 0
Processor Name: Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHz
Type: 0
Family: 6
Model: 17
Stepping: A
Revision: A07
Maximum CPUID Level: D
L1 Instruction Cache: 2 x 32 KB
L1 Data Cache: 2 x 32 KB
L2 Cache: 6 MB
Packaging: LGA775
Enhanced Intel SpeedStep(R) Technology: Yes
MMX(TM): Yes
Intel(R) SSE: Yes
Intel(R) SSE2: Yes
Intel(R) SSE3: Yes
Intel(R) SSE4: Yes
Enhanced Halt State: Yes
Execute Disable Bit: Yes
Intel(R) Hyper-Threading Technology: No
Intel(R) 64 Architecture: Yes
Intel(R) Virtualization Technology: Yes
Expected Processor Frequency: 3.16 GHz
Reported Processor Frequency: 3.16 GHz
Expected System Bus Frequency: 1333 MHz
Reported System Bus Frequency: 1333 MHz

The following information has been extracted using CPU-Z 1.54 [CPUID].

Number of cores 2 (max 2)
Number of threads 2 (max 2)
Name Intel Core 2 Duo E8500
Codename Wolfdale
Specification Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHz
CPUID 6.7A
Extended CPUID 6.17
Core Stepping E0
Technology 45 nm
Core Speed 1995.0 MHz
Multiplier x FSB 6.0 x 332.5 MHz
Rated Bus speed 1330.0 MHz
Stock frequency 3166 MHz

Instructions sets MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, EM64T, VT-x
L1 Data cache 2 x 32 KBytes, 8-way set associative, 64-byte line size
L1 Instruction cache 2 x 32 KBytes, 8-way set associative, 64-byte line size
L2 cache 6144 KBytes, 24-way set associative, 64-byte line size
FID/VID Control yes
FID range 6.0x - 9.5x
Max VID 1.263 V

A.1.2 Chipset

Northbridge Intel Q45/Q43 rev. 03
Southbridge Intel ID3A1A rev. 02
Graphic Interface PCI-Express
PCI-E Link Width x16
PCI-E Max Link Width x16
Memory Type DDR2
Memory Size 4096 MBytes
Channels Dual, (Symmetric)
Memory Frequency 399.0 MHz (5:6)
CAS# latency (CL) 6.0
RAS# to CAS# delay (tRCD) 6
RAS# Precharge (tRP) 6
Cycle Time (tRAS) 18
Row Refresh Cycle Time (tRFC) 52
Command Rate (CR) 2T
MCHBAR I/O Base address 0xFD000000
MCHBAR I/O Size 4096

A.1.3 Memory - 2 x 2 GiB

SMBus address 0x50
Memory type DDR2
Module format Regular UDIMM
Manufacturer (ID) Elpida (7F7FFE000000000)
Size 2048 MBytes
Max bandwidth PC2-6400 (400 MHz)
Part number EBE21UE8AFFA-8G-F
Serial number 2B7EB960
Manufacturing date Week 25/Year 09
Number of banks 2
Data width 64 bits
Correction None
Nominal Voltage 1.80 Volts
EPP no
XMP no

A.1.4 Software - Windows XP Professional

Windows Version Microsoft Windows XP Professional Service Pack 3 (Build 2600)
DirectX Version 9.0c

A.2 Computer system 2

Computer system 2 consists of two quad core Core microarchitecture CPUs running at 2.33 GHz connected to a northbridge through Front Side Buses (FSBs) and runs Windows XP.
A.2.1 Processor - Intel® Xeon® CPU L5410 @ 2.33GHz.

Information is available at [IntelL5410]. The following information has been extracted using the Intel(R) Processor Identification Utility [IntelPIU].

