AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY
Faculty of Electronics, Communications and Automation
Department of Signal Processing and Acoustics

Aleksi Aalto

Dynamic management of multiple operating systems in an embedded multi-core environment

Master's Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.

Espoo, May 7, 2010

Supervisor: Professor Jorma Skyttä
Instructors: Professor Tatsuo Nakajima and D.Sc. (Tech) Vesa Hirvisalo

AALTO UNIVERSITY SCHOOL OF SCIENCE AND TECHNOLOGY	ABSTRACT OF THE MASTER'S THESIS

Author: Aleksi Aalto

Name of the Thesis: Dynamic management of multiple operating systems in an embedded multi-core environment

Date: May 7, 2010 **Number of pages:** xi + 69

Faculty: Faculty of Electronics, Communications and Automation

Professorship: S-88 Signal Processing

Supervisor: Prof. Jorma Skyttä

Instructors: Prof. Tatsuo Nakajima and D.Sc. (Tech.) Vesa Hirvisalo

Modern embedded devices, such as smartphones, have grown into complex computer systems that provide a rich set of functionality for their users while still maintaining real-time responsiveness for their low level functions such as radio communication or camera control. The embedded market is very competitive, especially in end-user mobile devices, making it desirable to reduce manufacturing costs without compromising device performance wherever possible. The ever-growing user demand for more computing-intensive applications coupled with tight energy budgets has led the embedded manufacturers to seek performance gains from multi-core architectures, much like their desktop counterparts. However, multi-core architectures have little to provide in performance gains when used with applications developed with traditional software design methods that are aimed at single-core archictures.

This thesis presents a system-level solution for effectively using the parallel computing power of multi-core processors. DynOS SPUMONE, a concept of using a light weight virtualization layer to dynamically dispatch different OSes on different cores, was developed. The concept is to run real-time tasks, such as device control for peripherals, on real-time capable operating systems running on dedicated cores only when they are actually needed. This could be used to eliminate separate physical chips on the device, which would reduce manufacturing costs. A prototype implementation of DynOS SPUMONE was built, verified and evaluated.

The results show that the DynOS SPUMONE concept is realizable with reasonable engineering costs and without significant drops in real-time performance.

Keywords: embedded systems, real-time, virtualization, operating systems, Linux

AALTO-YLIOPISTO	DIPLOMITYÖN
TEKNILLINEN KORKEAKOULU	TIIVISTELMÄ

Tekijä:	Aleksi Aalto		
Työn nimi:	Monen käyttöjärjestelmän dynaaminen hallinta sulautetussa moniydinjärjestelmässä		
Päivämäärä:	7. toukokuuta 2010	**Sivuja:**	xi + 69
Tiedekunta:	Elektroniikan, tietoliikenteen ja automaation tiedekunta		
Professuuri:	S-88 Signaalinkäsittely		
Työn valvoja:	Prof. Jorma Skyttä		
Työn ohjaajat:	Prof. Tatsuo Nakajima ja TkT Vesa Hirvisalo		

Modernit sulautetut laitteet, kuten älypuhelimet, ovat kasvaneet monimutkaisiksi tietokonejärjestelmiksi, jotka tarjoavat samaan aikaan niin rikasta käyttäjäkokemusta kuin reaaliaikaista suorituskykyä alemman tason laitteille, kuten kameralle tai radiolle. Kilpailu sulautettujen järjestelmien markkinoilla on kovaa, etenkin loppukäyttäjille myytävissä mobiililaitteissa, mikä johtaa tarpeeseen vähentää laitteiden valmistuskustannuksia vaikuttamatta laitteen suorituskykyyn. Pöytäkoneiden markkinoilla jo pitkään tapahtunut siirtyminen moniydinsuorittimen käyttöön on viime aikoina alkanut tapahtua myös sulautetuissa järjestelmissä, joiden haasteena on jatkuvasti kasvava vaatimustaso suorituskyvylle ja toisaalta taas tiukat rajoitukset energiankäytölle. Moniydinsuorittimista ei kuitenkaan saada toivottua suorituskyvyn lisäystä, jos ohjelmistokehitystä jatketaan vanhoilla, yksiydinsuorittimille tarkoitetuilla toimintatavoilla.

Tässä työssä esitellään systeemitason ratkaisu moniydinprosessorien rinnakkaisen laskentavoiman tehokkaaseen käyttöön. Työssä kehitettiin ratkaisu nimeltä DynOS SPUMONE, jonka perustana on käyttää kevyttä virtualisointikerrosta ajamaan samanaikaisesti eri käyttöjärjestelmiä moniydinprosessorin eri ytimillä tarpeen mukaan. Ideana on ajaa tarvittaessa reaaliaikaista suorituskykyä vaativat ohjelmat omalla ytimellään käyttäen reaaliaikakäyttöjärjestelmää. Ratkaisua voitaisiin käyttää säästämään sulautettujen laitteiden valmistukuluissa poistamalla nykyisen tarpeen käyttää erillisiä piirejä ajamaan reaaliaikasovelluksia. Työssä kehitettiin myös DynOS SPUMONE:en perustuva prototyyppi, joka verifikoitiin ja arvioitiin.

Työn tulokset osoittavat DynOS SPUMONE:en pohjautuvien ratkaisujen olevan toteutettavissa erittäin kohtuullisin suunnittelukustannuksin ilman mainittavaa vaikutusta systeemin reaaliaikaiseen suorituskykyyn.

Avainsanat: sulautetut järjestelmät, reaaliaika, virtualisointi, käyttöjärjestelmät, Linux

Acknowledgments

The work described in this thesis was made in the Distributed and Ubiquitous Computing Laboratory (DCL) of Waseda University during 2008-09. I wish to offer my sincere thanks to Professor Tatsuo Nakajima for making this unforgettable experience possible.

I am very grateful to Dr. Vesa Hirvisalo for tirelessly guiding me through the whole process and to Professor Jorma Skyttä for always patiently supporting the work.

I would like to offer warm thanks to everyone at DCL for making my year there a great one. I am especially thankful to Wataru Kanda, Yuki Kinebuchi, Hiroo Ishikawa, Yu Yumura and Hiromasa Shimada for their great support and friendship.

Many great friends kindly provided much needed help for the writing process. I am particularly grateful to Antti Ukkonen and Kalle Korhonen for offering a lot of insightful feedback and to Heli Virtanen, Jussi Judin, Marko Crivaro and Tommi Salminen for proofreading.

I am also very thankful to Vili Lehdonvirta for all of his help during the whole process. Further, the external pressure provided courtesy of Pyry Lehdonvirta was an enormous help for me, for which I wish to thank warmly.

The writing of this thesis proved to be a long and a hard process that I could not have finished without the support and encouragement always offered by my family and relatives, godparents, friends and my Mai. For this I would like to express my deepest gratitude.

