AALTO UNIVERSITY
School of Science and Technology
Faculty of Electronics, Communications and Automation
Degree programme of Communications Engineering

Jaakko Kotimäki

Measuring system activity on multi-core and multi-processor platforms

Master's Thesis
Espoo, May 3, 2010

Supervisor: Professor Heikki Saikkonen, Aalto University
Instructor: Vesa Hirvisalo, D.Sc., Aalto University

AALTO UNIVERSITY
School of Science and Technology
Faculty of Electronics, Communications and Automation
Degree Programme of Communications Engineering

ABSTRACT OF MASTER'S THESIS

Author: Jaakko Kotimäki

Title of thesis:
Measuring system activity on multi-core and multi-processor platforms

Date: May 3, 2010 **Pages:** ?? + 51

Professorship: Software technology **Code:** T-106

Supervisor: Professor Heikki Saikkonen
Instructor: Vesa Hirvisalo, D.Sc.

The recent emergence of multi-core and multi-processor systems has brought parallel programming back into the spotlight. Despite being a widely studied concept with a few well founded general solution programming models and a lot of specific solutions for certain problems, the behavior and performance of a parallel program is difficult to grasp for even the most experienced programmers.

There are a lot of metrics for measuring operating system activity and performance: CPU time per process, system queue length, number of context switches and so on. However these metrics only give a quantitative view of the activity when the understanding and improving of parallel program performance requires visualization of hardware and software thread interaction.

In this Master's Thesis I present how a kernel event tracer can be applied for measuring the behavior of software and hardware threads in a multi-core and multi-processor systems. The resulting visualization of thread-level interaction can be then used for the analysis of parallel program execution and the identification of possible problematic sections that further benefit the understanding of parallel programming.

Keywords: multi-processor system, event tracing, parallel programming
Language: English

AALTO-YLIOPISTO
Teknillinen korkeakoulu
Elektroniikan, tietoliikenteen ja automaation tiedekunta
Tietoliikennetekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Jaakko Kotimäki

Työn nimi:
Moniydin- ja moniprosessorijärjestelmien aktiivisuuden mittaaminen

Päiväys: 3. toukokuuta 2010 **Sivumäärä:** ?? + 51
Professuuri: Ohjelmistotekniikka **Koodi:** T-106
Työn valvoja: Professori Heikki Saikkonen
Työn ohjaaja: Vesa Hirvisalo, TkT

Rinnakkaisohjelmointi on tullut moniydin- ja moniprosessorijärjestelmien suosion myötä pinnalle. Rinnakkaisohjelmointia on tutkittu kauan ja joitakin perustavanlaatuisia ratkaisuja ohjelmointimalleihin on tarjolla. Niin ikään on olemassa useita tiettyihin erikoistapauksiin keskittyviä ohjelmointimalliratkaisuja. Tästä huolimatta rinnakkaisohjelmien käyttäytymisen ymmärtäminen on kokeneillekin ohjelmoijille vaikeaa. Lisäksi tämänhetkiset rinnakkaisohjelmointityökalut eivät täysin tue rinnakkaisohjelmointia.

Käyttöjärjestelmän toiminnan mittaamiseen on olemassa useita metriikoita, kuten prosessin käyttämä aika prosessorilla tai kontekstin vaihtojen määrä. Nämä metriikat antavat kuitenkin vain kvantitatiivisen näkymän toiminnasta, kun taas rinnakkaisen ohjelman toiminnan ymmärtäminen vaatii käsitystä rauta- ja ohjelmatason säikeiden käyttäytymisestä.

Tässä diplomityössä on mitattu jäljitystyökalulla moniydin- ja moniprosessorijärjestelmiä erilaisilla rinnakkaistetuilla työkuormilla. Tulosten havainnollistuksesta voidaan rinnakkaisohjelman suoritusta analysoida ja tunnistaa ongelmakohtia rauta- ja ohjelmatason säikeiden käyttäytymisessä.

Avainsanat: moniprosessorijärjestelmä, tapahtumajäljitys, rinnakkaisohjelmointi
Kieli: englanti

Acknowledgements

I thank Professor Heikki Saikkonen and my instructor, D.Sc. Vesa Hirvisalo, for taking me under their wing, and offering me excellent guidance and advice during the process. Moreover, I thank my fellow co-workers in the ESG research group without whom the process would have been much harder; thank you Timo Töyry, Juho Äyräväinen, Marjukka Kokkonen, Sami Kiminki and Antti Miettinen.

Most of all, I am grateful to my family, my mother Erika, my father Tuomo, and my siblings Ulrika and Veikko, for support and understanding during the years. Furthermore, I thank Sanna Suoranta and my aunt Johanna for support and advice.

Espoo 3. toukokuuta 2010

Jaakko Kotimäki

Abbreviations and Acronyms

API	Application Program Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture (by Nvidia)
DSP	Digital Signal Processor
GCC	GNU Compiler Collection
GNU	The GNU Project (a free software project)
GPU	Graphics Processing Unit
IP	Intellectual Property, pre-designed hardware circuit
IP	Internet Protocol
ISA	Instruction Set Architecture
JTAG	Joint Test Action Group, used as a port name
LAN	Local Area Network
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
MPI	Message Passing Interface
MPSoC	Multi-Processor System on Chip
NFS	Network File System
NoC	Network on Chip
NTP	Network Time Protocol
NUMA	Non-Uniform Memory Access
OMAP	Open Multimedia Application Platform
PCI	Peripheral Component Interconnect
SCU	Snoop Control Unit
SDK	Software development kit
SD/MMC	Secure Digital MultiMediaCard
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMP	Symmetric multi-processing
SoC	System on Chip
TBB	Intel Threading Building Blocks
TCP	Transmission Control Protocol

TDP	Thermal Design Power
UMA	Uniform Memory Access
USB	Universal Serial Bus
YUV	Luma-chrominance color scape

Contents

List of Tables

List of Figures

1. Introduction

This Master's Thesis is based on a project carried for preparing for future research on multi-processor system-on-chips (MPSoC) that were devised to be the next platform architecture for mobile phones. At the time of the experiment the planned hardware was not available. Therefore the basic concept was recreated by simulating the devised platform and its circumstances. The experiment was composed of two different setups: an ARM based multi-core system that is featured as the general purpose processor of the proposed MPSoC and a multi-processor Intel based system, a cluster, for apprehending the effects of processing element interconnect and network-on-chip (NoC) behavior.

1.1 Scope

The demand for faster processing time and greater throughput as well as the need for better energy efficiency has lead to the popularity of multi-core and multi-processor systems in the computer hardware market. Furthermore, multi-core and multi-processor based solutions are emerging in the embedded system devices. However, taking advantage of the parallelism of the architecture is problematic in the scope of the currently used programming models.

Parallel programming is a widely studied concept with a few well founded programming models for a general solution and a lot of specific solutions for a small subset of parallel problems. The behavior and performance of a parallel program is difficult to grasp for even the most experienced programmers. In addition, the current programming development models do not support parallel programming to the full and the developing tools for it are scarce.

1.2 Problem

The research question of this Master's Thesis work was to find out how the interaction of hardware and software threads in a multi-processor system can be measured and visualized in practice. The aim is to give a better understanding of the behavior of parallel programs for an application developer.

1.3 Result

The results of this Master's Thesis are composed of measurements with two different setups: a multi-core and a multi-processor system. With these systems measurements were made using kernel tracing while running different parallel workloads selected for the experiment. From these resulting thread-level traces a visualization about the run-time activity of the parallel workloads was created. Studying these visualizations an analysis about the program behavior could be made as well as an identification of possible problematic sections that could affect the overall performance or scalability.

1.4 Structure of Thesis

This Master's Thesis is organized as follows:

Chapter 2 takes a look at some of the general concepts and views of the current parallel processing scene consisting of elements from both hardware and software field. This constitutes a review of current and near future System-on-Chip (SoC) solutions and processors. Additionally, a study of the general principles of Message Passing Interface (MPI) is conducted.

Chapter 3 takes the measuring techniques under inspection. This includes a review of performance metrics, the selection of appropriate workloads and the available monitoring techniques.

The experiment setup is described and discussed in Chapter 4. A detailed view of the connectivity of the different measurement components is given with the involving technical specifications. Moreover, measurement workloads and their parameters are described separately.

Finally the results are presented and discussed in Chapter 5, while reviewing the overall success of the work. And at last, the work is summed up in Chapter 6.

2. Background

In this chapter I take a brief look at the hardware and software components that are relevant for multi-core and multi-processor systems in the scope of this Master's Thesis. This includes the general concepts of parallel computing architecture together with a review of some of the existing processors and the parallel programming models.

2.1 Parallel computing architecture

Michael J. Flynn created the classical classification of parallel computer architecture known as the Flynn's taxonomy [15]. The classification is based on whether the architecture can perform single or multiple instructions on single or multiple sets of data. Using this classification, four characterizations of computer architectures are defined:

- The single instruction single data (SISD) is a system capable of executing sequential instructions to a single set of data without any parallelism.
- Single instruction multiple data (SIMD) is an architecture that implements instruction sets operating on vectors, where a single instruction is applied to all of the data in one operation. This type of operation is common in multimedia applications featuring computer graphics.
- Multiple instruction single data (MISD) is a rare architecture, and most commonly used for fault tolerance on critical systems.
- The multiple instruction multiple data (MIMD) is the classification of distributed systems e.g. most multi-processor and multi-core systems.

Furthermore, the occurrence of parallel execution can be implemented on multiple places of the system architecture [13]. The levels suitable for parallelism are instruction, thread and data level:

- Instruction level parallelism is the measure of how many operations can be performed simultaneously by a computer program.
- Thread level parallelism or task parallelism is the form of parallelism that distributes the execution of a computer program across multiple processors.
- Data level parallelism focuses on distributing the data across multiple processors.

2.2 Memory

Main memory access is critical in multi-core and multi-processor systems and affects the way parallel programming is implemented. The existing memory models depend on the characteristics of the multi-processor system and there are a lot of variants. However, generally the memory model in a parallel computer is either shared memory or distributed memory.

2.2.1 Shared memory

In a parallel system with shared memory model the same memory base is offered to all of the processors. Single address base offers a communication channel between the processors when implemented properly. Therefore, the most important feature of shared memory is to maintain cache-coherence, to have a mechanism for keeping the consistency of stored data between the local caches of processors and the main memory. Shared memory access is further categorized to uniform memory access (UMA) and non-uniform memory access (NUMA).

UMA is a shared memory architecture where the access time to any memory is invariant of the location of a memory module or the processor accessing it. The UMA model is not widely used in current parallel computer architectures whereas the NUMA model is featured on most commercial multi-processor systems. NUMA is a memory access scheme where a single shared address

space is visible to all CPUs. The memory access time on a NUMA system varies according to where the memory is located on the system: the access time to the processors' local memory is shorter than the access time to a non-local memory like main memory. This is the simplest form of NUMA. In a more complicated system an interconnect mechanism is used for providing access to processors that are not directly connected to the shared memory. Then there are solutions based on custom hardware and interconnects that connect groups of processors. [14]

2.2.2 Distributed memory

Distributed systems are coarsely defined as a group of systems connected with a reasonably fast network and each system runs a separate operating system and thus separate memory. In a distributed memory system the processors can operate only on their local memory and therefore interprocessor communication has to be done through specific channels.

Benefits of distributed memory are the effective usage of each systems' local resources as there is no competition from other processors for the system bus or memory. Also, the amount of processors is not limited as the only limitation for system size scalability is the interconnect network. Moreover, there is no need to address cache-coherence since each processor operates with its own data.

Contrary to shared memory systems, where the operating system takes care of most of the difficulties in parallel programming, the key issue or downside in programming distributed memory systems is the data distribution among the processors. In a distributed memory system, the interprocessor communication is more demanding and requires message passing between the processors, thus creating overhead.

2.3 Processors

The processor or the Central Processing Unit (CPU) is the active portion of a computer system that follows the instructions of a computer program. In this section I take a look at different types of processor setups, starting with the general idea of System-on-Chips (SoC) and continuing to Multiprocessor System-on-Chips (MPSoC) along with examples of existing implementations.

2.3.1 System-on-Chip

System-on-Chip (SoC) is a computer system integrated into a single integrated circuit (IC) chip. The computer system featured on a SoC typically includes a processor, a selection of different memory blocks, clocks and timers, and external interfaces. Contrary to microcontrollers SoCs have an operating system instead of a specialized software. SoCs are usually built from smaller previously approved hardware blocks (IP blocks) to reduce the designing cost and complexity, but thoroughly custom SoCs also exist.