Intel(R) Processor Identification Utility
Version: 4.20.20090811
Time Stamp: 2010/01/21 10:19:46
Operating System: 5.1-2600-Service Pack 3
Number of processors in system: 2
Current processor: #1
Active cores per processor: 4
Disabled cores per processor: 0
Processor Name: Intel(R) Xeon(R) CPU L5410 @ 2.33GHz
Type: 0
Family: 6
Model: 17
Stepping: 6
Revision: 60C
Maximum CPUID Level: A
L1 Instruction Cache: 4 x 32 KB
L1 Data Cache: 4 x 32 KB
L2 Cache: 2 x 6 MB
Packaging: LGA771
Enhanced Intel SpeedStep(R) Technology: Yes
MMX(TM): Yes
Intel(R) SSE: Yes
Intel(R) SSE2: Yes
Intel(R) SSE3: Yes
Intel(R) SSE4: Yes
Enhanced Halt State: Yes
Execute Disable Bit: Yes
Intel(R) Hyper-Threading Technology: No
Intel(R) 64 Architecture: Yes
Intel(R) Virtualization Technology: Yes
Expected Processor Frequency: 2.33 GHz
Reported Processor Frequency: 2.33 GHz
Expected System Bus Frequency: 1333 MHz
Reported System Bus Frequency: 1333 MHz

A.3 Computer system 3

Computer system 3 consists of two quad core Nehalem microarchitecture CPUs running at 3.06 GHz connected to each other and a northbridge through QuickPath Interconnect (QPI) and to 2 x 2 GiB DDR3 memory in single channel and runs Windows Server 2008 R2.
A.3.1 Processor - Intel® Xeon® Processor X5560 @ 3.06 GHz.
Information is available at [IntelX5560].

Table 9.1. Cache Parameters of Processors based on Nehalem Microarchitecture [IntelNehalem]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L1D</td>
<td>32 KiB</td>
<td>8</td>
<td>64</td>
<td>4</td>
<td>1</td>
<td>Writeback</td>
</tr>
<tr>
<td>L1I</td>
<td>32 KiB</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>L2U</td>
<td>256 KiB</td>
<td>8</td>
<td>64</td>
<td>10</td>
<td>Varies</td>
<td>Writeback</td>
</tr>
<tr>
<td>L3U</td>
<td>8 MiB</td>
<td>16</td>
<td>64</td>
<td>35-40</td>
<td>Varies</td>
<td>Writeback</td>
</tr>
</tbody>
</table>

The following information has been extracted using the Intel(R) Processor Identification Utility [IntelPIU].
Intel(R) Processor Identification Utility
Version: 4.20.20090811
Time Stamp: 2010/01/04 12:15:51
Operating System: 5.1-2600-Service Pack 2
Number of processors in system: 2
Current processor: #1
Active cores per processor: 4
Disabled cores per processor: 0
Processor Name: Intel(R) Xeon(R) CPU
Type: 0
Family: 6
Model: 1A
Stepping: 4
Revision: D
Maximum CPUID Level: B
L1 Instruction Cache: 4 x 32 KB
L1 Data Cache: 4 x 32 KB
L2 Cache: 4 x 256 KB
L3 Cache: 8 MB
Packaging: LGA1366
Enhanced Intel SpeedStep(R) Technology: Yes
MMX(TM): Yes
Intel(R) SSE: Yes
Intel(R) SSE2: Yes
Intel(R) SSE3: Yes
Intel(R) SSE4: Yes
Enhanced Halt State: Yes
Execute Disable Bit: Yes
Intel(R) Hyper-Threading Technology: Yes
Intel(R) 64 Architecture: Yes
Intel(R) Virtualization Technology: Yes
Reported Processor Frequency: 3.06 GHz
Reported System Bus Frequency: 133 MHz
Reported QuickPath Interconnect Speed: 6.39 GT/s
Reported Integrated Memory Controller Frequency: 1333 MHz

The following information has been extracted using CPU-Z 1.54 [CPUID].