Kivenlahti, May 7, 2010

Aleksi Aalto

Contents

1 Introduction **1**
1.1 Problem . . . 2
1.2 Contribution . . . 2
1.3 Related work . . . 5
1.4 Thesis structure . . . 7

2 Background **8**
2.1 Embedded systems . . . 8
2.2 Multi-core processors . . . 14
2.3 Operating systems . . . 20
2.4 Virtual machines . . . 27

3 Building blocks of the system **35**
3.1 SPUMONE . . . 35
3.2 RP1 - multi-core processor . . . 40
3.3 SH-Linux with CPU-hotplug . . . 45
3.4 TOPPERS/JSP . . . 49

4 Implementation **50**
4.1 DynOS SPUMONE . . . 50
4.2 Changes to SPUMONE . . . 54
4.3 Changes to Linux . . . 55
4.4 DynOS kernel module for Linux . . . 56

5 Verification and evaluation **57**
5.1 Verification . . . 57

5.2 Performance 58
5.3 Engineering costs 60

6 Conclusions **62**
6.1 Discussion 62
6.2 Proposals for future work 64

Bibliography **65**

List of Figures

1.1 A use case of the DynOS SPUMONE concept. 4
1.2 Replacing a separate camera chip with a multi-core DynOS SPUMONE solution. 5

2.1 Design information flow for embedded systems. [31] 11
2.2 Relative speedup with various sequential proportions. 16
2.3 Kernel organization. 23
2.4 Main computer system interfaces. [44] 31

3.1 A system running on SPUMONE. [23] 37
3.2 A system running on multi-core SPUMONE. [22] 39
3.3 Block diagram of the RP1. [39] 41
3.4 SH-Linux multi-core boot-sequence. 47

4.1 The basic operation flow of DynOS SPUMONE. 51
4.2 Architectural hierarchy of the prototype DynOS SPUMONE. 53

5.1 Verification test setup. 58

List of Tables

3.1 CPU-maps of SH-Linux. 46

4.1 DynOS user functions. 52

5.1 Linux core-hotplug execution times on RP1. 59
5.2 Timer interrupt handling delay in TOPPERS/JSP on a SH-4A. [24] 60
5.3 Code changes per guest OS for multi-core SPUMONE. [22] 61
5.4 Code changes per component for DynOS SPUMONE. 61

Acronyms

ABI Application Binary Interface.

ADC Analog-to-Digital Converter.

AMP Asymmetric Multiprocessing.

API Application Programming Interface.

ASCII American Standard Code for Information Interchange.

ASIC Application-Specific Integrated Circuit.

ATX Advanced Technology Extended.

BIOS Basic Input/Output System.

BKL Big Kernel Lock.

CIS Car Information System.

CPU Central Processing Unit.

DDR Double Data Rate.

DSM Distributed Shared-Memory.

DSP Digital Signal Processor.

FLOPS Floating-Point Operations Per Second.

FPGA Field-Programmable Gate Array.

FPU Floating-Point Unit.

GPOS General-Purpose Operating System.

HLL High-Level Language.

IP Intellectual Property.

IPC Instruction Per Cycle.

IPI Inter-Processor Interrupt.

ISA Instruction Set Architecture.

JVM Java Virtual Machine.

LED Light-Emitting Diode.

LOC Lines Of Code.

MCU Micro-Controller Unit.

MIMD Multiple instruction streams, multiple data streams.

MIPS Millions of Instruction Per Second.

MISD Multiple instruction streams, single data stream.

MMU Memory Management Unit.

NMI Non-Maskable Interrupt.

NUMA Nonuniform Memory Access.

OS Operating System.

PC Personal Computer.

PCB Printed Circuit Board.

PLL Phase-Locked Loop.

RISC Reduced Instruction Set Computer.

ROM Read-Only Memory.

RTOS Real-Time Operating System.

SIMD Single instruction stream, multiple data streams.

SISD Single instruction stream, single data stream.

SMP Symmetric Multiprocessing.

SRAM Static Random Access Memory.

TLB Translation Lookaside Buffer.

UMA Uniform Memory Access.

USB Universal Serial Bus.

VCPU Virtual Central Processing Unit.

VM Virtual Machine.

VMM Virtual Machine Monitor.

Chapter 1

Introduction

Until recently, advances in computer technology have been relatively easy to apply from the software point of view. Progress has been made by improving old technologies thus eliminating the need for fundamental changes in software. Semiconductor manufacturing processes have been getting better, as depicted by Moore's law, providing steady growth in computing performance for decades without having to touch the fundamental concepts of computer architecture. Alas, this road has come to its end as the conventional solutions cannot anymore be improved significantly by enhancing the manufacturing technology.

Although the operating frequencies of processors cannot anymore be raised with reasonable cost, many processors can now be placed on the same chip for almost the same price as one. For some years, the trend in desktop computers has been multi-core processors. This is quickly becoming reality also in the embedded domain. Especially high-end embedded systems, like mobile phones, show no saturation in demand for more and more computing power to serve applications that are steadily getting more complex and rival already anything that was available on the best desktop machines only a few years ago. Coupled with hard constraints for energy usage, the acute need for new solutions that would maintain the performance growth in embedded systems is leading the industry to widely adopt multi-core processors in near future.

What is not yet that well established, is how to make good use of this sudden new computing power that is overtly parallel. Programming models have not kept up with the development of computer hardware and the basic programming flow is still fairly straightforward and strictly sequential. Concurrent programming models have not become mainstream except for a narrow sector of specialized applications even though they have been around for already decades. The problem is that with the existing programming models, most of the applications do not parallelize very well. One way to increase the practical effectiveness of the extra parallel computing power is to dedicate certain cores for certain functions instead of parallelizing single processes. Modern operating systems are already capable of running independent

processes simultaneously on different cores. Taking this one step further, cores could also be dedicated to certain operating systems.

But why run different operating systems on one chip? The answer is better utilization of the systems resources and a more cost-effective manufacturing process. In modern embedded systems, there are already multiple operating systems running parallel in the device. They are just located on different physical chips. Having multiple physical chips in a device costs a lot to manufacture. Integrating some of the chips together into one chip would cut manufacturing costs. There is however a reason for using multiple chips. The functions performed by the different chips and their operating systems are various and require dissimilar features from the processors and operating systems. E.g. on a mobile phone one chip can be responsible for the radio communications, requiring real-time responsiveness and thus a real-time operating system. At the same time another chip could take care of user interface and running user applications, benefiting from a rich software infrastructure provided by a general-purpose operating system like Linux. Combining these functions to run on the same physical chip would reduce manufacturing costs, but the design of such a custom chip could still be expensive. A way to use off-the-shelf multi-core processors for both real-time and general-purpose tasks would be the most cost-effective solution.

1.1 Problem

When integrating diverse functionalities into one multi-core chip it is obvious that one operating system alone will not be capable to meet all of the contradicting requirements. There is clearly a need to use multiple operating systems at the same time. This could be achieved by using virtualization methods. Traditional virtualization methods, which have been successfully used already for decades in the server domain, are nevertheless not well suited for embedded applications that have strict power and real-time performance constraints. A virtualization system aimed for embedded devices should thus have very light virtualization costs and it should enable dynamical assignment of the processor cores to different operating systems. **The goal of this thesis work is to find out if it is feasible to build such a system with reasonable engineering costs**.

1.2 Contribution

In this thesis we present DynOS SPUMONE, a concept of using the cores of an embedded multi-core processor effectively by making it possible to dynamically assign different operating systems to the processor cores. This gives the system programmer the opportunity to use the most suitable operating system for the task at hand. Whereas real-time processing capability or very low overheads are needed,

the general-purpose operation system can be laid aside from some core and replace it momentarily with a real-time operating system. After the need for special computing has ended the real-time operating system can be shut down again and the general-purpose operating system can regain control of the core.

The DynOS SPUMONE concept is best illustrated by a simple use case. In figure 1.1 the first picture, phase 1, shows a processor with four cores. A general-purpose operating system (GPOS) has populated all of the four cores and is running various applications on them. Then in Phase 2, applications with hard real-time requirements need to be run on a real-time operating system (RTOS). The GPOS starts migrating its running application from Core 3 to other cores. After the migration has finished, the GPOS removes itself from the core and the RTOS and its applications are launched, leading to the situation depicted in Phase 3. When the need for real-time processing has ended (Phase 4), the RTOS in turn shuts itself down and the GPOS is called to regain control of Core 3. In the end (Phase 5) the GPOS has again control of all the cores.