2.3.2 Network-on-Chip

Network-on-Chip (NoC) is a new approach to on-chip communication; the main idea is to replace traditional wiring between chips with a general purpose network much in the same way as a packet-switched or a circuit-switched telecommunications network. The on-chip network is similarly constructed from multiple links interconnected by switches or routers.

The advantages over previous architectures of dedicated point-to-point wiring or shared buses is scalability and enhances in performance. Also the level of parallelism is increased when the links in the NoC can operate simultaneously on different data packets. Other benefits are in the physical design of chips, where the NoC architecture can reduce the complexity of wiring design and thus gain advantage in power consumption, signal noise, reliability etc. Furthermore, NoC architecture supports modularity and IP block reuse, providing advantages on the system design in the Multiprocessor System-on-Chip architecture.

NoCs are still pretty much an ongoing research and many of the details are still quite open. The existing concepts and techniques of computer networking cannot be adapted straight into use in NoCs since the core benefits would suffer in transition. Routing algorithms should be simple and still take into consideration the unique situation where the network status in terms of e.g. contention is available and can be acted upon. In addition the limitations to network topology are only few when it comes to realizing them into silicon and it is possible to make application specific topologies. [5, 11]

2.3.3 Multiprocessor System-on-Chip

Multiprocessor System-on-Chip (MPSoC) is a set of processing units or components interconnected via a communication network or in the future the Network-on-Chip concept. Currently the most known MPSoC architectures are with identical (homogeneous) processing units, such as the multi-core processors of Intel and AMD. Nevertheless, the architectural combination of different (heterogeneous) processing units is the point of interest in the field of embedded systems, usually used with multimedia and telecommunications applications. [28]

Several heterogeneous MPSoCs are already available and the next generation of Texas Instruments Open Multimedia Application Platform the OMAP4 is within reach. OMAP4 will be a MPSoC featuring a multi-core ARM processor together with digital signal processor (DSP) and other processing elements. In the following, I take a quick review on existing implementations.

Cell BE

Cell BE of Cell Broadband Engine is a microprosessor architecture developed conjointly by Sony, Toshiba and IBM. Cell is most commonly known as the processor of the Sony PlayStation 3 game console (PS3). It is a multi-core chip that is composed of one PowerPC Processor Element (PPE) and multiple Synergistic Processor Elements (SPE). The chip in PS3 has one PPE and eight SPEs.

The Cell chip is a heterogeneous MPSoC where the PPE is a general purpose processor for running the operating system and the SPE units are designed for computationally intensive floating point operations. SPE consists of a Synergistic Processor Unit (SPU), a local storage and a Memory Flow Controller (MFC). MFC provides connection between the local storage and the SPU as well as the connection to the Element Interconnect Bus (EIB) interconnection between all of the PPE and SPE elements. [20]

The Element Interconnect Bus (EIB) is actually a circuit-switched solution of Network-on-Chip architecture. EIB has four 16-byte wide data rings, two in each direction and a shared command bus that connect the 12 elements of Cell BE. Access to the data rings (and the command bus) is controlled by a central data arbiter that implements round-robin arbitration with two priority levels: highest priority for the memory controller (MIC) and lower

priority to all other elements. The command bus uses lossless flow control in the form of tokens. Each element has to have a free token to request the command bus that is returned once the command is complete. Up to 64 outstanding requests per element is allowed. [2]

Nomadik

Nomadik platform developed by STMicroelectronics is a heterogeneous multiprocessor system-on-chip targeted at mobile applications. It is one of the early multimedia MPSoC architectures available and is featured on smart phones manufactured by Samsung. Nomadik is composed of a general-purpose processor and several DSP subsystems. The Nomadik STn8815 has an ARM926EJ RISC processor and hardware accelerators for video, audio, imaging and graphics connected with a multilayer crossbar interconnect bus, advanced microcontroller bus architecture (AMBA). [27]

2.3.4 Multi-core processor

Multi-core processor is a processor composed of two or more independent processing elements integrated into a single integrated circuit die. There are no common characteristics of a multi-core processor and the details of inter-core communication or the allocation of caches vary between implementations. In the following, I review the multi-core processors used in the measurements of this Master's Thesis.

ARM11 MPCore

The ARM11 MPCore [3] chip has four ARM11 MPCore CPUs with Vector Floating Point (VFP) processors. The processors have their own 32KB Level 1 caches and instruction caches. Also there is a 1MB of unified, bypassable Level 2 cache. The chip also features a Snoop Control Unit (SCU), a private timer and watchdog unit per processor and JTAG-based debug.

The Snoop Control Unit (SCU) maintains the coherency between the L1 caches of the ARM11 MPCore processors. It interfaces the MPCore CPUs with each other and with the L2 cache.

The instruction set architecture (ISA) of the ARM11 MPCore is ARMv6

[7]. It is backwards compatible with ARMv5 through compliant memory management and exception handling. ARMv6 has multiple enhancements over ARMv5 most of which improve parallel execution and multiprosessing.

The multiprosessing has been improved with new data sharing and synchronization capabilities. Additional Single Instruction Multiple Data (SIMD) instructions have been added to ISA to improve the support for data level parallelism. Thread level parallelism improvements are in exception handling of multithreading on multiple processors. These include new instructions in the ARMv6 ISA. The memory management has been improved and has now faster average instruction fetch and data latency. Also the processor has to spent less time in waiting for instructions or data cache misses to be loaded.

Intel Atom

The Intel Atom processor family is a line of ultra-low-voltage x86 processors. They are manufactured on a 45 nm process technology and have a specified maximum TDP (Thermal Design Power) of 4 W. Thermal design power represents the maximum power needed for dissipating the heat generated by the chip.

There are one and two core versions of the Intel Atom and the processor supports hyper-threading that enables two virtual processors for each physical core on a system. Furthermore, in addition to the x86 (IA-32) instruction set, they support the x86-64 instruction set. [9]

The architecture of a Intel Atom system is similar to other x86 systems: processor is connected to the chipset that consists of two parts, the Northbridge and the Southbridge. The processor is connected via the 533 MHz Front-side bus (FSB) to the Northbridge that is the memory controller. Southbridge is the I/O controller and is connected to the Northbridge. [8]

2.4 Computing cluster

A computing cluster is a set of computers, commonly called nodes, interconnected together to form a single computer. It is a way of providing a low-cost alternative to high-performance multi-processor systems.

There are several solutions for creating a computing cluster, but the general

architecture is the same. Top of the line solutions employ very fast InfiniBand interconnect setups and high-end computers together with specialized software. However, a similar structure can also be comprised by inexpensive hardware of consumer grade computers networked together by a common Ethernet switch. The basis of this approach was found at the National Aeronautics and Space Administration (NASA) where this cheap architectural design was employed to form a computing cluster called Beowulf which later became the classification of clusters of this type.

The basis of the structure of a Beowulf cluster is a group of identical inexpensive computers that run free and open source software like the Linux operating system. The interconnect is realized using common TCP/IP LAN and the computers interact by message passing. Therefore, the architecture of a cluster can be separated into two components: hardware components and software components.

2.4.1 Hardware components

A computing cluster consists of cluster nodes that provide the computing and data storage capability, and the interconnect network that connects the nodes and enables communication. Furthermore, some commonly used components are a front-end node used for accessing and administering the cluster, and a file server for providing shared storage for the nodes.

The basic hardware requirements for a computing node is to have as many processors or cores, and as much memory as possible. Disk space is not as essential, just enough for the operating system is sufficient, although sometimes it is beneficial to have local disk space on a node as secondary storage. Computing nodes can also be entirely diskless employing network boot from a server. Moreover, it is advisable to have multiple network interfaces for providing dedicated networks for the communication channel and file server traffic, and even provide a separate interface for management and monitoring.

The key characteristic for the interconnect network hardware is bandwidth. Depending on the applied parallel application the interconnect network is usually the bottleneck of the cluster. The low-end approach is to use LAN Ethernet for the interconnect and a one gigabit Ethernet switch can be sufficient in many cases. The more sophisticated solutions are based on fiber channel or InfiniBand interconnects and can provide much greater bandwidth. [4]

2.4.2 Software components

The software components can be divided further into programming tools and management tools. The channel of communication between the computing nodes is in an essential role in distributed computing since the usual methods e.g. shared memory are not available. The basis of computing node interaction in a computing cluster is a message passing library like the Message Passing Interface (MPI) or the Parallel Virtual Machine (PVM). MPI is a parallel programming model for distributed memory systems and I cover it more thoroughly on section 2.6.1. PVM is a parallel programming model as well that can also be run on a heterogeneous computing cluster comprising of different operating systems.

All the necessary components for building and managing a cluster are usually supplied with every operating system distribution. Most essential is a remote login program like secure shell (ssh) or remote shell (rsh) used as the communication channel between the nodes. Distributed secondary storage e.g. Network File System (NFS) is also quite fundamental for the execution of parallel software. Otherwise, there is no obligatory management software. Nevertheless, there are several open source software distributions that are directly aimed for clustering. These distributions can be considered as middleware and they include more sophisticated tools for managing the cluster e.g. automated node setup and monitoring of nodes. One of the most popular is the Rocks Cluster Distribution that is implemented on the CentOS Linux distribution. [4]

2.5 Kernel

Linux is a UNIX variant operating system created by Linux Torvalds of Helsinki university. It has become the most popular open source operating system to date and involves thousands of developers world-wide. Linux is highly portable and supports all the relevant hardware platforms.

The Linux operating system is divided into user space and kernel space. Applications and libraries reside in user space whereas the kernel resides in kernel space. Respectively user space and kernel space have different address spaces and communication between them is handled using system calls. [12]

The Linux kernel like most UNIX operating system kernels is monolithic by

design. In a monolithic architecture virtually all of the operating system functionality resides in the kernel opposite to the microkernel architecture where a very small set of functions is implemented in the kernel and the rest of the services are run on top of the kernel.

According to one of the most definitive operating system books, the Modern Operating Systems by Andrew S. Tanenbaum [34], the typical operating system consists of four major major components: process management, memory management, file management, and I/O device management. This breakdown of operating system elements applies to Linux as well. A more detailed dissection of the Linux kernel is to divide it into the main subsystems according to the elements of its software architecture. A case study of the Linux kernel architecture by Ivan T. Bowman [6] introduces the concrete and conceptual design to comprise of process scheduler, memory manager, file system, network interface, and inter-process communication. Based on these definitions, I have divided the operation of Linux kernel to the following basic components:

- Process management is the part of the kernel that handles everything about processes and their execution. The allocation of execution time for the processes in the CPU is handled by the process scheduler.
- The Linux memory manager implements virtual memory and handles the paging of memory to support it. It keeps track of which pages are full, partially full, or empty and provides the swapping of pages to secondary storage e.g. disks.
- File system is the long-term storage of information in operating systems. Linux supports multiple different file systems and provides a virtual file system (VFS) for a common interface abstraction for them.
- Network interface or the network stack provides the protocols for networking e.g. IP, TCP and UDP.
- Inter-process communication (IPC) is needed for the coordination of communication between processes themselves and with the kernel. IPC mechanisms supported by Linux are signals, pipes, sockets, message queues, semaphores and shared memory.
- The largest part of the Linux kernel source code is in the device drivers. Device drivers and I/O device management in general provide the lower-level functions that the higher-level abstractions need e.g. the virtual file system and the network stack. [12]

2.6 Parallel programming models

Parallel computing is implemented either with shared memory or with distributed memory. Shared memory means that the memory is shared between processing elements in a single address space, whereas distributed memory is distributed, logically and/or physically. Also a hybrid exists, distributed shared memory, where the processing element has both a local memory and access to the memory of other processing elements.

There exists a lot of different approaches to parallel programming; these parallel programming models constitute of a few general solutions and a lot of specialized solutions for a smaller area of parallel problems. Parallel programming models can be implemented as a library (POSIX Threads, MPI, Intel Thread Building Blocks (TBB)), or languages (Haskell, Erlang, CUDA), or as APIs (OpenMP). From these the most general implementations are MPI and OpenMP; MPI uses distributed memory and OpenMP shared memory. In the following, I cover more thoroughly the parallel programming models relevant for this Master's Thesis.

2.6.1 Message Passing Interface

Message Passing Interface (MPI) is a *de facto* standard for parallel programming on some distributed systems, especially scientific computing clusters. MPI is language independent and has bindings for all the major programming languages. Moreover, MPI has been implemented for almost every distributed memory architecture there exists.