Number of cores 4 (max 8)
Number of threads 4 (max 16)
Name Intel Xeon X5560
Codename Gainestown
Specification Intel(R) Xeon(R) CPU X5560 @ 2.80GHz (Engineering Sample)
Package (platform ID) Socket 1366 LGA (0x0)
CPUID 6.A.4
Extended CPUID 6.1A
Core Stepping C0/C1
Technology 45 nm
Core Speed 3195.4 MHz
Multiplier x FSB 24.0 x 133.1 MHz
Rated Bus speed 3195.4 MHz
Stock frequency 2800 MHz
Instructions sets MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x
L1 Data cache 4 x 32 KBytes, 8-way set associative, 64-byte line size
L1 Instruction cache 4 x 32 KBytes, 4-way set associative, 64-byte line size
L2 cache 4 x 256 KBytes, 8-way set associative, 64-byte line size
L3 cache 8 MBytes, 16-way set associative, 64-byte line size
FID/VID Control yes
Turbo Mode supported, enabled
Max non-turbo ratio 21x
Max turbo ratio 24x
Max efficiency ratio 12x
TDP Limit 95 Watts
TDC Limit 85 Amps
Core TDP 85 Watts
Uncore TDP 10 Watts
Power @ 12x 25 Watts
Power @ 13x 30 Watts
Power @ 14x 35 Watts
Power @ 15x 40 Watts
Power @ 16x 47 Watts
Power @ 17x 55 Watts
Power @ 18x 63 Watts
Power @ 19x 72 Watts
Power @ 20x 83 Watts
Power @ 21x 95 Watts
Max bus number 255
Attached device PCI device at bus 255, device 2, function 1
Attached device PCI device at bus 255, device 3, function 4

A.3.2 Chipset

Northbridge Intel 5520 rev. 13
Southbridge Intel 82801JR (ICH10R) rev. 00
Graphic Interface PCI-Express
PCI-E Link Width x1
PCI-E Max Link Width x1
Memory Type DDR3
Memory Size 4036 MBytes
Channels Single
Memory Frequency 665.7 MHz (2:10)
CAS# latency (CL) 9.0
RAS# to CAS# delay (tRCD) 9
RAS# Precharge (tRP) 9
Cycle Time (tRAS) 24
Row Refresh Cycle Time (tRFC) 74
Command Rate (CR) 1T
Uncore Frequency 2662.9 MHz
A.3.3 Memory - 2 x 2 GiB

SM Bus address 0x50
Memory type DDR3
Module format RDIMM
Manufacturer (ID) Micron Technology (2C00000000000000)
Size 2048 MBytes
Max bandwidth PC3-10700H (667 MHz)
Part number 18JSF25672PDY1G4DY
Serial number FA1812EA
Manufacturing date Week 37/Year 08
Number of banks 8
Nominal Voltage 1.50 Volts
EPP no
XMP no

A.3.4 Software - Windows Server 2008 R2

Windows Version Microsoft Windows Server 2008 R2 (Build 7600)
DirectX Version 11.0
Appendix B Source code of test programs

B.1 Used at operating system level

```c
#include <windows.h>
#include <stdio.h>
#include <process.h>

const unsigned int MEMSIZE=1<<29;//512MiB
const unsigned int ARRAYSIZE=MEMSIZE/sizeof(int);

unsigned __stdcall Thread1(void* pArguments)
{
    unsigned int* array1;
    unsigned int b,c,d=0;
    array1=(unsigned int*)malloc(MEMSIZE);
    if(array1==NULL)
        printf("Failed to malloc array1 of size %d\n",MEMSIZE);
    for (c=0; c<100; c++)
    {
        for (b=0; b<ARRAYSIZE-c; b++)
        {
            array1[b]=0;
        }
    }
    array1[0]=d;
    _endthreadex(0);
    return 0;
}

int main()
{
    LARGE_INTEGER start,stop,proc_freq;//Declare timing variables
    HANDLE handle;//Declare handle
    QueryPerformanceFrequency(&proc_freq);//Get processor frequency
    handle=(HANDLE)_beginthreadex(NULL,0,&Thread1,NULL,CREATE_SUSPENDED,NULL);//Create threads
    SetThreadAffinityMask(handle,128);//Set thread affinity masks
    QueryPerformanceCounter(&start);//Start counter
    ResumeThread(handle);//Start thread
    WaitForSingleObject(handle,INFINITE);
    QueryPerformanceCounter(&stop);//Stop counting
    CloseHandle(handle);//Destroy the thread object
    printf("%.21f\n",(double)(stop.QuadPart-start.QuadPart)/proc_freq.QuadPart);//Output timing
}
```