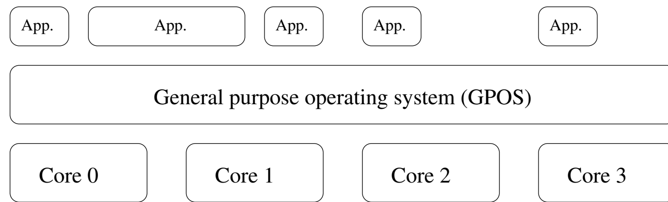
Example application concept
Mobile phone with an embedded camera

A modern mobile phone that is equipped with a camera is a great example how using the DynOS SPUMONE concept could ease design and reduce manufacturing costs. Today, these phones utilize a specialized chip to control the camera cell. In addition to the general-purpose operating system (GPOS) that controls the main functionality of the phone, a real-time operating system (RTOS) is used on the camera chip to meet the hard real-time constraints of operating the camera device. This naturally increases both design and manufacturing costs, as there is at least one extra chip to be soldered and the printed circuit board (PCB) has to be designed more carefully to fit all the necessary components in the very confined space that is available inside a modern mobile phone.

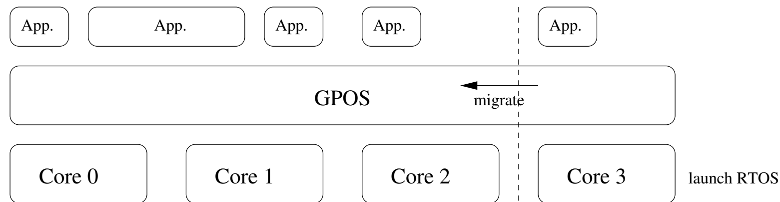
These costs could be reduced by using the DynOS SPUMONE concept. The camera of a mobile phone is not used all the time and when not used, the computing power of the extra chip is in vain. As multi-core processors become mainstream also in mobile phones, the camera chip could be left out and replaced by using one core of the multi-core processor for the camera only when needed, as illustrated in figure 1.2. This way no extra chips would have to be added to the PCB and the computing power now reserved only for the camera would be available for any applications.

To obtain this, a system like DynOS SPUMONE is needed. Laying between the multi-core processor and the operating systems, DynOS SPUMONE can launch the RTOS needed to control the camera on one core when the camera is needed and let the GPOS control the core at other times. In cases where multiple cores are available, the RTOS could be run on multiple cores to enable parallel processing of the camera data thus speeding it up. GPOS could even be suspended during camera operation

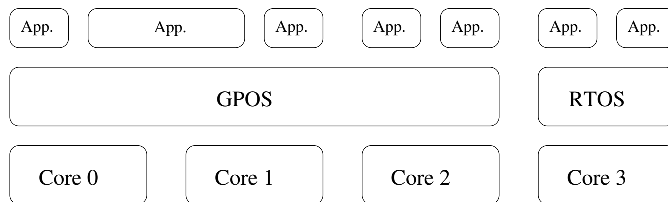
Phase 1 – Initial stage. GPOS controls all resources.



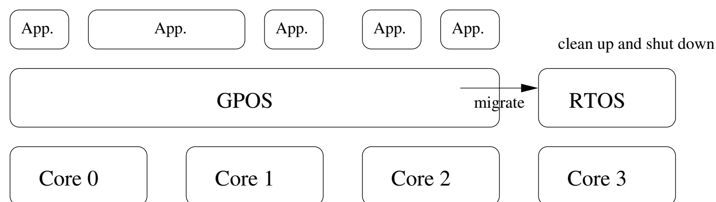
Phase 2 – GPOS migrates applications away from Core 3 and RTOS is launched.



Phase 3 – GPOS and RTOS run simultaneously.



Phase 4 – RTOS has finished its tasks and shuts down. GPOS regains control of Core 3.



Phase 5 – GPOS controls again all resources.

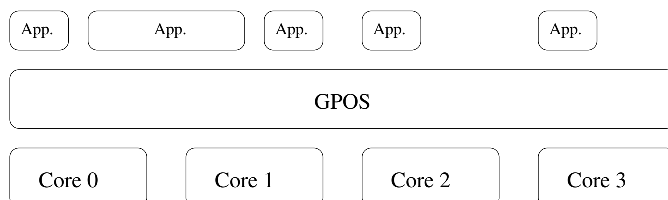


Figure 1.1: A use case of the DynOS SPUMONE concept.

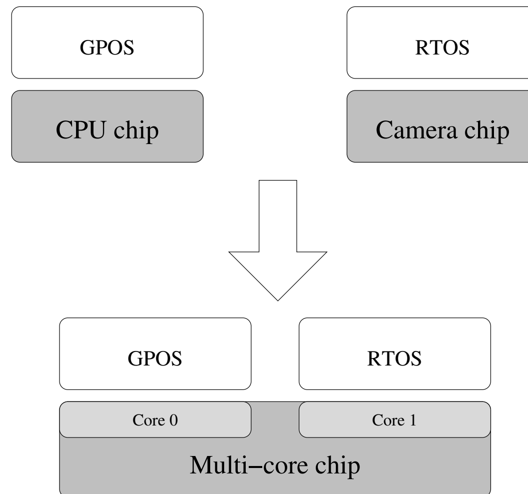


Figure 1.2: Replacing a separate camera chip with a multi-core DynOS SPUMONE solution.

and be revoked only by interrupts or ending of the camera operations. The concept offers many ways for the developer to use the available hardware more flexibly.

1.3 Related work

As embedded systems are getting more popular and complex, a clear need for more efficient hardware and new software solutions has emerged. Multi-core processors seem to be the choice of tomorrow not only for conventional desktop and server computing but also for embedded systems. There is already talk of the next generation of parallel computing called *many-core architecture* where hundreds or even thousands of small cores would be fitted on one silicon die to provide an energy- and space-efficient parallel processing environment. [6]

What is not yet clearly distinguishable is how to make good use of this new processing power in embedded systems. The choice of an operating system for e.g. a modern mobile handset is increasingly difficult. Where stripped down versions of established OSes like Linux or Windows are able to provide code portability from desktop systems and a rich set of functionality, they do perform badly for real-time tasks. Real-time OSes on the other hand have no problems dealing with real-time constraints but are inherently lacking in code that would run on them without costly modifications.

One approach to gain real-time capabilities without losing the benefits of a GPOS has been to modify existing GPOS kernels to perform better for real-time tasks. ASMP-Linux, ASymmetric MultiProcessor Linux, is one of these attempts. The approach is to modify the Linux kernel in a way that would allow it to run true real-time

applications on a multi-core processor alongside normal Linux applications. The approach resembles that of DynOS SPUMONEs in that they reserve at least one core for real-time applications only. Although aiming for simplicity and performance, ASMP-Linux at its current state does not perform sufficiently well for hard real-time systems. The solutions maintainability in the long run is also questionable, as it requires some modifications to the kernels scheduler. [5]

Virtualization for embedded systems is a relatively new, but emerging field that is seen as an answer for the somewhat contradictory demands posed by modern embedded systems. The conventional wisdom about virtual machines (VM) has nonetheless been, that they can not provide correct timing even if they are para-virtualized. [36] This is naturally not acceptable for a system with real-time constraints.

Some research projects have went around this assumed limitation by constructing VMs with artificial architectures that are able to handle a set of high-level tasks with no resemblance to existing hardware architectures. This way the devices can be controlled with a minimal code base reducing both memory size requirements and the energy needed for transmission of new program code in cases dynamic reprogramming is needed. This approach is attractive for systems that have very limited memory and energy budgets, e.g. sensor networks. [30] [48] Though for more complex systems it won't be sufficient. Using custom architectures like these does not allow running common desktop software in the system and it also does not allow running multiple OSes simultaneously.