MPI provides topology, synchronization and communication functionality between processes. Basic concepts of MPI are:

- A **group** is an ordered set of processes from 0 to N-1, where N is the number of processes in a group.
- A **communicator** is the communication handle for a process group. It provides the means to conduct communications using the message passing routines.

Point-to-point functions are used for communication between two specific processes. The simplest functions are `MPI_Send` that allows a process to

send a message and `MPI_Recv` that sets the process to receive a message from another. Also functions for blocking and non-blocking point-to-point communication mechanisms are specified.

Mechanisms for addressing all processes or a subset of a process group are called collective functions. `MPI_Bcast` is the MPI equivalent of a network broadcast. The MPI-2 specifies three one-sided communication routines: `MPI_Put` for writing data into a memory on a remote process, `MPI_Get` for reading from a memory of a remote process and `MPI_Accumulate` for combining the contents of the origin buffer with that of a target buffer.

The `MPI_Put` and `MPI_Get` are not as strict communication methods compared to the point-to-point communication mechanisms `MPI_Send` and `MPI_Recv`. Where as in the point-to-point message passing the sender and the receiver are synchronized when transferring data, in the one-sided communication the sender is free to continue processing once the data is sent without the involvement of the receiving processor.

MPI programs are run using a job launcher script (usually `mpirun` or `mpiexec`) that starts the MPI job with the desired amount of processes and on the desired hosts given on command line or in a config file.

There are several implementations of the MPI standard. Both commercial and open source. The most commonly used implementations are Open MPI and MPICH2.

Open MPI [16] is an open source implementation of MPI. It is a merger of three MPI implementations: FT-MPI from the University of Tennessee, LA-MPI from Los Alamos National Laboratory and LAM/MPI from Indiana University with contributions from the PACX-MPI team at University of Stuttgart. Open MPI supports a wide range of operating systems and hardware platforms.

MPICH2 [18] is the initial implementation of the MPI standard (both MPI-1 and MPI-2) by Argonne National Laboratory. It is freely available as open source.

2.6.2 OpenMP

OpenMP [26] is a shared memory parallel programming API. It is based on compiler directives called pragmas that are used to define a parallel region

in source code and more specifically clauses for the nature of region e.g. the applied synchronization or data sharing. The compiler then interprets the definitions into parallel code.

2.6.3 POSIX threads

POSIX threads is a shared memory parallel programming API for managing threads. It is part of most of the UNIX variant operating systems.

3. Measuring techniques

This chapter focuses on different methods available for system activity and behavior measurements. These methods conclude both intrusive and non-intrusive means as well as real-time and post-processing methods.

The method is coarsely as follows: selecting the evaluation technique, selecting the workload, the designing of experiments, analysis and interpretation of data, and presentation of data.

3.1 Selecting evaluation technique

According to Raj Jain's book on The Art of Computer Systems Performance Analysis [19] the three techniques for performance evaluation are: analytical modeling, simulation and measurement. In this Master's Thesis measurement is the obvious technique for performance evaluation since we are measuring a real system even though it is a recreation simulating the devised future hardware.

3.2 Selecting performance metrics

Selecting the right metrics for performance evaluation is crucial. Metrics related to system performance measuring can be time, rate or resource oriented where the corresponding metrics are responsiveness, productivity or utilization. There are also metrics that are associated with service, error and unavailability, where the related metrics are speed, reliability and availability. [19]

Responsiveness can be measured by response time, turnaround time or reac-

tion time. Response time is defined as the time lapsed from the request by the user to the response of the system. Turnaround time is a metric for batch jobs and the time between the submission of a batch job to the completion of its output. Reaction time means the time from submission of a request to the beginning of its execution.

Rate can be measured by throughput that is the number of requests per unit of time. Utilization is measured as the ratio of busy time and elapsed time over a period of time. The measurement of reliability is the probability of errors or the mean time between errors. The availability is defined as the time a service is available for requests. The time when a system is available is called uptime and the mean of uptime is known as Mean Time To Failure (MTTF).

The metrics applied in the measurements of this Master's Thesis are resource oriented. The system activity, the interaction of software and hardware threads, is evaluated based on the balance of the distribution of parallel execution of different processors and therefore the efficient use of system resources.

3.3 Selecting workloads

The performance measurement of a system should be repeatable and comparable for later measurements. Therefore the activity of the system has to be well defined and preferably generated with a workload or a benchmark to set a standard for measuring.

A workload can be real or synthetic. When a system is measured while being used in normal operation, the workload can be considered as a real workload. It can be non-repeatable especially when dealing with a large system or application and therefore unsuitable in certain performance measurements. On the other hand, a synthetic workload can be repeated in a controlled manner and without a change. Synthetic workloads have been developed for measuring certain features of a system or its sub-components and they try to mimic a real workload by trying to apply similar load to a system with the same characteristics. Regardless, the performance of the feature cannot be easily put into the context of a real workload. In the following sections I cover some common benchmarks and review the basis of selecting appropriate types of workloads for measurements.

3.3.1 Common benchmarks

A benchmark is usually considered as a synonym to a workload and no serious distinction has been defined between the two. However, benchmarking can be seen as the process of performance comparison between multiple systems and the workloads used in the measurements as benchmarks [19]. In the following, I review a few commonly used benchmarks.

Whetstone benchmark

Whetstone benchmark [10] is a synthetic benchmark developed for measuring the computing power of computers featuring such tests as array addressing, fixed- and floating-point arithmetic, subroutine calls, and parameter passing. The benchmark is a CPU-bound test focusing mostly on floating-point performance and is considered as a representative of small CPU intensive applications.

Dhrystone benchmark

Dhrystone benchmark [36] is a synthetic benchmark developed for measuring the integer performance of a computer. Dhrystone benchmark can be considered as a counterpart for Whetstone benchmark as it does not test floating-point arithmetic, hence the name of the benchmark. Like Whetstone it is small in size and fits easily into L1 cache when executed and cannot be considered to represent a real application.

Rhealstone benchmark

Rhealstone benchmark is a fine-grained real-time operating system (RTOS) benchmark measuring the average duration of basic operations of an operating system. These measurements conclude task switching time, task preemption time, interrupt latency time, semaphore shuffling time, deadlock breaking time and datagram throughput time. [21, 22]

Hartstone benchmark

Hartstone benchmark [37] is a application-oriented benchmark for real-time operating systems. It implements a series of synthetic tests with increasing demands of resources while measuring the number of missed deadlines.

SPEC benchmark suite

Systems Performance Evaluation Cooperative (SPEC) [31] is a nonprofit corporation formed by computer vendors to establish a industry standard for performance evaluation. The SPEC benchmark suite consists of multiple different benchmarks for various circumstances.

3.3.2 Selecting appropriate workload

When reviewing workloads for a measurement the four main considerations are the services exercised by the workload, the level of detail, representativeness, and timeliness. [19]

The system under measurement can be considered as a service provider that offers services, therefore the selection of a workload that will exercise the right services is important as well as determining what is a service.

Choosing the level of detail in creating a workload can be a detailed list of all requests possible or just concentrating on the most frequent ones. Possible levels in order of the least detail to the most are: most frequent request, frequency of request types, time-stamped sequence of requests, average resource demand, and distribution of resource demands.

Representativeness refers to how well a workload behaves compared to a real application. Three main attributes of a workload: arrival rate, resource demands, and resource usage should be same or proportional to that of the application simulated.

Timeliness is the criteria of how well the workload follows the changes of user behavior. The usage pattern of an application changes over time as the user invents new demands for the system. Therefore, a continuous development of the workload is advisable.

Other minor considerations are the loading level, impact of external components, and repeatability. Loading level refers to the measure of stress the workload puts the system under, whether the workload should test the system on full capacity, beyond capacity, or with the load of a real workload. Impact of external components is the effect of components outside the system that can impact the performance e.g. I/O devices. Repeatability is the ability to duplicate the results without much variance.

Considerations for the experiment

The common benchmarks introduced are too limited to be of use as parallel workloads except the SPEC benchmarks. The SPEC benchmarks for MPI and OpenMP would have been good workloads for my measurements, but the bureaucracy involving the purchase of the benchmarks and the time it would have taken to deliver them led to the decision to search for other workloads.

In the light of the presented selection criteria, when considering the system under study and the desired results I hope to produce, the main factors regarding the selection of the workloads are the services exercised, representativeness, and timeliness.

The purpose of the study is to visualize thread interaction in the system to better understand parallel programming. Therefore, the service I will be exercising in the system is thread interaction in the light of different parallel programming models so the workloads should bring forth possible problem areas in the field.

Timeliness is important because the system under study is a prototype of a next generation mobile platform where the usage profile can very well change from the current behavior of traditional mobile phone applications to the direction of multimedia. Furthermore, the multi-processor platform offers new possibilities performance-wise. Hence, the workloads should explore the possibilities of fitting this behavior for parallel execution.

Representativeness is taken into account by using real applications as some of the workloads.

3.4 Monitoring system activity

A monitor is a tool used for observing system activity. In general, the operation of a monitor is to observe performance, collect statistics, analyze data, and display results. There are several types of monitors that can be categorized into software monitors, hardware monitors, and firmware monitors. Firmware monitors are a mixture of both software and hardware monitors, and they are usually used in diagnostic purposes of embedded systems e.g. the diagnostic lights of a mother board of a computer system.

Moreover, monitors can be classified based on implementation level, trigger mechanism, or result displaying ability. Implementation level classification is based on whether the monitor is implemented as a software monitor, hardware monitor, firmware monitor, or hybrid monitor. Trigger mechanism can be classified to event-driven monitors and sampling monitors. Event-driven monitor is triggered into action when certain circumstances are met, the opening of a file for instance. Sampling monitors trigger at fixed time intervals. Classification by result displaying ability is further categorized as on-line monitors and batch monitors. On-line monitor displays the system activity during the measurement either continuously or at fixed intervals, whereas batch monitors first collect the data and analysis is done afterwards, even with a separate tool [19]. In the following sections I cover software and hardware monitors more thoroughly.

3.4.1 Software monitor

Software monitors are used for the monitoring of operating systems and applications at a higher level. An overview of relevant design issues and choice of software monitors are featured on this section.

Activation mechanism of a software monitor can be based on trap instruction, trace mode, or timer interrupt. Trap instructions are instrumented into the operating system kernel and contain a callback to the data collection routine that is executed whenever the trap is reached. Trace mode is a mode available in some processors where the execution is interrupted after every instruction or every branch for data collection and recording. Timer interrupt based software monitors are sampling monitors that interrupt the execution on fixed intervals using the timer-interrupt service provided by the operating system to collect data.

Data collection of software monitors is usually handled through buffers for minimizing the effects of I/O operations. The use of multiple buffers diminishes the effects further as well as adequate buffer size that keeps in balance between the rate of the input and the write out to slower memory.

3.4.2 Hardware monitor

Hardware monitors on the other hand are separate systems that are connected to the measured system via probes or a connector. The JTAG debug port is standardized and widely used hardware monitor connection. Hardware monitors consume no resources of the connected system and have a higher input rate in comparison to software monitors. Moreover, they are less likely to introduce bugs into the monitored system and can collect data from the beginning of the boot sequence of the monitored system. However, the monitoring of higher level information is tedious and easier to accomplish with software monitors.

3.4.3 Instrumentation

There are different ways for implementing instrumentation in software development. These approaches can be classified to two distinctive categories: source-level instrumentation and binary instrumentation.

Source-level instrumentation is done either manually by the programmer by adding instructions to the program code for measuring the execution on run-time or via an automated approach where the instrumentation is added automatically to the code by a specific tool acting according to a policy.

Manual source-level instrumentation is usually done at higher level either through the common print statement or taking advantage of such APIs as Java logging, Apache logging or UNIX syslog.

Automated source-level instrumentation is done when compiling the program. Compiler instruments the code at compile time and the profiling is done when the program is run. For instance, the GNU Compiler Collection (GCC) offers a lot of options for automatic instrumentation: the option `-finstrument-functions` automatically generates tracepoints for every entry and exit to functions. Furthermore, the compile time option `-pg` of GCC instruments the code for profiling using gprof the GNU profiling tool.

Binary instrumentation is the instrumentation of an already compiled binary. Both static and dynamic approaches exist. Static binary instrumentation is quite uncommon and mostly implemented on architectures with a fixed length instructions such as MIPS. Dynamic binary instrumentation on the other hand is the basis of several relatively new solutions such as Kernel Dynamic Probes (Kprobes) for Linux and DTrace for Solaris. The instrumentation technique is to insert trap instructions at target addresses that when encountered transfer the execution to the instrumentation code via an interrupt.