B.2 Used at system architecture level

```c
#include <windows.h>
#include <stdio.h>
#include <process.h>

const unsigned int MEMSIZE=1<<27;//128MiB
const unsigned int ARRAYSIZE=MEMSIZE/sizeof(int);

unsigned __stdcall Thread1(void* pArguments)
{
    unsigned int* array1;
    unsigned int b,c,d=0;
    array1=(unsigned int*)malloc(MEMSIZE);
    if(array1==NULL)
        printf("Failed to malloc array1 of size %d\n",MEMSIZE);
    for (c=0; c<200; c++)
    {
        for (b=0; b<ARRAYSIZE-c; b++)
        {
            d=array1[b];
        }
    }
    array1[0]=d;
    _endthreadex(0);
    return 0;
}
```
int main()
{
    LARGE_INTEGER start, stop, proc_freq;// Declare timing variables
    HANDLE handles[8];// Declare handles
    QueryPerformanceFrequency(&proc_freq);// Get processor frequency
    handles[0]=(HANDLE)_beginthreadex(NULL,0,&Thread1,NULL,CREATE_SUSPENDED,NULL);// Create threads
    handles[1]=(HANDLE)_beginthreadex(NULL,0,&Thread1,NULL,CREATE_SUSPENDED,NULL);// Create threads
    handles[2]=(HANDLE)_beginthreadex(NULL,0,&Thread1,NULL,CREATE_SUSPENDED,NULL);// Create threads
    handles[3]=(HANDLE)_beginthreadex(NULL,0,&Thread1,NULL,CREATE_SUSPENDED,NULL);// Create threads
    handles[4]=(HANDLE)_beginthreadex(NULL,0,&Thread1,NULL,CREATE_SUSPENDED,NULL);// Create threads
    handles[5]=(HANDLE)_beginthreadex(NULL,0,&Thread1,NULL,CREATE_SUSPENDED,NULL);// Create threads
    handles[6]=(HANDLE)_beginthreadex(NULL,0,&Thread1,NULL,CREATE_SUSPENDED,NULL);// Create threads
    handles[7]=(HANDLE)_beginthreadex(NULL,0,&Thread1,NULL,CREATE_SUSPENDED,NULL);// Create threads
    SetThreadAffinityMask(handles[0],1);// Set thread affinity masks
    SetThreadAffinityMask(handles[1],16);// Set thread affinity masks
    SetThreadAffinityMask(handles[2],2);// Set thread affinity masks
    SetThreadAffinityMask(handles[3],32);// Set thread affinity masks
    SetThreadAffinityMask(handles[4],4);// Set thread affinity masks
    SetThreadAffinityMask(handles[5],64);// Set thread affinity masks
    SetThreadAffinityMask(handles[6],8);// Set thread affinity masks
    SetThreadAffinityMask(handles[7],128);// Set thread affinity masks
    QueryPerformanceCounter(&start);// Start counter
    ResumeThread(handles[0]);// Start threads
    ResumeThread(handles[1]);// Start threads
    ResumeThread(handles[2]);// Start threads
    ResumeThread(handles[3]);// Start threads
    ResumeThread(handles[4]);// Start threads
    ResumeThread(handles[5]);// Start threads
    ResumeThread(handles[6]);// Start threads
    ResumeThread(handles[7]);// Start threads
    WaitForSingleObject(handles[0],INFINITE);// Wait for single object
    WaitForSingleObject(handles[1],INFINITE);// Wait for single object
    WaitForSingleObject(handles[2],INFINITE);// Wait for single object
    WaitForSingleObject(handles[3],INFINITE);// Wait for single object
    WaitForSingleObject(handles[4],INFINITE);// Wait for single object
    WaitForSingleObject(handles[5],INFINITE);// Wait for single object
    WaitForSingleObject(handles[6],INFINITE);// Wait for single object
    WaitForSingleObject(handles[7],INFINITE);// Wait for single object
    QueryPerformanceCounter(&stop);// Stop counting
    CloseHandle(handles[0]);// Destroy the thread objects
    CloseHandle(handles[1]);// Destroy the thread objects
    CloseHandle(handles[2]);// Destroy the thread objects
    CloseHandle(handles[3]);// Destroy the thread objects
    CloseHandle(handles[4]);// Destroy the thread objects
    CloseHandle(handles[5]);// Destroy the thread objects
    CloseHandle(handles[6]);// Destroy the thread objects
    CloseHandle(handles[7]);// Destroy the thread objects
    printf("%.2f
}
B.3 Used at memory hierarchy level

```assembly
.code
start:
mov eax, alloc(1073741824)
mov ecx, 0
loopy:
mov ebx, [eax+911191543]
mov ebx, [eax+343523495]
... (100,000x)
mov ebx, [eax+261645419]
mov ebx, [eax+275857221]
inc ecx
cmp ecx, 8000000
jnz loopy
free eax
exit
end start
```