Some projects have been trying to adapt existing VMMs for embedded systems, the most noticeable being the highly popular high-level language execution environment Java Virtual Machine (JVM). There have been various approaches of bringing the flexibility of the Java-language and the real-time constraints of an embedded systems together. All of these are compromises that either cripple the implementation with proprietary real-time capable libraries or ignore the issue altogether. [19] The widely popular virtual machine monitor (VMM) Xen has also been ported to an equally popular embedded processor architecture, the ARM. [20] Although using the para-virtualization technique, the embedded Xen VMM still remains unreasonably big and resource-hungry for an embedded system and can not provide efficient enough inter-OS communication that would fulfil the needs of a true multi-OS system. [16]

There are also VMMs available that have been built from the start for complex embedded systems. Two commercial ones are the VLX developed by VirtualLogix and OKL4 that is developed by Open Kernel Labs. Both VMMs target the mobile handset market with OKL4 claimed to have an install base of 250 million mobile phones. Both of them use the paravirtualization paradigm thus having a rather high performance but requiring modifications to the guest OSes. VLX is proprietary thus not much is known about its internals. OKL4 is based on a L4 micro-kernel designed for embedded systems. Both of the VMMs aim to have only a thin layer between the hardware and the guest OSes. However they do have some extensive functionalities that may render them unsuitable for hard real-time cases where direct interaction

with the hardware would be desired. The required effort to port new OSes or new versions of already ported OSes to the VMMs is also unclear. [51] [16]

VLX and OKL4 are the closest match to the approach taken in DynOS SPUMONE. A thin virtualization layer should enable a RTOS and a GPOS to co-exist on the same hardware and to allow for dynamic partitioning between the guest OSes on a multi-core system. An approach like this seems particularly attractive for modern embedded systems that serve a range of applications ranging from hard real-time to modern user interfaces. Whereas VLX and OKL4 aim for a rather wide functionality with a reasonably small VMM, DynOS SPUMONE aims for absolute simplicity to provide no-compromise hard real-time performance with no interruptions at all. This way also the porting of new or legacy OSes will require only a minimal set of code modifications and can be completed with minimal engineering effort. A lot is left for the system designer to ensure but as embedded systems are inherently closed systems, this trade-off is seen as acceptable.

1.4 Thesis structure

In this thesis the DynOS SPUMONE concept, introduced above, is examined through implementing a prototype and discussing its features. Background information aimed at helping to understand the concept both from motivational and technical viewpoints is introduced in chapter 2. The building blocks used to implement the prototype are introduced in chapter 3 and the implementation details are presented in chapter 4. Verification and evaluation of the prototype are discussed in chapter 5. Last, conclusions of the work are shown and proposals for future work are given in chapter 6.

Chapter 2

Background

Chapter summary

In this section the basic principles and techniques that lay the ground to understand both the need for DynOS SPUMONE and it's technology are introduced. We start by discussing embedded systems, the focus domain of this work. Embedded and real-time systems are defined and embedded hardware and software are introduced. Next we introduce the target hardware platform; multi-core processors. Issues concerning their scalability and performance are discussed and hardware techniques to accommodate for them are presented.

As DynOS SPUMONE is intended to enable running dynamically multiple operating systems, also the basics of operating systems are introduced. Two operating system topics, real-time operating systems and Linux with its kernel modules, are examined more thoroughly in order to aid in understanding the operation of the prototype implementation better. As SPUMONE itself is a very light virtual machine monitor, virtualization techniques and virtual machine monitors are also introduced.

2.1 Embedded systems

In this section embedded systems are introduced. The DynOS SPUMONE concept is aimed for today's embedded systems that range from traditional devices with very constrained resources to multimedia-capable devices armed with a lot of computing power. First, important definitions concerning embedded systems are introduced. Then, the market situation of embedded systems is discussed. Last, principles of designing embedded systems are introduced.

Definitions

There is no single decisive definition of embedded system albeit many have been proposed. Presented next are three definitions from standard embedded systems textbooks.

> Embedded systems are information processing systems that are embedded into a larger product and that are normally not directly visible to the user. [31]

> Embedded systems are those that are found in a system that is not itself a computer. [26]

> An embedded system is a combination of computer hardware and software - and perhaps additional parts, either mechanic or electronic - designed to perform a dedicated function. [4]

The common theme in these definitions is that an embedded system is a computer that is not a desktop computer. Embedded systems have some common characteristics. Embedded systems have to be *dependable* and *efficient*. They are *dedicated towards a certain application* and they have *dedicated user-interfaces* (as opposed to a standard desktop user-interface). Many embedded systems are hybrid systems and they are typically *reactive* (they are in interaction with the environment and execute at a pace set by that environment). In addition, embedded systems often have *real-time-constraints*. [31]

As can be seen from these definitions, embedded systems cover a remarkable range of application domains and can be found practically everywhere in the modern world. Some of the key areas where embedded systems are constantly deployed include automotive, aircraft, telecommunications, medical systems, military applications, consumer electronics and robotics. [31] The broad application spectrum means that the range of the devices also varies greatly in both computing power and price. An embedded system without hard requirements for raw processing power can do very well with just an 8-bit micro-controller unit (MCU) costing as little as 0.2$ whereas a modern gaming console can embody a processor with an architecture rivaling even desktop PC's and accordingly having a unit cost up to as much as 200$.[17]

Embedded market

Although usually invisible to the end user and mostly unheard by the general public, embedded systems constitute the majority of the microprocessor market. Raw estimates from their quota of the worldwide processor sales range from 86% [17] (based on data from the year 2000) to 99% [4]. The diversity of embedded systems can be seen also from these figures as they differ significantly from each other. Both

of the figures are very dependent of the used definition of an embedded system. The definition of a processor itself can have a significant influence on such figures. In addition a modern desktop PC contains not only the desktop processor but a wide range of embedded processors taking care of bus traffic, memory control, basic input/output system (BIOS), video and sound, etc.

Embedded systems may dominate the processor market in sheer volume of shipped units, but they are not dominating the market in terms of revenue. Looking at data from 2002 [50], only 2% of all the semiconductors produced were processors, yet they generated 30% of the total revenue of the processor market. Again 8-bit micro-controllers were by far the most popular processors with a share over 50% of all the processors shipped. However 8-bit MCUs are very cheap in comparison to more advanced processors, thus generating only 15% of the total revenue of the processor market. 32-bit embedded processors were approximated to constitute 98% of the whole 32-bit market leaving 32-bit PC-processors with only a 2% market share. Yet 32-bit PC processors were solely responsible for 50% of the revenue of the whole processor market.

Taking into consideration the growing complexity of embedded applications and also the above mentioned revenue-distribution between processor types, it is no wonder that also embedded processors are becoming more and more complex. Smartphones are a good example of a new breed of embedded systems whose design is concentrated less to interaction with the physical environment through sensors and actuators but instead encompass a wide variety of functions normally affiliated with standard desktop computers, e.g. multimedia, games and connection to the Internet. In addition to traditional challenges like real-time requirements, the new kind of mobile computing systems must find ways to deal with new challenges posed by portability, wireless communication and mobility requirements. [28] Investment bank RBC has forecast that as soon as in 2011 the sales of smartphones will surpass that of desktop PC's. [45] This kind of development has been described as disappearing computer or ubiquitous computing. [31] This underlines the development that computers are transforming from mere objects in the directions of everyday tools that are more or less invisible to the user.