In the following, I briefly present some of the existing software monitors for operating system instrumentation.

Kernel Dynamic Probes

Kernel Dynamic Probes (Kprobes) [23] is an interface for inserting breakpoints into a running Linux kernel without disruption. Kprobes provides a dynamic mechanism to instrument the kernel with breakpoint instructions at a given address and collect the data when the breakpoint is tripped. It comes with the Linux kernel currently and is built as a kernel module that can be loaded or unloaded to the kernel whenever needed.

SystemTap

SystemTap [33] is a dynamic tracing framework developed for the Linux operating system. The trace activation mechanism of SystemTap is based on instrumenting the Linux kernel using the Kernel Dynamic Probes (Kprobes). SystemTap includes a scripting language that simplifies the creation of probes through Kprobes and better enables the sharing of found solutions and reuse.

DTrace

DTrace [32] is a dynamic tracing framework similar to SystemTap but developed by Sun Microsystems for the Solaris operating system. It is also available for BSD UNIX derivatives like Mac OS X and FreeBSD, and the support for Linux is also under development. The operation of DTrace is based on thousands of kernel probes inserted to the operating system kernel that fire when specific circumstances are met. The tracing is controlled

by scripts written in the D programming language, a language developed especially for DTrace.

Linux Trace Toolkit next generation

Linux Trace Toolkit next generation (LTTng) [24] is a static instrumentation for the Linux kernel that is based on the Linux Trace Toolkit (LTT), a previous attempt at the matter. The activation mechanism of LTTng is a set of probes in the Linux kernel. Whenever these probes are triggered by an event the corresponding data is logged by the LTTng, like the trap instruction mechanism described in section 3.4.1. The support for dynamic instrumentation using Kprobes is under development.

Considerations for the experiment

Based on the operating system in use (Linux), the ease of taking into use, and the active and mature development of the software monitor, I have selected LTTng as the software monitor for the measurements in this Master's Thesis.

3.5 Presentation of results

Selecting the right presentation method for results is as important as the measurements themselves. A good chart gives the maximum information in minimum effort from the reader.

The type of variables, quantitative or qualitative, is an important factor that influences the choice of the method of presentation. Quantitative variables are numeric and either discrete or continuous. Qualitative variables express states, levels, or categories. Continuous variables are displayed using a line chart, whereas discrete or qualitative variables are better displayed with a column chart or a bar chart. Typically performance results are displayed with line charts, bar charts, and histograms. Furthermore, there are presentation methods developed for performance analysis that can be better applied to the examination of computer system performance: Kiviat graphs and Gantt charts. [19]

Kiviat graphs illustrate the balance or imbalance of a system effectively and

attribute-level comparison between different systems is effortless. However, it cannot be used for the visualization of thread interaction and therefore is not suitable for the visualization of system activity that I am pursuing.

Gantt charts are used for illustrating schedules and commonly used in project managing for work scheduling. When considering scheduling of an operating system, it can be described as a kind of project managing in itself, therefore the Gantt chart is an obvious choice for the purpose of this Master's Thesis. Moreover, LTTV, the visualization tool of LTTng, uses a form of Gantt chart to display its results and therefore further assuring the choice.

4. Experiments

In this chapter I take a look at the experiment setup and measurement arrangements for the contributing part of this Master's thesis. Measuring equipment is described in detail introducing the key elements and characteristics of the hardware utilized. Moreover, the fundamental principles of the measuring software are examined and the connectivity of the entire setup is presented with both hardware-software interaction. Furthermore, the measured applications are introduced and their functionality is described along with how they fit into the experiment as measurement workloads.

4.1 Measurement equipment

The measuring equipment used in the experiments consists of two different hardware setups running the Linux kernel, the parallel programming models and the tracing tool. For the benefit of understanding the behavior of the workloads and their performance results, I introduce next the components of the equipment thoroughly and with enough detail. In addition, the tracing tool, LTTng, is reviewed more carefully covering its operation and design principles.

4.1.1 ARM11 MPCore

The first measurement setup is built on top of an ARM11 MPCore [3] system. The setup in use is a prototype ARM11 MPCore chip introduced in section 2.3.4. This chip is mounted to a RealView baseboard. The OS for the setup is Debian Linux for ARM.

The RealView baseboard [29] has the NEC Corporations implementation of

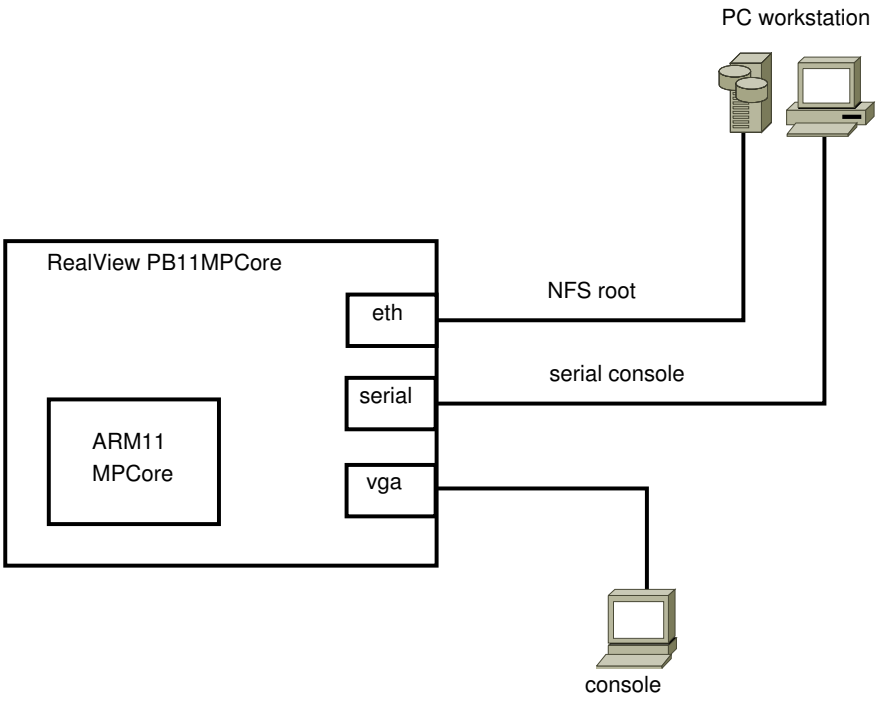

Figure 4.1: *ARM11 MPCore measurement setup.*

the ARM11 MPCore chip. The chip is connected via a 64-bit AMBA AXI interface to the Northbridge. The baseboard is packaged in an ATX case with standard ATX power supply unit and PC-like set of peripherals including PCI and PCIe buses, USB host controller, SD/MMC slot, Ethernet and the usual HCI components: keyboard, mouse, video (DVI) and audio. There is also a console interface via RS-232 and a JTAG connector for debugging.

The Northbridge contains the following implementations of system components: static and dynamic memory controllers, DMA controller, LCD controller, AXI controller and interfaces to the ARM11 MPCore chip, AHB-Lite interface to the Southbridge and PCI/PCI-X interface. The Southbridge provides the implementations of several peripheral components and interfaces, such as: audio codec, PS2 keyboard/mouse, multimedia and smart card interfaces, UART, Watchdog module, Dual-Timer module, Real Time Clock, AHB interface to the baseboard Compact Flash memory.

In addition, the measurement system consisted of a PC workstation that acted as the file server for the ARM11 MPcore using NFS for the root file system, measurement and workload applications, and storing the measurement data. The whole measurement system with connections is described

in figure 4.1. Boot image and boot parameters were stored to a SD flash memory chip and operated through the serial console via the PC workstation using a terminal program (minicom). The PC workstation was also used to cross-compile the kernel for the system.

Memory support for the ARM11 MPCore was found very experimental on the Debian Linux. The most stable setup and almost the only working setup was to address only 256 MB of the 512 MB of memory present. Using more than 256 MB usually hung up the system, apparently due to the memory in use being overwritten. With some patches to the system, it was possible to address a few megabytes more, but nothing nearly useful. This memory problem took a lot of time to discover, as I was looking for the problem elsewhere, mainly on the configuration of the kernel build.

Open MPI was my first choice for the ARM11 MPCore system, but as it turned out, the support for Open MPI is not on par with Debian Linux for ARM: components that Open MPI depended were not yet ported to the system. In the scope of this Master's thesis it would have been a side track too far to venture porting the missing components. As the other MPI implementation was not likely to interfere with the results, I went a head with the MPICH2 that compiled and worked on the system without any trouble.

Considering the performance of the ARM11 MPCore, I made the decision to use `rsh` as the communication channel of MPICH2 instead of the more secure `ssh`. Moreover, a secure communication channel was not thought as necessary in the experiment setup as the connections were made via the loop back network device, and because the system was not connected to the Internet.

The MPICH2 job launcher script is `mpirun` and the command that was used to start the workloads was:

```
mpirun -np <num. of procs> <workload> <workload parameters>
```

where the `-np` stands to specify the number of processors to run on.

4.1.2 Intel Atom cluster

For a real inter-system experiment, measurements were conducted on a small cluster consisting of Intel Atom processors. The setup is a front node plus

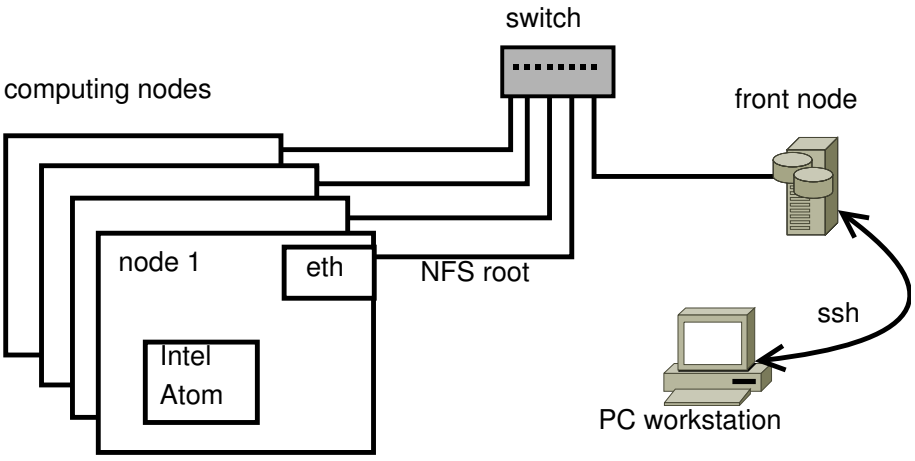

Figure 4.2: *Intel Atom cluster measurement setup.*

four "slave" nodes as described in Table 4.1.

Table 4.1: Intel Atom cluster specifications

Feature	Front node	Slave nodes
Board	Intel D945GCLF2	Intel D945GCLF
Processor	Intel Atom 330	Intel Atom 230
Cores	2	1
FSB	533 MHz	533 MHz
Clock speed	1.6 GHz	1.6 GHz
L2 cache	512 KB per core	512 KB
Memory	1 GB	2 GB

Both the front node and slave nodes support hyper-threading that provides them with logical processors twice the count of cores, thus the total logical processors on the front node is four and on a slave node is two.

The setup of the cluster was a bit similar to the ARM11 MPCore system: the front node provides the slave nodes' root file system via NFS. In addition, the clocks of the slave nodes were synchronized to the front node using NTP to avoid them from drifting apart. OS for the system was Linux and Ubuntu server in particular. The measurement setup with connections is described in figure 4.2.

The Intel Atom cluster was not as exotic hardware as the ARM11 MPCore, so there were no issues against using Open MPI as the MPI implementa-

tion. Moreover, as the cluster was going to be used for other experiments on another project with Open MPI, I wanted to go along with the original plan. The Open MPI installed was a precompiled package from the Ubuntu repositories and was setup without any difficulties.

I selected `ssh` as the communication channel for Open MPI between the nodes. The choice was obvious as there were not any performance issues to be considered like in the case of the ARM11 MPCore system.

The job launcher script is the familiar `mpirun` with the same parameters as with MPICH2. Due to the nature of the system, I also needed to provide a list of participating nodes of the cluster:

```
mpirun -machinefile <machines> -np <num. of procs> <workload>
<workload parameters>
```

where the `-machinefile` is a list of the nodes participating in the run and `-np` stands to specify the number of processors to run on.