Begin
Allocate variable number of bytes
For ecx = 0 to BIG_NUMBER (run +/- 10s)
Read random data from array
... (100,000x)
Read random data from array
Next ecx
Free memory
End

B.4 Used at processor core level

```c
#include <windows.h>
#include <stdio.h>
#include <process.h>

int main()
{
LARGE_INTEGER start,stop,proc_freq; //Declare timing variables
unsigned long a,branchpoint;
double b=9999999;
QueryPerformanceFrequency(&proc_freq);//Get processor frequency
for(branchpoint=0; branchpoint<32768; branchpoint=branchpoint+100)
{
    QueryPerformanceCounter(&start);//Start counter
    for (a=0;a<99999999,a++)
    {
        if (rand()<branchpoint)//RAND_MAX=32767
            b=b*-9;
        else
            b=b*9;
    }
    QueryPerformanceCounter(&stop);//Stop counting
    printf("",b);
}
}
```
Appendix C Event based sampling details

C.1 Ratios
The ratio descriptions in this paragraph are available in VTune’s user’s guide and at http://www.sesp.cse.clrc.ac.uk/html/SoftwareTools/vtune/users_guide/mergedProjects/analyzer_ec/mergedProjects/reference_olh/pmm/ratios/

C.1.1 L1 Data Cache Miss Rate
Equation: L1D_REPL / INST_RETIRED.ANY
Definition: High value of this ratio indicates that the code misses the L1 data cache too often and pays the penalty of accessing the L2 cache.

C.1.2 L2 Cache Miss Rate
Equation: L2_LINES_IN.SELF.ANY / INST_RETIRED.ANY
Definition: A high L2 cache miss rate indicates that the running workload has a data set larger than the L2. Some of the data might be evicted without being used. Unless all the required data is brought ahead of time by the hardware prefetcher or software prefetching instructions, bringing data from memory has a significant impact on the performance.

C.1.3 L1 Data Cache Miss Performance Impact
Equation: 8 * L1D_REPL / CPU_CLK_UNHALTED.CORE
Definition: High value of this ratio indicates that the code misses the L1 data cache too often and pays the penalty of accessing the L2 cache.

C.1.4 Bus Utilization
Equation: BUS_TRANS_ANY.ALL_AGENTS * 2 / CPU_CLK_UNHALTED.BUS
Definition: High bus utilization indicates heavy traffic between the processor and the memory. Memory sub-system latency might impact the performance of the program. For compute-intensive applications, high bus utilization generally indicates you should look for opportunities to improve data/code locality. For other types of applications (copying large amounts of data from one memory area to another, for example) it may be desirable to maximize bus utilization.

C.1.5 TLB miss penalty
Equation: PAGE_WALKS.CYCLES / CPU_CLK_UNHALTED.CORE
Definition: High value of this ratio indicates that many cycles are spent on TLB misses. Reducing the number of TLB misses may improve performance.