Design

The design of an embedded system comprises of both hardware and software design, usually more or less simultaneously. These disciplines have traditionally been quite separated. However increasing complexity of the designs and multiple, often even contradictory constraints have led to a need for more simultaneous and interactive hardware/software co-design processes. [28] The goal is to find a combination of hardware and software that meets the needs of the application in the most efficient way. As the designs get more complex and the competition in the market gets tougher, external requirements like unit cost and time-to-market constraints become increasingly important. This has made the possibility to reuse existing designs an

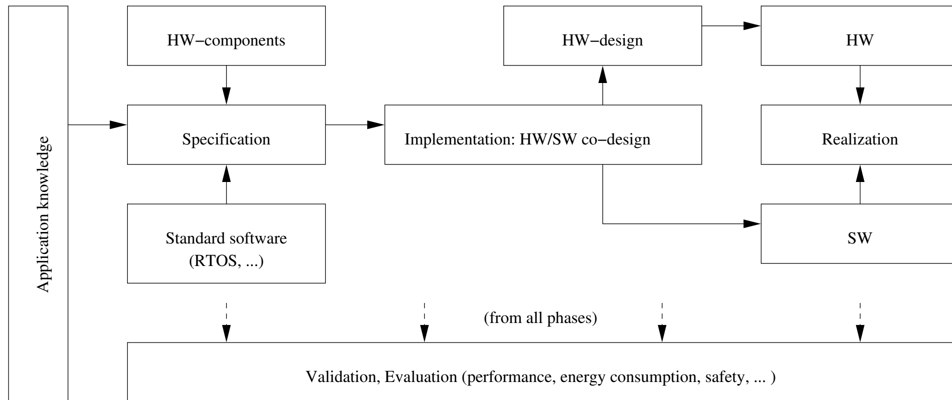


Figure 2.1: Design information flow for embedded systems. [31]

essential part of design, resulting in design paradigms like platform-based design. [31]

The critical design issues usually driving the development of embedded systems are *price*, *power consumption* and *application-specific performance* in contrast to price-performance and graphics performance that drive accordingly the design of desktop PC's. [17]

A simplified design information flow for embedded systems is shown in figure 2.1. The design flow starts with defining a specification which should comprise of all the available information about the application. Also considerations of standard hardware components and standard software should be included already in the specification. The implementation step can be divided roughly to hardware and software design and their integration to the final realization of the product. Parallel to each step of the design should also run validation of the existing design against the specification and evaluating it for various metrics, e.g. performance or power consumption, to make certain that the design is turning out as expected.

However, the possible, and usually even inevitable, feedbacks and design step repetitions that occur during design of embedded systems are not visible in the figure. For example, during the design of some hardware modules new constraints could be found, leading to retouching of the specification that in turn could affect the whole design process as some components might have to be designed and verified again. [31]

Real-time concepts

One of the most definitive characteristics of embedded systems is real-time requirements. A representative definition of a real-time system is given in [26].

> A real-time system is a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure.

A failed system is further defined as "a system that cannot satisfy one or more of the requirements stipulated in the formal system specification".

Practically any system could be interpreted as a real time system, as no practical system can have an infinite response time. Common programs, like a word processor, need to fulfill some reasonable timing requirements, as it would be impossible to use them effectively with for example a response delay of 10 seconds. Thus it is a common practice in literature to distinguish between *soft* and *hard* real-time systems.

> A soft real-time system is one in which performance is degraded but not destroyed by failure to meet response-time constraints. [26]

Displaying video on a mobile handset is an example of a soft real-time system. It can drop frames due to missing some streaming deadlines thus degrading the viewing experience, but still not causing anything irreversible.

> A hard real-time system is one in which failure to meet a single deadline may lead to complete and catastrophic system failure. [26]

An anti-lock braking system (ABS) of a car is an example of a hard-real time system. An ABS failing its time constraints could lead to degraded braking performance, which could have fatal consequences.

Hardware

Due to embedded systems covering such a wide range of diverse applications, embedded devices employ hardware of a wide variety. Roughly categorized, embedded systems use hardware for input/output-functions (I/O), communications and for control and processing. I/O-functions are used for connecting the system to the surrounding physical world through sensors and actuators. Communications are used to connect the system internally and to establish connections with other systems. Control and processing are used to process the I/O-data to achieve the purpose of the system. [31]

I/O-functions that are responsible for connecting the system to the physical world are implemented by sensors and actuators. Sensors transform physical quantities

such as weight, voltage, temperature etc. into electrical signals that can be digitized with the help of sample-and-hold-circuits and analog-to-digital-converters (ADC). Actuators do the exact opposite, they transform energy and electrical signals from the control part of the system to physical quantities. Motors and loudspeakers are common examples of actuators.

Communications are used to make contact between the I/O-functions and the control and processing parts of the system. Nowadays a rising trend is to establish connections between embedded systems and also between embedded and non-embedded systems. A wide variety of communication methods for an equally wide variety of communication needs exist, ranging from trivial wired protocols all the way to complicated radio communication schemes. The design of communication hardware is a demanding task as there are a lot of requirements to be taken into account, such as real-time behavior, efficiency and fault tolerance.

The heart of an embedded system lies in the control and processing part. In the most trivial applications, only analog electronics can be used to implement for example simple control loops, but for practical embedded systems one of the three main categories is used: Application-Specific Integrated Circuits (ASIC), reconfigurable logic or programmable processors. The choice between these three has to be made through considering the desired performance in various areas, such as energy-efficiency and unit price.

As their name suggests, ASICs are chips that are tailored specifically for the needs of the applications. This makes them the most energy-efficient choice thus offering the best performance measured in Operations/Watt. However designing and manufacturing ASICs is very expensive, making them feasible only for projects where the cost of the system is of lesser importance or when the product is going into mass production.

Reconfigurable logic, commonly referred to as Field-Programmable Gate Arrays (FPGA), are standard chips that can be reconfigured to provide functionality normally implemented by ASICs. They are more expensive than standard processors but offer much of the properties of ASICs but with a remarkably lower starting cost. This makes them feasible for systems needing good energy-performance but which are not intended for mass production. FPGAs are also used for prototyping due to the small starting costs.

There is a wide variety of standard processors used in embedded systems ranging from basic 4-bit micro-controllers through Digital Signal Processors (DSP) to 32-bit multimedia processors. Standard processors provide the best flexibility as the functionality of the whole system can be easily changed with just reprogramming the processor. Standard processors have also a cheap unit price, making them feasible for mass production. They lose, however, to ASICs and reconfigurable logic in energy-efficiency. Despite of this, the wide range of available processors makes it possible to find a suitable processor for many energy-efficient systems. For example, DSPs often measure better than standard CPUs in operations/watt. [31]

Embedded software

The software used in embedded systems differs from traditional computing in that it deals with phenomena stemming from the physical world. Rather than processing input data to output data, embedded software has to interact with its surroundings taking into consideration properties like absolute time and power consumed, matters that have been of lesser interest in more traditional computing methodologies. For an embedded program to be correct and thus safe to use, issues like *timeliness* and *concurrency* have to be taken into account. In addition, embedded software can further be characterized as having to cope with liveness, reactivity and heterogeneity. [27]

Timeliness is at the very heart of embedded systems and their programming, yet absolute time as a concept is casually disregarded in many of the programming paradigms. Improvements in computing performance mostly come from introduction of statistical methods, such as branch prediction or delicate caching schemes. These do raise overall throughput of the system but as a side-effect have also a negative impact on the predictability and even reliability of the system. This, of course, renders architectures using such techniques useless in the case of strict real-time constraints, which are very common in embedded systems.