4.2 Linux Trace Toolkit Next Generation

Linux Trace Toolkit Next Generation (LTTng) is an OS kernel tracer that generates traces of both Linux kernel space and user space operation. It has a low impact on system performance and moreover, it is a low overhead tool suitable for performance measurements.

The operation of LTTng is divided into three parts: the user space controlling part called `lttctl`, user space daemon `lttd` and the kernel part. `lttctl` is the command-line control application through which the controlling of kernel tracing and starting and stopping of a trace is done among other things. The user space daemon `lttd` is started by `lttctl` and is responsible for the writing of trace data to disk, whereas the kernel part is responsible for the tracing in the kernel.

The general operation of LTTng is described in figure 4.3. Kernel events are traced by the kernel part of LTTng and written through DebugFS. When the circular buffer of a trace channel on DebugFS is filled, the data is passed to the user space daemon `lttd` that writes the data to disk. The sizes of circular buffers can be set manually upon starting the trace.

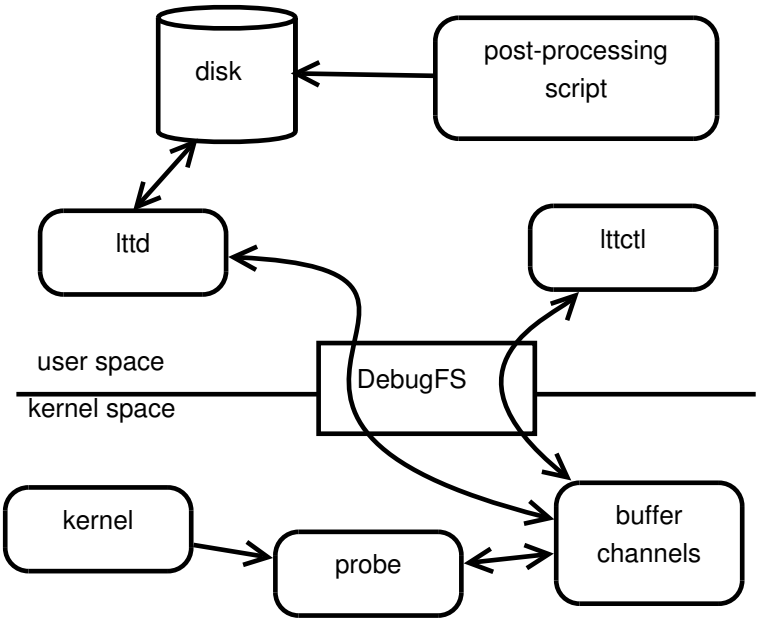

Figure 4.3: *LTTng operation diagram.*

User space events are recorded with two different paths: slow tracing path and fast tracing path. The slow tracing path is for events that have a low event throughput and is handled through a system call. The fast tracing path is for user space events that have a high throughput. The fast path is a library (`libltt-usertrace-fast`) with which the traced thread communicates through a circular buffer. When the buffer is full, it is dumped to a thread-specific companion process that writes the trace data directly to disk. [25]

The operation events that the LTTng keeps track of when tracing are listed in Table 4.2. The usual behavior of a thread is that it spends a lot of time in either WAIT, SYSCALL of USER_MODE events. SOFTIRQ events happen a lot too, but they are very sort and can seldom be seen as the resolution of the graphs is not enough when viewing the whole trace.

Due to the limitations of Linux kernel time precision LTTng reads directly the CPU timestamp counters that are converted to nanoseconds during the post-processing of the trace.

Table 4.2: LTTng operation events of threads

WAIT	I/O wait - thread is waiting for a device.
SYSCALL	Thread makes a system call to the OS.
USER_MODE	Thread runs accordingly to its program.
SOFTIRQ	Software interrupt by a device driver.
FORK	Thread is being forked.
CPU_WAIT	Thread is waiting for CPU to process.
EXIT	Thread exits.

4.3 Workloads

The workloads for the experiment were chosen to depict some of the common usage scenarios in computing to show how an application considered as familiar might perform quite unexpectedly when run in parallel – an analog to the hardships and different ways of thinking parallel programming requires.

The workloads can be categorized to workloads that were implemented for this experiment i.e. the quicksort variations and to workloads that are in a way proven concepts through wide usage in well-known software projects and were available as implementations of third parties.

4.3.1 Quicksort

Quicksort is generally recognized as the fastest sorting algorithm in an average case. The nature of this workload is to give some insight into the behavior of a well-known algorithm when translated to a parallel form, i.e. to see if the parallelization of the algorithm leads to improvements.

The nature of the quicksort algorithm sometimes leads to situations, where the sortable arrays vary in size considerably during the execution and thus easily create an unbalanced distribution of sortable elements among the sorting partitions. This phenomenon might become an issue when parallel execution is in question as some of the processing capacity is not utilized to the full extend.

Due to limitations experienced during preliminary testing, the data set of an array of the size of one hundred thousand (100000) was chosen for the experiments run on ARM11 MPCore. For the Intel Atom cluster, the data

set was chosen to be somewhat greater since a comparison in overall performance against ARM11 MPCore would be pointless. In addition, a greater number of sortable data might magnify problems in the implementation, if any. Therefore the array size of one million (1000000). The parameter of the random seed value of 32526 were kept the same for both platforms for the sake of consistency.

MPI quicksort

The parallelization of quicksort with MPI is a bit more complex than when dealing with shared memory, as the data has to be sent to different nodes. Therefore initial data partitioning was needed for the sortable data set before any sorting could be done. The implementation of the initial partitioning is a kind of a pre-sort, where the data is ordered according to randomly selected pivot, much like in a typical quicksort algorithm. The implementation goes as follows:

Let's assume there are four available computing nodes: p0, p1, p2 and p3.

1. Divide the data to each computing node
2. The first node p0 (the root node) chooses a pivot by random from its data set and then broadcasts it to the other nodes
3. The nodes are divided to lower and upper halves and they exchange values based on the pivot i.e. in a four node case p0 exchanges values with p2 and p1 with p3
4. First nodes in both lower and upper half choose another pivot by random and broadcast it to other nodes in their half
5. Based on the pivot, the nodes exchange values
6. Each node sorts its values using quicksort
7. The values are combined in order of the nodes

The MPI setup of the Intel Atom cluster for this workload was to use Open MPI with ssh as the communication channel between the nodes.

OpenMP quicksort

Parallelization of a shared-memory solution was done with moderate ease as it did not have to include any pre-sorting or other considerations with the partitioning of the sortable data set. The algorithm used is a general quicksort rigged with insertion sort. Parallelization was realized by OpenMP sections where a new parallel scope is opened on every recursion. The OpenMP compiler used for the compiling of the workload is the GNU Compiler Collection (GCC) 4.3.2 featuring support for OpenMP 2.5.

4.3.2 Data compression

Data compression is a common workload in many applications and is considered to be highly parallelizable. Some form of data compression is used almost in every device that handles significant amounts of data. It is typically embedded to the file system and therefore quite transparent to the user.

Two data compression applications were selected as workloads: parallel BZIP2 (MPIBZIP2) and parallel GZIP (PIGZ). The compression data was selected to be a 2 MB file of filled with randomly generated data.

MPIBZIP2 [17] is a parallel version of the popular bzip2 [30] data compression utility. The parallelism is implemented as a hybrid using both MPI and POSIX threads.

PIGZ [1] is a parallel version of the GNU zip data compression utility. It uses the POSIX Threads for parallel execution.

4.3.3 Video encoding

Video encoding is featured on many mobile phones nowadays and can be considered as one of the key applications in the near future of mobile devices. Therefore, I have selected a video encoder as one of the workloads.

The x264 [38] is a library for encoding H264/AVC video streams that is used in multimedia players e.g. VLC multimedia player [35]. The purpose of the workload is to have a somewhat real life workload for the experiment. The parallelism of the x264 encoder is implemented with POSIX Threads. The

input file for encoding was a YUV video file containing 50 frames of video with a resolution of 352x288.

4.4 Conclusion

The objectives for this chapter were the presentation of the experiment setups and measuring components, their interaction and the corresponding measuring workloads with description on how they work. With the presented facts it should now be apparent how one would arrange these experiment setups and perform the measurements again.

The pitfalls of completing the measurements manifested in both hardware and software: establishing the setup proved to be harder on the experimental hardware of ARM11 MPCore, but the experiments were run on it easily. On the contrary, the Atom cluster was easily set up for the measurements, but the actual measurement had its problems due to the unsupported operation of the measuring software LTTng on a cluster.

5. Results

The results of individual measurements described and setup in Chapter 4 are discussed in this chapter. First, I take a look at the experiences and difficulties with setting up and operating the measurement software and hardware, then a discussion of the results obtained with each workload along with analysis of their run-time behavior.

5.1 LTTng measurements

Setting up the LTTng for tracing and operating it was straightforward and the ARM11 MPCore behaved in a stable manner with only two kernel crashes during the workload measurements. Taking into account that the hardware is still a prototype, this was not that unacceptable.

LTTng includes a post-processing tool called LTTV (Linux Trace Toolkit Viewer) that has an API for developing post-processing modules to it. There is also a graphical user interface for viewing the traces, but it lacks the means of exporting these visualizations. Therefore, the measurement data was extracted by a LTTV module called `textDump` that dumps the whole trace data to ASCII text. Then this data was refined with a post-processing script consisting of shell scripts and other programming utilities to form graphs for further analysis.

The support for ARM11 MPCore was easily arranged since LTTng is architecture independent, although the kernel needed to be patched for LTTng support. Nevertheless some anomalies where experienced during the workload measurements:

- LTTng mapped system calls to the wrong process sometimes, and al-

ways to one of the rpciod0-3 processes instead of the user space process.

- A few of the trace records were broken and failed to read on post-processing.

These problems are most probably due to the experimental hardware and that LTTng is still very much under development. Nevertheless, these anomalies were easy to notice and the measurements were just run again and therefore did not affect the experiment results.

The support for Intel Atom was as straightforward as with the ARM11 MPCore. Due to the more familiar x86 architecture, a more recent kernel was available and thus a bit more advanced LTTng with support for dynamically allocated trace channels and a lot of bug fixes. The only real challenge was to synchronize the trace data from each node since LTTng does not yet support distributed tracing. Therefore, the data was recorded separately by each node and gathered and combined afterwards.

In addition, LTTng timestamps trace data according to the system uptime (CPU timestamp counter), which produced the need to synchronize the data between the nodes. This was addressed accordingly in the post-processing script. Furthermore, the data needed to be processed separately before merging together to avoid misplacing local context-switches as inter-system ones. This was also taken into account in the post-processing script.

5.2 Workloads

As described in section 4.3, the workloads were chosen to depict some of the common usage scenarios that come up when executing some everyday applications. The presumption was that these familiar actions might perform in a way not expected when run in parallel, and as I go through each workload below, this was just the case on some of them.

The validity of the experiments was ensured by running each workload measurement ten times and comparing the resulting LTTng profiling images with each other for major differences. No other verifications were used as the nature of the obtained data is not applicable with the usual proofing methods and cannot not be subjected to statistical analysis.

The measurements on the ARM11 MPCore were first run on two processors

and then on four processors. This setup is then used for the analysis of scalability of the workloads.

5.2.1 MPI quicksort

The activity of MPI quicksort was measured on the Intel Atom cluster described in section 4.1.2. During the first measurements, tracing created such an abundant amount of trace data that it did not only affect the workload performance gravely, but the parsing of trace data for visualization as well. Turned out that the dynamically allocated trace buffers for some of the trace channels were not large enough thus overflowing quickly and resulting in writing the data straight to disk instead of memory. The write procedure created a recursive-like condition where the data collection ended up contributing to the trace data as the computing nodes of the cluster have their root file systems on NFS over the same network they communicate with. The result was an overwhelming amount of data that needed to be written to a relatively slow media which resulted as a lot of dropped trace data. After assessing the problem, the measurement was then carried out using larger buffer sizes for the kernel, network and memory manager channels.

The operation of MPI quicksort when run on the slave nodes of the cluster with full eight cores in use is shown in figure 5.1. The controlling node is node 1.1 and it starts by creating an array of random numbers for sorting and divides it evenly among the other nodes (the short green period run on user mode). After that, in a series of very short periods between user mode and system call state (see table 4.2 for description of the different LTTng events), the nodes negotiate the pivots and exchange data (as described in the experiments chapter at section 4.3.1). The long runs on user mode after that is the regular quicksort algorithm run on the nodes' data. In the end, the sorted data is sent back to the controlling node, the node1.1, to be concatenated.