C.1.6 Clocks per Instruction – CPI
Equation: CPU_CLK_UNHALTED.CORE / INST_RETIRED.ANY
Definition: High CPI indicates that instructions are requiring more cycles to execute than they should. In this case there may be opportunities to modify your code to improve the efficiency with which instructions are executed within the processor. The CPI can get as low as 0.25 cycles per instructions. When the front-end cannot provide instructions and micro-ops at a rate high enough to fill the Reservation Station it is likely that the execution engine is starved waiting for micro-ops to execute and the measured CPI is high. High number of L0 DTLB misses indicates that the data set that the workload uses spans a number of pages that is bigger than the L0 DTLB. The high number of misses is expected to impact workload performance only if the CPI is low - around 0.8. Otherwise it is likely that the L0 DTLB miss cycles are hidden by other latencies.

C.2 Events
The event descriptions in this paragraph are available in VTune’s user’s guide and at http://www.sesp.cse.clrc.ac.uk/html/SoftwareTools/vtune/users_guide/mergedProjects/analyzer_ec/mergedProjects/reference_olh/pmm/events/
C.2.1 L1D_REPL
Definition: Cache lines allocated in the L1 data cache.
Description: This event counts the number of lines brought into the L1 data cache.

C.2.2 INST_RETIRED.ANY
Definition: Instructions retired.
Description: This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.

C.2.3 L2_LINES_IN.SELF.ANY
Definition: L2 cache misses.
Description: This event counts the number of cache lines allocated in the L2 cache. Cache lines are allocated in the L2 cache as a result of requests from the L1 data and instruction caches and the L2 hardware prefetchers to cache lines that are missing in the L2 cache.
This event can count occurrences for this core or both cores. This event can also count demand requests and L2 hardware prefetch requests together or separately.
Counts demand requests and requests from the L2 hardware prefetchers.

C.2.4 CPU_CLK_UNHALTED.CORE
Definition: Core cycles when core is not halted.
Description: This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios.
In mobile systems the core frequency may change from time to time. For this reason this event may have a changing ratio with regards to time. In systems with a constant core frequency, this event can give you a measurement of the elapsed time while the core was not in halt state by dividing the event count by the core frequency.

C.2.5 BUS_TRANS_ANY.ALL_AGENTS
Definition: All bus transactions.
Description: This event counts all bus transactions. This includes:
- Memory transactions
- IO transactions (non memory-mapped)
- Deferred transaction completion
- Other less frequent transactions, such as interrupts

C.2.6 PAGE_WALKS.CYCLES
Definition: Duration of page-walks in core cycles
Description: This event counts the duration of page-walks in core cycles. The paging mode in use typically affects the duration of page walks. Page walk duration divided by number of page walks is the average duration of page-walks. This can hint at whether most of the page-walks are satisfied by the caches or cause an L2 cache miss.
## References

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Title</th>
<th>Authors</th>
<th>Year</th>
<th>Conference/Source</th>
<th>URL</th>
</tr>
</thead>
</table>
[GSBC05] Paper
Automatic Scenario Detection for Improved WCET Estimation
Stefan Valentin Gheorghita, Sander Stuijk, Twan Basten and Henk Corporaal
2005
http://doi.acm.org/10.1145/1065579.1065610

[HP06] Book
Computer architecture
A Quantitative Approach
Fourth Edition
John L. Hennessy and David A. Patterson
2006
Morgan Kaufmann Publishers
ISBN 9780123704900
http://www.biblio.com/isbn/9780123704900.html

[IEC60027-2] Standard
Names and symbols for prefixes for binary multiples
IEC 60027-2 A.2 and ISO/IEC 80000
IEC website:
http://www.iec.ch/zone/si/si_bytes.htm
NIST website:
http://physics.nist.gov/cuu/Units/binary.html

[IntelE5540] Website
Intel® Xeon® Processor E5540 (8M Cache, 2.53 GHz, 5.86 GT/s Intel® QPI)
Intel Corporation
http://ark.intel.com/Product.aspx?id=37104