Concurrency is another key concept. The physical surroundings of an embedded system rarely succumb to nicely waiting for their turn in order to be taken care of. Instead, a lot of activity may be happening simultaneously. The traditional answer to dealing with this has been to use constructs like threads, processes, semaphores and monitors. However, these tools also contribute negatively to the system reliability as understanding them thoroughly becomes quickly impossible when the complexity of the case grows.

Many a solution has been proposed to deal with these matters, yet most of the embedded software relies on rather old and simple techniques. For example, real-time operating systems (see Section 2.3) are widely deployed to fulfill real-time constraints. However, these techniques are generally lacking in many of the features, which are of growing importance also in the embedded domain. A modern embedded device may incorporate networking and complex dynamic applications, effectively creating a need for newer computing paradigms to be used in addition to the traditional ones.

2.2 Multi-core processors

In this section, multi-core processors are introduced. DynOS SPUMONE is a concept that aims to improve the exploitation of parallel computing power provided by modern embedded multi-core processors. First, performance issues of multi-core computing are introduced. Then, different multi-core architectures are discussed. This is followed by an introduction to different memory architectures and caching

schemes used in multi-core architectures. The section is concluded with a discussion of synchronization issues for multi-core processors.

Overview

The computing performance of processors is recently more and more limited by power constraints thus eliminating the efficiency in further exploiting traditional techniques such as increasing the clock frequency or designing more complex pipelines. Due to increased power usage, processors with high frequencies are becoming increasingly expensive to use, especially in large quantities such as server farms. They are also becoming very difficult and thus expensive to package in a way that prevents them from being destroyed by their ever-growing heat generation. Also the practical limits of increasing performance by exploiting instruction-level parallelism with longer pipelines and superscalar structures have largely been met. The applications, however, continue to grow in complexity and there is no sign of the demand for more processing power of coming to settle. Clearly there is a need for new ways to increase the computing performance through new approaches, such as architectures that exploit thread-level parallelism. Multi-core processors are seen as a prominent answer to this need. [35]

Performance

The fundamental performance metric of a processor is the rate, at which it executes instructions. It can be formulated as

$$MIPS\ rate = f \times IPC, \tag{2.1}$$

where f is the processor clock frequency in MHz and IPC (Instructions Per Cycle) is the average number of instructions executed per cycle. The traditional approaches to increasing processor performance have been to increase the clock frequency and to increase the IPC by using schemes based on pipelining. These approaches are however reaching to their limits due to complexity and power constraint concerns. The need for more computing power, however, is not on the decline. Multi-core computing provides a new approach to increase the IPC rate without introducing a lot of additional complexity to the design. [47]

Theoretical limits for the relative speedup of running a single program on multiple cores in parallel can be obtained by using Amdahl's law. It was stated already in 1967 in [1] and later formulated as:

$$Speedup = \frac{1}{s + \frac{p}{n}}, \tag{2.2}$$

where s is the portion of time spent on sequential parts of the program that cannot

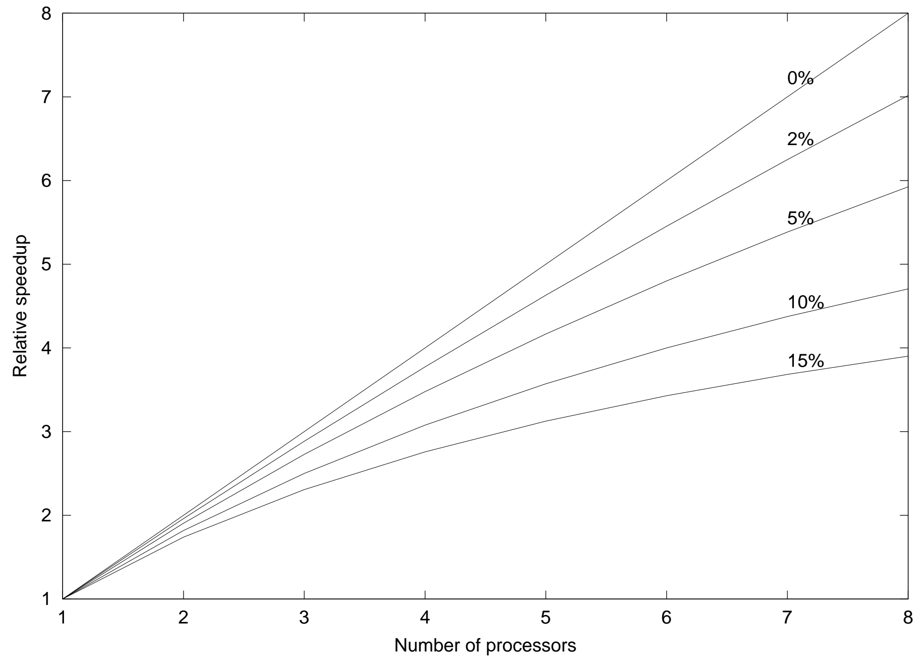


Figure 2.2: Relative speedup with various sequential proportions.

be parallelized and p is the portion of time spent on parallelized parts of the program. For mathematical simplicity it is defined that $s + p = 1$. [13]

The speedup gain of using multiple cores for running a single program clearly is largely dependent of the proportion of the programs code that can be parallelized. In the rather rare case of there being absolutely no sequential portions in the program ($s = 0$ and $p = 1$), the relative speedup will amount to the number of cores used. However, with increasing proportion of sequential parts, the gain will start to decrease quite quickly as can be seen from figure 2.2. With 10% sequential parts the speedup with 8 cores will be only less than 5 times and with 15% less than half of the theoretical maximum.

The use case of a multi-core computer, however, seldom is running a single program simultaneously on all of its cores, especially in the embedded domain. Independent processes, which are run simultaneously, can be distributed in many ways to all or some of the cores by the operating system scheduler. In these cases the performance bottleneck of the system might be in the system overhead of communicating and migrating processes between cores, instead of the portion of sequential parts in the programs. This can lead to a much worse performance than anticipated. Even worse, the performance can actually degrade if a job that parallelizes poorly is distributed to too many cores. This is caused by the overheads that come from controlling the execution of the program on multiple cores. [32]

Architectures

A multi-core processor is simply put a chip, which has multiple processor cores on the same die. Multi-cores often differ from multiprocessor architectures, which have many separate processor chips integrated on PCB-level or higher, in that the cores of a multi-core processor usually share more resources, e.g. I/O buses, cache and memory. [17]

Computer organizations can be coarsely divided into four categories of parallelism, a classification known as Flynn's taxonomy, which was introduced already in 1966 in [12]. The division is made by looking at instruction and data streams at the most constrained component of the hardware and its multiplicity to service the streams. Thus we get the following four categories:

1. Single instruction stream, single data stream (SISD)
2. Single instruction stream, multiple data streams (SIMD)
3. Multiple instruction streams, single data stream (MISD)
4. Multiple instruction streams, multiple data streams (MIMD)

Uniprocessors are SISD architectures. Processors based on the SIMD architecture are used for applications with a lot of data-level parallelism, such as graphic processing. Vector architectures, such as modern graphics processing units (GPU), are usually SIMD architectures. MISD architectures have not been produced commercially. A MIMD architecture enables thread-level parallelism as they fetch both their instructions and data independently, hence allowing multiple threads to be run in parallel. Multi-core processors are generally MIMD architectures. In addition to the four basic categories, also hybrid-designs, which are combinations of some of them, do exist. [17]

General-purpose multi-core processors are usually based on the MIMD model. Thread-level parallelism makes it easy to run independent processes on the cores making the architecture suitable for modern general-purpose computing, which is largely based on multitasking.