When taking a look at the overall performance, the load of different cores can be seen as unbalanced. Interestingly, the experiment brought forward the result with a drastically uneven distribution of sortable data that manifests most notably with node 4.1. It spends almost the whole execution time in wait state at the regular quicksort phase of the algorithm as it has nothing to sort. Moreover, cores node 2.1 and node 3.2 do over twice the work or more than the other nodes.

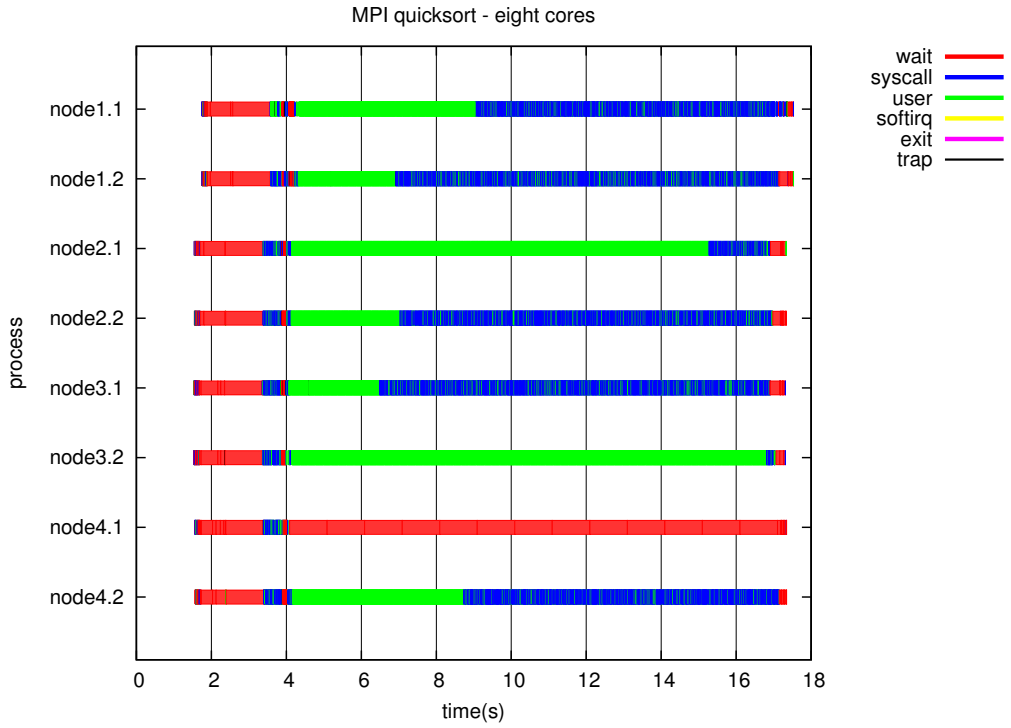

Figure 5.1: *MPI quicksort profiled running on eight threads.*

The discovered imbalance in performance could be quite severe when dealing with large data sets. In addition, increasing the number of processors might not contribute to the sorting efficiency, but result to more computing nodes with nothing to do due to an unsatisfactory distribution of the pivots.

Taking into account these presented facts and observations, a programmer can now device improvements for the task, or in this case the algorithm. Employing some form of a pre-sort or an initial partitioning for the data to figure out clever pivots for the algorithm could be considered as a solution for the problem.

The result of this measurement shows the usability of kernel tracing as a way to analyze performance issues, as it provides an illustrative understanding of the problems of the load distribution of the algorithm.

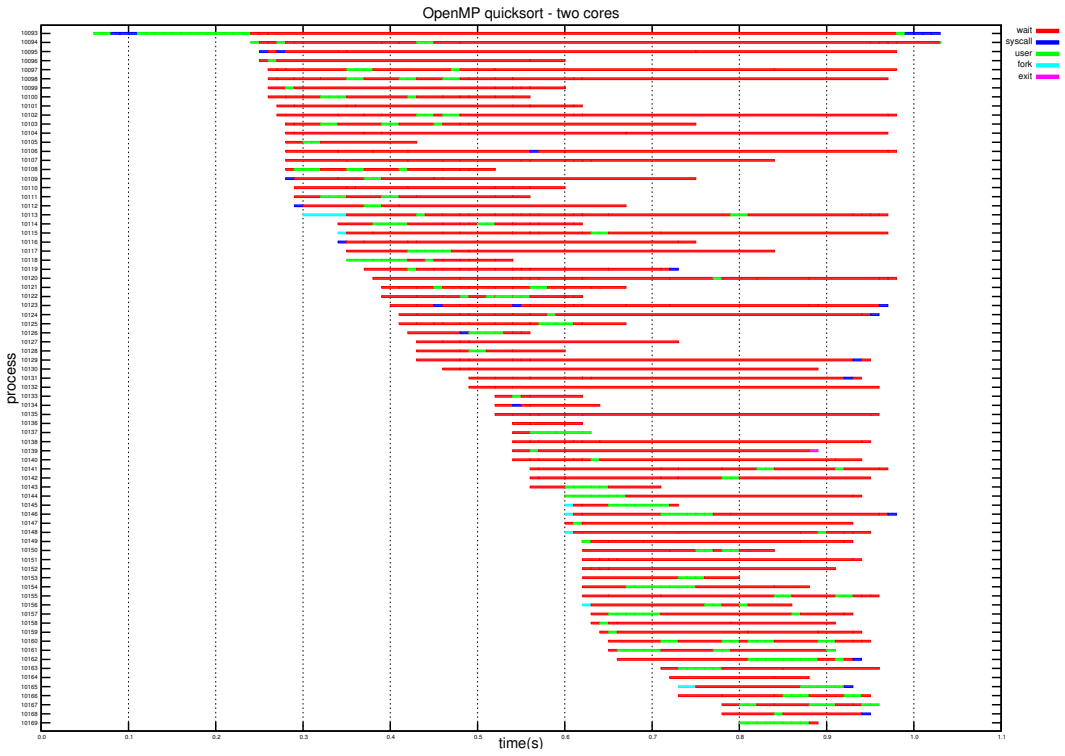

Figure 5.2: *OpenMP quicksort profiled running on two threads.*

5.2.2 OpenMP quicksort

The measurements for the OpenMP quicksort were conducted on the ARM11 MPCore setup (section 4.1.1). Encountered problems were memory related as the ARM11 MPCore ran out of memory with larger data sets as described later. The workloads were measured when run on two and then on four cores of the system. From these measurements the behavior and performance graphs were generated.

The operation of OpenMP quicksort can be seen in figure 5.2. The execution starts with one thread, doing initialization, allocating memory (SYSCALL state) and filling an array with random data (USER state). After these the first thread goes to state WAIT (red line) and the recursion algorithm of quicksort begins. The large amount of threads are due to the nested `omp parallel` scopes of OpenMP created on each iteration of the recursion loop.

The trace profile of quicksort suggests that OpenMP might not be the right way to implement a parallel quicksort or that it needs a different approach

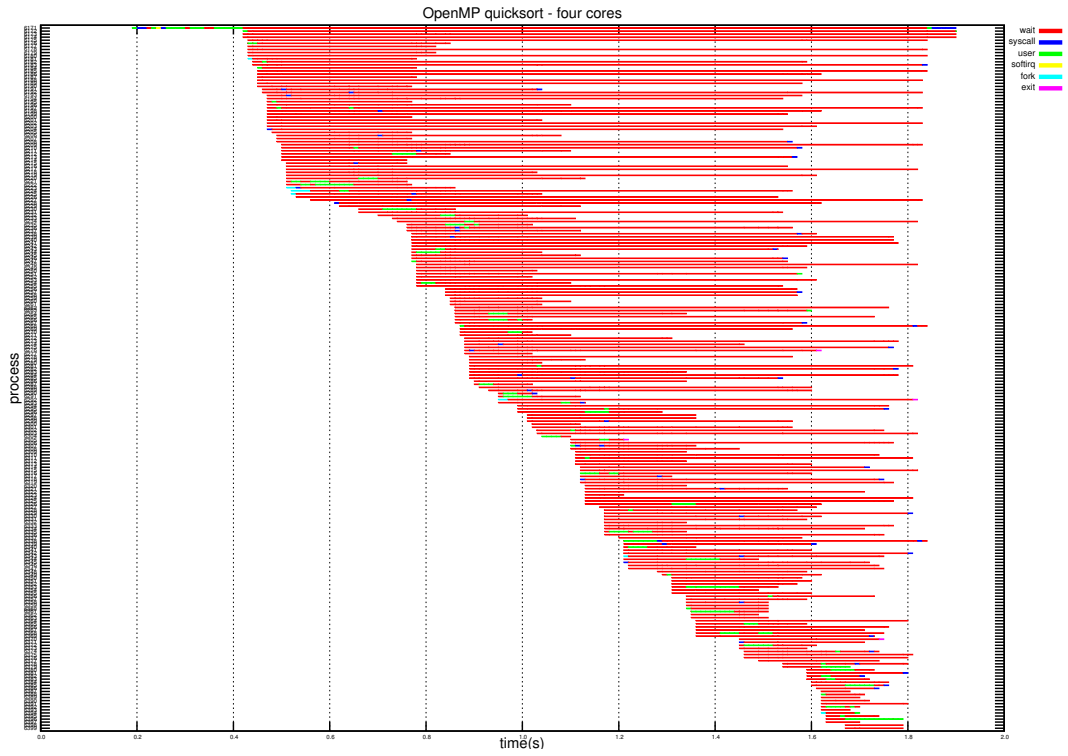

Figure 5.3: *OpenMP quicksort profiled running on four threads.*

on parallelizing the algorithm. When run on four threads (figure 5.3) the amount of `omp parallel` scopes was so large that the ARM11 MPCore ran out of memory on larger data sets. Therefore, the size of the sortable array was set to one hundred thousand (100000) as the preliminary measurements suggested. Furthermore, most of the parallelization effort goes to supporting the appalling amount threads and the sorting efficiency gained is almost next to nothing as the algorithm sorts the data set in just a slightly shorter time then when run on a single core.

Nevertheless, the simplicity and ease of applying support for parallel execution via OpenMP has its merits and the result is still a decrease in the execution time versus when running the algorithm only on one processor. Therefore OpenMP can be considered very usable for a different type of parallel problem.

The result for the measurement was exciting as the obtained trace graph acts almost as a self-explanatory view for the problems experienced. The memory shortage and the poor efficiency of the algorithm are apparent at once. Combined with some knowledge of OpenMP and the source code of

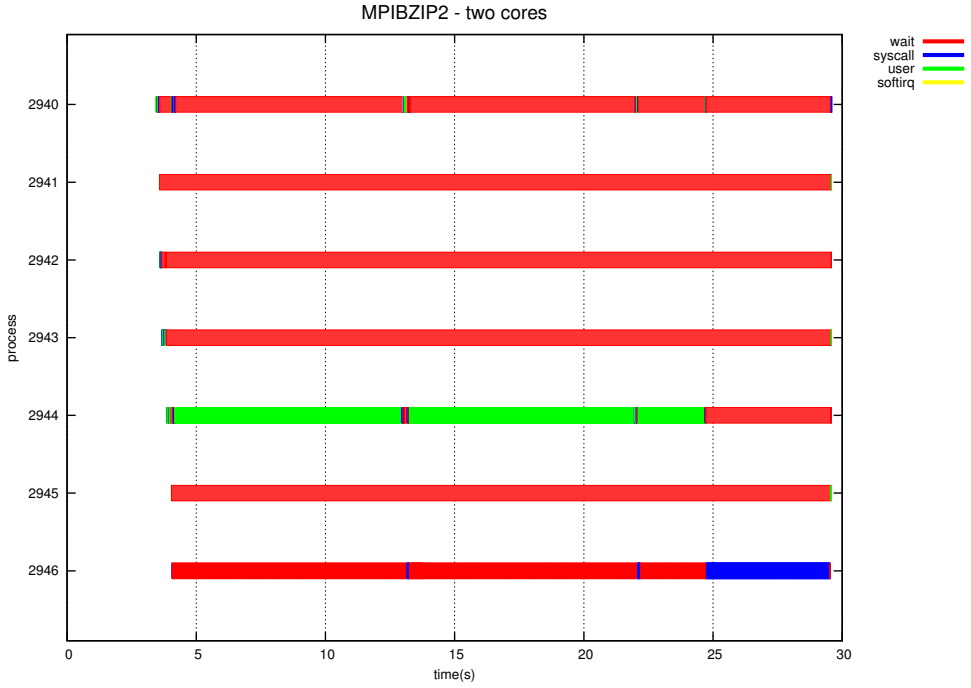

Figure 5.4: *Parallel BZIP2 (MPIBZIP2) profiled running on two threads.*

the workload, the problems can be pinpointed accurately without a doubt.