[IntelE8500] Website
Intel® Core™2 Duo Processor E8500 (6M Cache, 3.16 GHz, 1333 MHz FSB)
Intel Corporation
http://ark.intel.com/Product.aspx?id=33911

[IntellA32] Datasheet
Intel® 64 and IA-32 Architectures Software Developer’s Manual
Intel Corporation
5 volumes
June 2009

[IntelL5410] Website
Intel® Xeon® Processor L5410 (12M Cache, 2.33 GHz, 1333 MHz FSB)
Intel Corporation
http://ark.intel.com/Product.aspx?id=33090

[IntelNehalem] Datasheet
Nehalem Microarchitecture and SSE4.2 Optimization Guide
Rev. 0.35
Reference Number: 401308-002
October 2008
Intel Corporation

[IntelPIU] Website
Intel® Processor Identification Utility - Intel® Processor Identification Utility support
Version: 4.20.20090811
Intel Corporation
http://www.intel.com/support/processors/tools/piu/
[IntelS5520HC] DataSheet
Intel® Server Boards S5520HC and S5500HCV
Technical Product Specification
Intel order number E39529-0010
Revision 1.3
August 2009
Intel Corporation

[IntelVTune] Website
Intel® VTune – Intel® Software network
Intel® VTune™ Performance Analyzer 9.1 Update 2 for Windows
Intel Corporation

[IntelX5560] Website
Intel® Xeon® Processor X5560 (8M Cache, 2.80 GHz, 6.40 GT/s Intel® QPI)
Intel Corporation
http://ark.intel.com/Product.aspx?id=37109

[JPEG2000] Website
Joint Photographic Experts Group (JPEG)
JPEG 2000 is a new image coding system that uses state-of-the-art compression
techniques based on wavelet technology
http://www.jpeg.org/jpeg2000/

[KMG01] Paper
Selective Branch Inversion: Confidence Estimation for Branch Predictors
Artur Klauser, Srilatha Manne and Dirk Grunwald
2001
http://dx.doi.org/10.1023/A:1026436021514

[LAW06] Paper
FPGA Architecture for Real-Time Video Noise Estimation
François-Xavier Lapalme, Aishy Amer and Chunyan Wang
2006
Image Processing, 2006 IEEE International Conference on
http://dx.doi.org/10.1109/ICIP.2006.312918

[PH97] Book
Computer organization & design
David A. Patterson & John L. Hennessy
2nd edition
1997
ISBN 9781558604285
http://www.biblio.com/isbn/9781558604285.html

[PQC’09] Paper
Hardware support for WCET analysis of hard real-time multicore systems
Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat and Mateo Valero
2009
International Symposium on Computer Architecture, Proceedings of the 36th
annual international symposium on Computer architecture
http://doi.acm.org/10.1145/1555754.1555764

[Pus05] Paper
Experiments with WCET-Oriented Programming and the Single-Path Architecture
Peter Puschner
2005
Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE
International Workshop on
http://dx.doi.org/10.1109/WORDS.2005.36
[SM09] Paper
Real-Time Predictability on Multi-Processor and Multi-Core Architectures
Gheorghe Sebestyen, Ramona Marfievici
2009
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5284734

[SPEC] Website
Standard Performance Evaluation Corporation (SPEC)
http://www.spec.org/

[Tan06] Book
Structured computer organization
Andrew S. Tanenbaum
Fifth edition
2006
Prentice Hall
ISBN 9780131485211
http://www.biblio.com/isbn/9780131485211.html

[VA08] Presentation
High Performance IO with Next Generation Intel® Microarchitecture (Nehalem)
Anil Vasudevan and Sunil Ahluwalia
Intel Developer Forum
2008
Intel Corporation
http://www.benchmark.rs/tests/editorial/Nehalem_munich/presentations/NHM_High-Performance_IO.pdf

[WEE+08] Paper
The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools
Reinhard Wilhelm, Jakob Engblom andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat and Per Stenström
2008
ACM Transactions on Embedded Computing Systems (TECS)
http://dx.doi.org/10.1145/1347375.1347389