Memory architectures

Multi-core architectures are differentiated mostly by their organization of different levels of memory. Processors based on the MIMD model can be divided in to two categories based on the way the memory is organized. *Centralized shared-memory architecture*, or *uniform memory access* (UMA), use a central shared memory that all of the cores have access to through their individual caches. Advantages of this organization are uniform latency from memory for all cores and simplicity in design. However, the architecture scales badly and is practically not feasible for larger amount of cores because the requirements for memory bandwidth grow too large.

Processors utilizing a centralized shared-memory architecture are called symmetric (shared-memory) multiprocessors (SMP).

In a *physically distributed memory architecture*, the memory is spread across the cores. Each core has its own local memory and the caches associated with it. The cores are attached to a high-bandwidth interconnection network through which data is shared between processors. Some amount of shared memory may also be used. The benefits in distributing memory are its substantially easier and cost-effective scalability and fast access to the local memory for the individual cores. Downsides of the architecture are that the data exchange between processors become more complex and that taking advantage of the higher bandwidth in accessing local memory requires additional software design efforts.

Multi-core processors using distributed memory can be further organized in two ways. The physically separate memories can be handled as logical one memory space with each of the memories accessible from any of the cores. These kind of architectures are called distributed shared-memory (DSM), or because of the variable latencies when accessing memory in different physical locations, nonuniform memory access (NUMA). The other way is to keep the local memories of the cores private and not accessible by other cores. This makes the individual cores effectively separate computers. In these architectures data exchange between processors has to be implemented through message-passing mechanisms. These message-passing multiprocessors are usually not used for multi-core designs due to their bigger latencies in data exchange between cores. The architecture is mainly used in clusters with parallelized workloads. [17]

Caches

In practice, multi-core processors use some variation of the NUMA architecture. Differences come from cache organization between the cores. Normally, two or three levels of cache between a core and the main memory are used. L2 caches can be either dedicated to individual cores or shared between them all. Somewhat unexpectedly, shared L2 cache can have benefits in comparison to dedicated L2 caches, especially for situations when two cores are working on the same data, which will be available for all in fast cache memory after one core has accessed it once in the main memory. This makes also inter-core communication easy to implement through shared memory areas. A shared L3 cache can be used to make L2 caches local while still receiving the advantages of shared cache. [47]

For multi-core processors, extra care has to be taken to ensure cache coherency. As at least L1 cache is local to each core, two cores might end up having different views of the same memory location. This happens e.g. if core A has read memory location X and still holds the read value in its local cache, and core B goes to change the value in the same memory location. Without any precautions, core A would be left with an outdated value for X in its local cache.

Memory coherency is defined as follows. A memory system is coherent if the following requirements are fulfilled.

1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
3. Writes to the same location are *serialized*; that is, two writes to the same location by any two processors are seen in the same order by all processors. [17]

The first requirement serves to preserve the program order. The second requirement ensures that cores cannot continuously read old data values. The last requirement is that of *write serialization*. It makes sure all of the cores see consequent writes in the same order, thus eliminating the risk of a core to be left with an old value in its cache, if it would see e.g. two writes in reverse order.

Obviously, memory coherence is a very important issue and has to be taken in to account when designing multi-core processors. The caches have to provide both *migration* and *replication* for memory blocks that are shared between cores. In multi-core processors, *cache coherence protocols* are usually implemented in hardware for efficiency reasons. There are two different approaches for implementing cache coherence protocols. *Directory-based protocols* implement a central directory where the sharing status of memory blocks is stored and can be checked by caches prior to operations. *Snooping protocols* use a distributed model of the sharing statuses where all of the caches are interconnected by some bus and constantly snoop the data traffic in the bus to update the local cache if other cores write to the same memory locations. Directory-based protocols are somewhat heavier to implement, but they scale up better than snooping protocols those scalability is inherently limited by the broadcast nature of the protocols. Implementation of snooping protocols is then again cheaper as the preexisting memory buses can be used also for interrogating the status of the caches whereas for a directory-based protocol, a centralized data structure has to be implemented, taking up expensive die real estate. For this reason, today's multi-core processors ensure cache coherency usually by snooping protocols. However, the situation may change if the core-count of processors keeps on growing, rendering snooping protocols eventually unusable.

Synchronization

Multi-core processors have to provide synchronization mechanisms for programs so that they can accommodate for situations where two or more cores try to simultane-

ously perform operations to the same memory location. This is important not only for data-storing but also for accessing other resources like peripheral devices. On the hardware level, it is necessary to be able to perform uninterruptible read and write operations. Such operations are called *atomic*. Lock and unlock synchronization operations can then be constructed in software using the atomic operations. Further, more complex synchronization schemes can be built using the lock operations. As the number of cores in a processor grows, synchronization schemes introduce more latency to the system. Thus appropriate attention should be taken in software to ensure effective use and distribution of common resources. [17]

2.3 Operating systems

In this section operating systems, the most visible part of DynOS SPUMONE for the user, are introduced. We start by introducing and defining what operating systems are and for what they are used. Then we take a look at the two key organization types of operating system kernels, monolithic kernels and microkernels. In addition we distinguish between general-purpose operating systems and real-time operating systems and introduce key concepts concerning both. Finally we present the Linux operating system and its modular kernel structure that is of great importance in the implementation of the DynOS SPUMONE.

Overview

In the early days of computing, computers were capable of running only a single program at a time. Operators interacted with the computer hardware directly and machine time was mostly reserved by external measures, such as by booking via a sign-up sheet. As computer hardware became capable of processing more and more data, new ways to improve the utilization and usability of the computers also started to appear. Operating systems have since evolved from these early attempts to a point where virtually every computer, only the most basic embedded systems aside, runs an operating system. [47]

Operating systems are collections of programs on the top of computer hardware that control the execution of applications and provide an interface between programs and the computer hardware. The main component of an operating system is the kernel, or nucleus that contains the most frequently used operations. The kernel is stored in main memory and runs in the privileged mode of the processor, obtaining thus unrestricted access to all of the computers resources. The main task of the operating system is to divide these resources among user and system processes.

The design of operating systems can be thought of having three main objectives:

- Convenience, an operating system makes a computer more convenient to use.

- Efficiency, an operating system allows the computer system resources to be used in an efficient manner.
- Ability to evolve, an operating system should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service. [47]

The first objective concerns not only the end users, who will usually benefit from standardized ways of accomplishing standard tasks, but most of all programmers whose program development burden is greatly facilitated by the convenient interface to the computer hardware provided by the operating system. This way the programmer does not have to worry about the details of the computer hardware. In addition the programs become easier to port between different platforms.

In the second objective the operating system is seen as a resource manager. An operating system is responsible for dividing the computer resources, such as central processing unit (CPU) time, memory and I/O devices, between all the user programs. It should accomplish this in an efficient manner in order to both enhance the user experience and maximize the usage possibilities of the hardware.

The last objective comes from the inevitable evolution of computer hardware that drives both upgrading and eventually even redesign of operating systems. As old hardware evolves and upgrades are published, operating systems need to adapt to these changes. Also new services and fixes to faults will have to be introduced over the course of the operating systems life cycle. All this implies that operating systems should be designed in a way that makes it easy for developers to develop them further. This again implies they should be constructed in a modular manner with well-specified interfaces between the modules. As in any large programming project, a well-written and organized documentation is essential in achieving the required ease of development.