5.2.3 Data compression - parallel BZIP2

The MPIBZIP2 [17] is a parallel implementation of the bzip2 [30] an open source block-sorting file compressor. The measurements of the MBIBZIP2 workload were done on the ARM11 MPCore system (section 4.1.1) using a 2 MB file of randomly generated ASCII data as the input for compression. Random data has poor compression rate, but the comparison of compression efficiency was not on trial in these measurements.

The measurements of the MPIBZIP2 workload were done with two and then on four cores. The progress and basic events of the workload program can be seen from the resulting LTTng profiling images of figure 5.4 and figure 5.5. MPIBZIP2 starts a controlling thread that divides the file into blocks and passes them to the computing threads. The presence of multiple threads is due to the hybrid nature of the workload program as it uses both MPI and POSIX threads. File writing seems to be dedicated to the last thread,

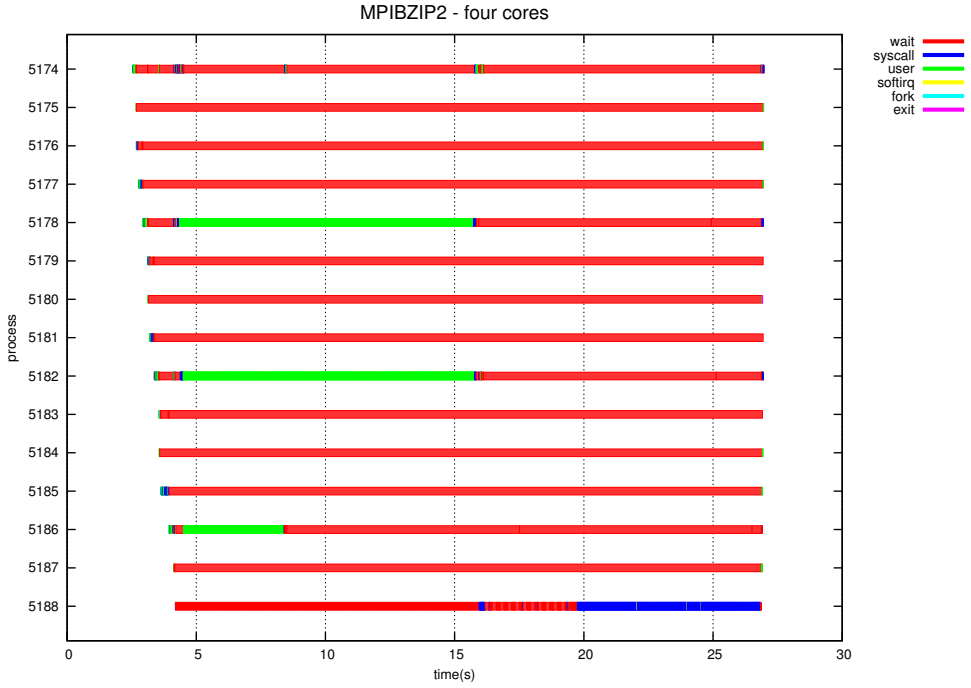

Figure 5.5: *Parallel BZIP2 (MPIBZIP2) profiled running on four threads.*

as it spends a lot of time requesting system calls after the blocks have been compressed successfully. Other POSIX threads act only as helper threads for some minor chores, since they do very little but wait most of the execution time.

At first look, the parallelization of BZIP2 appears to be somewhat naïve. The data distribution seems imbalanced, as seen on figure 5.5 where the workload is run on four cores: only three of them seem to participate in the compression calculation and the data distribution can be considered as unbalanced. However, when taking a closer look at the source code of MPIBZIP2, the nature of the unbalance becomes evident and is explained by the fact that BZIP2 uses a file block size of 900 KB by default and as the file size of the data used for compression is just 2 MB, there is not that much work for one of the compression threads.

Nevertheless, the use of a control thread that utilizes a whole processing node instead of participating in the raw computing of the compression seems like a waste of processing time, as is seen in the case when only two cores are used in figure 5.4. Here the other core gets to do all the hard work by compressing the file in three different blocks. This can be regarded as a

scalability issue when operating only on a few cores, but as MPI is usually deployed on clusters with a considerable amount of computing nodes, it can be disregarded and the need of a controlling thread might be more apparent then.

5.2.4 Data compression - parallel GZIP

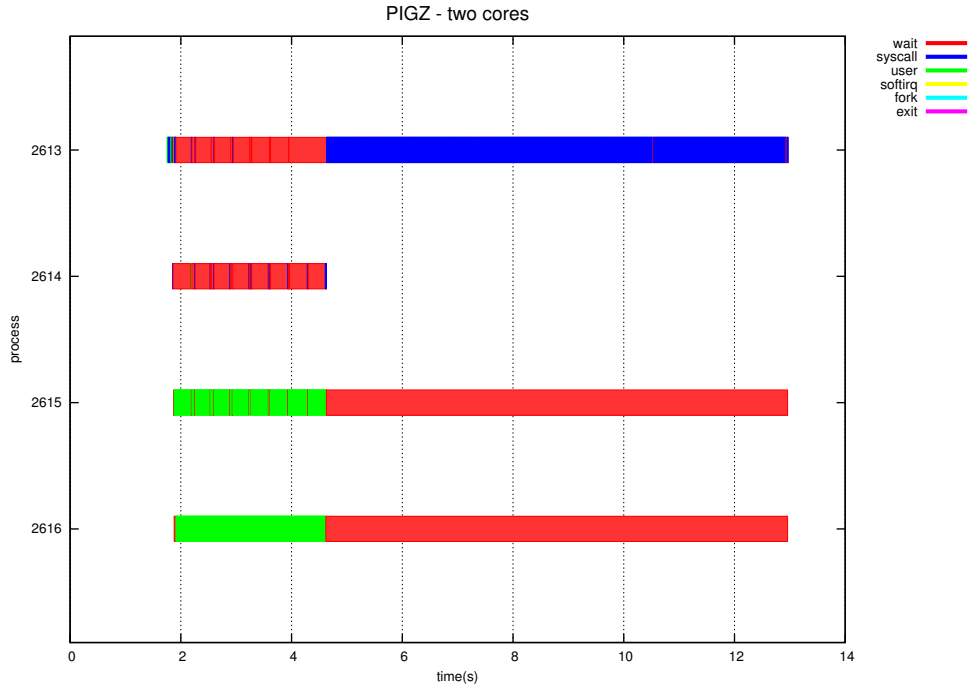

Figure 5.6: *Parallel GZIP (PIGZ) profiled running on two threads.*

Parallel GZIP (PIGZ) [1] is also a popular parallel data compression utility, similar to MPIBZIP2 on section 5.2.3. The difference regarding to the experiment is that the parallelization of PIGZ is implemented entirely with POSIX threads. The measurements were carried out on the ARM11 MPCore system described on section 4.1.1 using the same 2 MB file of randomly generated ASCII data as the input for compression as in the MPIBZIP2 measurement.

In similar fashion to the MPIBZIP2 experiment, the input data was first compressed with PIGZ using two cores and then with four cores. The progress and events of the execution of these measurements are shown in the resulting LTTng profiling images on figures 5.6 and 5.7. PIGZ breaks the input file into 128 KB chunks and compresses them parallel. The eight chunks per

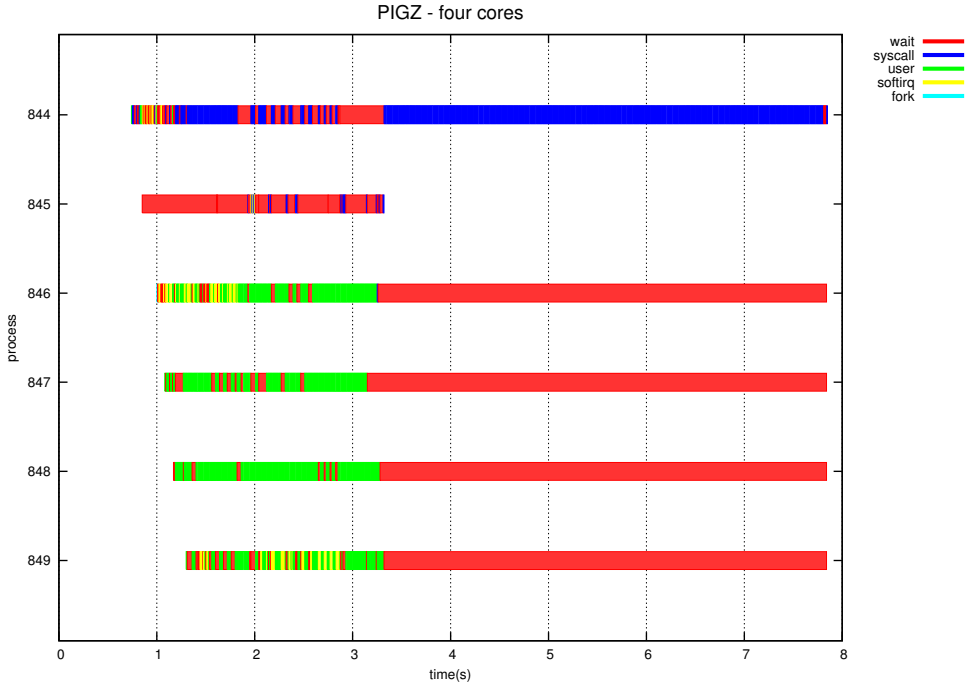

Figure 5.7: *Parallel GZIP (PIGZ) profiled running on four threads.*

compressing thread (when running on two threads) are evident from figure 5.6. Finally, the compressed data is written to the output as can be seen from the long SYSCALL state of the controlling thread.

PIGZ is an example of a parallel workload that does exactly as one would think it would: as figure 5.7 shows, it is compressing on every processing node and distributes the workload evenly. In addition, when compared to figure 5.6, it can be seen to scale nicely from two processing threads to four.

5.2.5 x264 video encoding

The x264 [38] is a library for encoding video streams. For example, the popular VLC multimedia player [35] uses it for encoding. The experiment was measured on the ARM11 MPCore system (section 4.1.1) where the x264 library was used to encode 50 frames of YUV images to a H264 video. The measurements were done first using two cores and then on the full four cores, the resulting LTTng images are on figures 5.8 and 5.9.

The run-time behavior of x264 video encoding is most evident in figure 5.9.

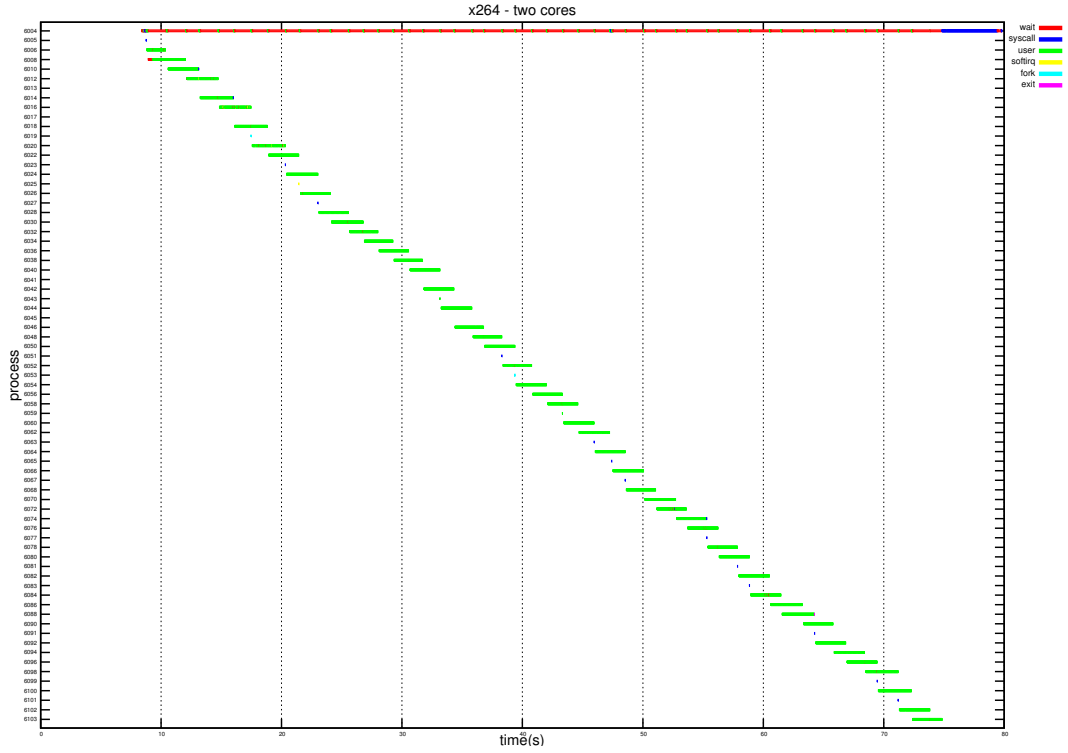

Figure 5.8: *x264 video encoding profiled running on two threads.*

The encoding is supervised via a control thread that starts a thread to encode a frame whenever there are available processing nodes. Interestingly, the threads are not used again, on the contrary a new encoding thread is started for every encoded frame. However, there are four threads encoding on any given moment and the encoding workload can be considered as evenly distributed among the processing units. The controlling thread writes the newly encoded video to the output as seen on the system call it makes after the last frame has been encoded.

Due to the nature of the encoded data, as it is easily split into autonomous pieces, a balanced distribution of the workload is achieved without difficulty. Moreover, the algorithm scales well as expected as can be seen from figure 5.8 where the x264 encoding is run on only two threads: here the control thread keeps the two processing units utilized similarly as in the four core version (figure 5.9).

The result of this experiment was a bit surprise at first, as the thread count became so large. All the same, the decision for creating a thread every time a frame is being encoded might be well considered as inter-thread communi-

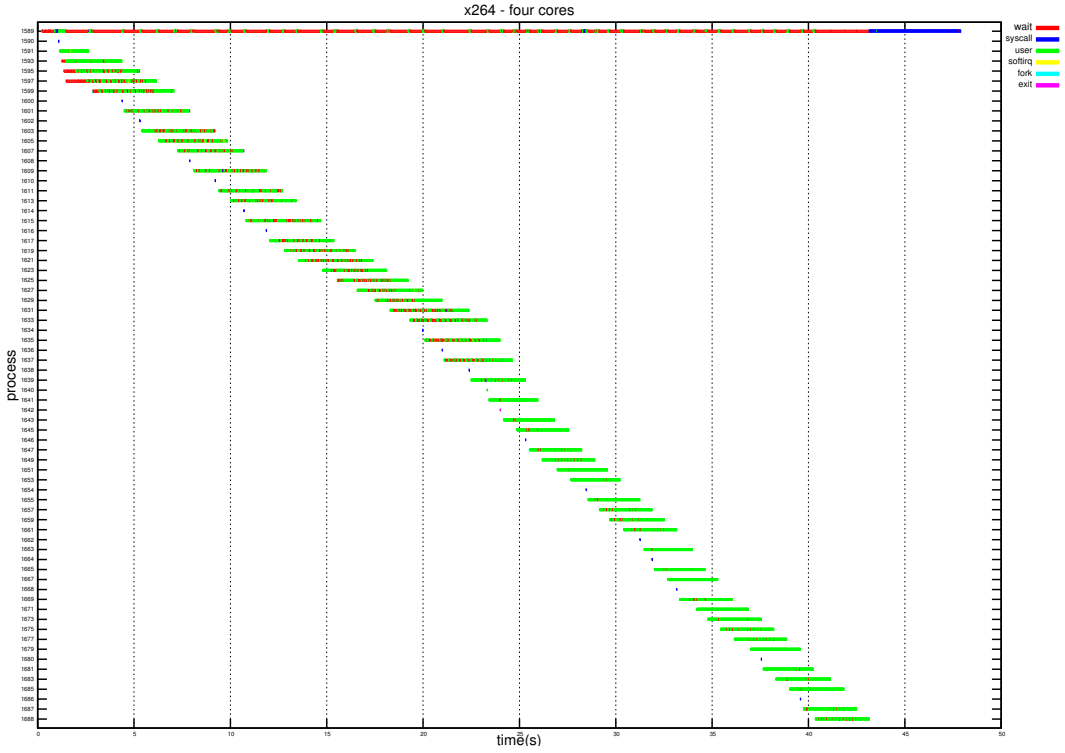

Figure 5.9: *x264 video encoding profiled running on four threads.*

cations can be difficult sometimes.

5.3 Conclusion

Setting up the measurement environment and measuring the workloads had its difficulties especially with the insufficient LTTng trace buffers on the Intel Atom cluster. Nevertheless, there were no major show-stoppers and profiling results were obtained for every workload measurement devised. The presented workloads, the two quicksort programs prepared for the experiments and the three other parallel programs, were profiled with the LTTng presenting interesting results. Therefore, setting up the LTTng to any architecture supporting Linux should be conceivable.

Most notably, the LTTng profiling revealed the cause of the inefficiency of the OpenMP quicksort solution and its memory consumption problems in section 5.2.2. Furthermore, the pondered uneven distribution of sortable items was

instantly evident from the results of the MPI quicksort in section 5.2.1.

The profiling of the other workloads, written by someone else, was also enlightening. The LTTng profiles seemed surprising at first glance, but at closer inspection together with some knowledge acquired from the source code these anomalies proved to be features of the programs.

Therefore, the LTTng profiling can be seen as quite useful, and the resulting graphs give a lot of needed information about the execution of a parallel program. However, the graphs do not reveal the inter-process communication between the threads, and thus it is left for the observer to interpret the behavior further by deciphering the source code.

6. Conclusions

This Master's Thesis described the practical setup for the implementation of multi-core and multi-processor system activity measurement and visualization. In this context, the system activity under inspection is the activity and interaction of hardware and software threads, and how the execution of a parallel program is mapped into them.

The featured multi-processor platforms were built for the preparation of future research on MPSoCs that were devised to be the basis of the next platform architecture in mobile devices. The MPSoCs featured for the mobile market are composed of a general purpose processor enclosed with special purpose processors like DSP and GPU. Therefore, two different system setups were built composing of an ARM based multi-core system acting as the general purpose processor and an Intel based multi-processor system for simulating the processing element interconnection.

The two implementations of quicksort programs as workloads were selected to show the complexity of parallelization of a well-known and understood algorithm. Furthermore, I selected workload applications depicting parts of some real-life mobile device usage where parallel execution is beneficiary.

The measurements show the complexity of applying parallelism into practice and that the available programming models are far from providing easily adaptable abstraction. Nevertheless, the LTTng tool used for visualizing the system activity is impressively useful and gives a valuable view of the parallel behavior "under the hood" when otherwise there would be no clue about what is happening.

I conclude that the LTTng tool is useful for apprehending the different states of execution of the measured program. Moreover, the balance of parallelization of the program can be established from the measurement graphs and conclusions about the overall performance can be drawn. However, there

is no way to understand the inter-process communication of the program threads from the graphs without consulting the source code and debugging.

Also, I see the LTTng as an applicable and easily applied tool for measuring the distribution and balance of parallel execution on MPSoCs, provided that the different processing elements of the MPSoC run Linux.

Future prospects of parallel programming are clear and one can already state that *the future is parallel*. At least in computing where the traditional processor architecture has met its limits and the hardware providers have been making multi-core processors for quite some time now. Moreover, more complex architectural solutions of multi-processor systems are under development and multi-core and multi-processor systems are becoming increasingly popular in mobile devices. The current trend of energy efficiency will continue to be the leading attraction that will further increase the demand of heterogeneous MPSoCs in the mobile device market.

Therefore the need for understanding the behavior of parallel programs will continue to be fundamentally important for evaluating potential parallel software solutions and reach these potential increases in performance and in energy efficiency.

Hence, the future development needs of the LTTng is the representation of inter-process communication on the measurement graphs and better support for multi-processor tracing. With these improvements and easier introduction the LTTng can become a formidable programming tool for parallel programming.

Bibliography

[1] Mark Adler. PIGZ - A parallel implementation of gzip for modern multiprocessor, multi-core machines. `http://www.zlib.net/pigz/`. (Referenced May 3, 2010).

[2] Thomas William Ainsworth and Timothy Mark Pinkston. Characterizing the Cell EIB On-Chip Network. *IEEE Micro*, 27(5):6–14, 2007.

[3] ARM11 MPCore Processor Technical Reference Manual, revision r1p0, 2008.

[4] Mark Baker. Cluster computing white paper. *IEEE Computer Society Task Force on Cluster Computing (TFCC)*, cs.DC/0004014, 2000.

[5] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of Network-on-Chip. *ACM Comput. Surv.*, 38(1):1, 2006.

[6] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 555–563, New York, NY, USA, 1999. ACM.

[7] David Brash. The ARM Architecture Version 6 (ARMv6), 2002.

[8] Intel Corporation. Intel Desktop Board D945GCLF Technical Product Specification, April, 2008.

[9] Intel Corporation. Intel Atom Processor 200 Series Datasheet, June, 2008.

[10] H J Curnow, B A Wichmann, and Tij Si. A synthetic benchmark. *The Computer Journal*, 19:43–49, 1976.

[11] William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 684–689, New York, NY, USA, 2001. ACM.

[12] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, 2005.

[13] John L. Hennessy David A. Patterson. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann publishers, Inc, 1997.

[14] Ulrich Drepper. What every programmer should know about memory. `http://people.redhat.com/drepper/cpumemory.pdf`, November 2007. (Retrieved Aug 14, 2008).

[15] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972.

[16] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[17] Jeff Gilchrist. MPIBZIP2. `http://compression.ca/mpibzip2/`. (Referenced May 3, 2010).

[18] Argonne National Laboratory Group. MPICH2. `http://www.mcs.anl.gov/research/projects/mpi/mpich2/`. (Referenced May 3, 2010).

[19] Raj Jain. *The Art of computer systems performance analysis*. John Wiley & Sons, inc., 1991.

[20] C. R. Johns and D. A. Brokenshire. Introduction to the Cell Broadband Engine Architecture. `http://www.research.ibm.com/journal/rd/515/riley.html`, 2007.

[21] R. P. Kar and K. Porter. Rhealstone: A real-time benchmarking proposal. *Dr. Dobbs Journal*, 14:14–24, 1989.

[22] Rabindra P. Kar. Implementing the rhealstone real-time benchmark. *Dr. Dobb's J.*, 15(4):46–55, 1990.

[23] Kprobes. Kernel Dynamic Probes. `http://sourceware.org/systemtap/kprobes/`. (Referenced May 3, 2010).

[24] LTTng. Linux Trace Tool, next generation. `http://lttng.org/`. (Referenced May 3, 2010).

[25] M. Desnoyers, M. R. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of Ottawa Linux Symposium 2006*, pages 209–224, 2006.

[26] OpenMP. Open Multi-Processing. `http://www.openmp.org/`. (Referenced May 3, 2010).

[27] M. Paganini. Nomadik: A Mobile Multimedia Application Processor Platform. In *ASP-DAC '07: Proceedings of the 2007 conference on Asia South Pacific design automation*, pages 749–750, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Katalin Popovici, Xavier Guerin, Frederic Rousseau, Pier Stanislao Paolucci, and Ahmed Amine Jerraya. Platform-based software design flow for heterogeneous MPSoC. *Trans. on Embedded Computing Sys.*, 7(4):1–23, 2008.

[29] RealView Platform Baseboard for ARM11 MPCore User Guide, 2008.

[30] Julian Seward. bzip2 - a program and library for data compression. `http://www.bzip.org/`. (Referenced May 3, 2010).

[31] SPEC. Standard Performance Evaluation Corporation. `http://www.spec.org/`. (Referenced May 3, 2010).

[32] Inc. Sun Microsystems. DTrace. `http://www.sun.com/bigadmin/content/dtrace/`. (Referenced May 3, 2010).

[33] SystemTap. `http://sourceware.org/systemtap/`. (Referenced May 3, 2010).

[34] Andrew S. Tanenbaum. *Modern Operating Systems, 2nd Edition*. Prentice Hall, 2001.

[35] VideoLAN. VLC media player. `http://www.videolan.org/`. (Referenced May 3, 2010).

[36] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.

[37] Nelson Weiderman. Hartstone: synthetic benchmark requirements for hard real-time applications. In *Proceedings of the working group on Ada performance issues 1990*, pages 126–136, New York, NY, USA, 1990. ACM.

[38] x264. x264 - a free h264/AVC encoder. `http://www.videolan.org/developers/x264.html`. (Referenced May 3, 2010).