Functions

Operating systems can be thought to have five key functions that form the basis for its operations:

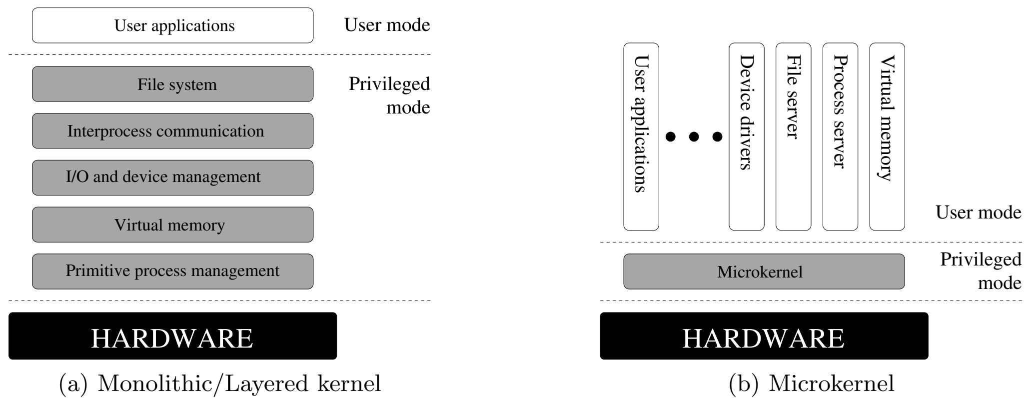
- Processes
- Memory management
- Scheduling and resource management
- Information protection and security
- System structure [47]

Process is an abstraction that helps the operating system to manage different applications on the computer. Every process has an execution context attached to it by the operating system that is not available for the process itself. The context includes all the information needed to run the process on the computer hardware and to manage it properly. This includes normally various processor registers and process information such as the priority of the process. A context switch is an event when the kernel changes the process to be run. This can happen due to a user process using a system call to invoke the kernel, an interrupt or the system timer. The context of the running process has to be changed to the context of the next process. The context of the old process has to be first saved, if the process was not exterminated finally.

In order to provide process isolation, automatic memory allocation and management as well as protection and access control, an operating system must provide proper memory management mechanisms. This is usually accomplished by using virtual memory. Programs are not shown the physical, real memory space but instead a virtual one that is freely used by the program as it wishes without having to take into consideration possible collisions with other process's or the operating systems data. This is taken care by the operating system. The virtual address spaces of all the processes are managed and stored in either the main memory or in the computers hard disk. When a process wants to read or write data from a virtual address, the operating systems memory management translates the virtual address into a real address.

An operating system has to manage its available resources, such as processors, main memory and I/O devices, and schedule them between processes in a manner that is both efficient and fair and still leaves the possibility of responding differentially to service request, which might be needed for example to accommodate for processes that have high priorities. Optimizing for efficiency is a balancing problem varying with the situation as the requirements for efficiency are usually contradictory. E.g. optimizing scheduling for total throughput does not produce optimal results for minimizing latency and vice versa.

As multi-user systems and networking have become ordinary, concerns about security have risen and demand attention from the design of operating systems. Securing data integrity, authenticity, confidentiality and availability is by no means an easy task, as the systems keep growing and thus developing into more complex entities that are hard to master and understand completely. This is one of the reasons why system structure is an important part of operating systems design. As the size of an operating system grows beyond millions of lines of code, as has already happened for the well-known desktop variants, it becomes essential to find ways to deal with the complexity. In the next chapter we take a look at how operating system kernels are structured.



(a) Monolithic/Layered kernel

(b) Microkernel

Figure 2.3: Kernel organization.

Kernel organization

Today, basically two main ways of organizing the design of an operating system exist, the monolithic kernel and the microkernel. The main difference between the two is that in a monolithic kernel, more or less all of the kernels services run in privileged mode inside the kernel, whereas in a microkernel, as the name suggests, the kernel size is kept as small as possible by keeping only the absolutely necessary parts of the operating system in the kernel. All other services are implemented as user mode processes, often called servers, which communicate with each other and the kernel via message-passing mechanisms. In practice modern monolithic kernels are not organized as big clumps, but, rather as piles of layers each running in privileged mode and providing separate services. Hence modern monolithic kernels of this kind are often referred to as by a more appropriate name, a layered kernel. The differences of the two approaches are illustrated in figure 2.3. [47]

The microkernel approach presents many design advantages. As many of the operation system services are provided as servers that run as user mode processes, the system becomes highly modular and flexible to use and develop. The services can then be more easily tailored to specific needs or even left out when not needed. Microkernels are also very extensible, as adding new services does not require modifying the kernel and can be done completely in user mode instead of the privileged mode. In addition, the small size of code that actually interacts directly with hardware makes the task of porting microkernels to new hardware architectures remarkably easier than porting monolithic kernels. [47]

Isolating server processes from each other to separate user mode processes is one of the main benefits of the microkernel approach. Research has shown that device drivers, often written by third parties, can have error rates up to three to seven times more than the rest of the kernel [9]. If they reside in the kernel and execute in privileged mode like in monolithic kernels, failures in device drivers can compromise the reliability of the whole system. For microkernels this is a problem of a smaller

scale, as the damages done by a faulty device driver will be isolated to its own process without affecting the kernel or other processes, thus enhancing the system reliability. Microkernel reliability is also affected by the fact that the actual kernels code base is very small and can thus be tested more thoroughly and even be formally verified [25]. With monolithic kernels that are magnitudes larger this is not possible and other, less reliable methods have to be used.

A downside of the microkernel approach is its performance. As messages are passed through the microkernel, accepting and decoding them will take longer than a simple service call would take. However, it has been shown that this can be avoided by optimizations and structural decisions so that a microkernel can achieve even real-time performance. [41]

Among practical modern operating systems microkernels are in minority. The governing desktop operating systems, Microsoft Windows and almost all UNIX kernels, including Linux, all have monolithic kernels. However hybrid kernels, such as XNU, which is the Apples OS X operating system is built, are becoming more common. Even more, modern monolithic kernels apply some of the microkernel paradigms, particularly that of modularity. Although monolithic, modern kernels are more than often implemented in a modular manner. For example, Linux is structured in a way that enables runtime insertion and removal of kernel modules. Linux kernel modules are introduced in section 2.3. However in embedded systems the benefits of the microkernel approach are very essential. Flexibility, extensibility, portability and most of all reliability provided by microkernels are all of great importance when designing embedded systems.

Embedded operating systems

As mentioned earlier, nowadays only the most trivial embedded systems suffice without a proper operating system. The complexity of modern embedded applications has lead to the need of scheduling, task switching and I/O management services. These needs are best served by an embedded operating system. Furthermore using an operating system enables a greater reuse possibility of already existing intellectual property (IP) thus providing a convenient way to deal with increasing complexity and also substantially lightening the design cost of new systems. [31].

Although a large variety of embedded systems exist, embedded operating systems share some common characteristics. One of the essential features of an embedded operating system is configurability. It is important to be able to flexibly tailor an embedded operating system to the varying needs of different applications. As embedded systems grow larger and more complicated, the amount of configuration options grows too, which makes the embedded operating systems more complicated. This in turn increases complexity of the whole systems verification process.

Another common feature of embedded operating systems is that due to the large variety of possible peripheral devices, no device needs to be supported by all the

versions of the operating system. For slow tasks like networking or disk management it may be profitable to use dedicated tasks instead of kernel drivers. In addition, embedded operation systems do not necessarily need protection mechanisms. This applies especially for systems that are dedicated to a particular function and never have untested applications running on them. The rationale in leaving away the protection mechanisms is in saving the extra overhead. It should be noted though that this depends highly from the systems level of complexity. For example, it would be intolerable to have no protection mechanisms in modern mobile handsets as they are used to run a lot of code that is not made or tested by the device manufacturer.

The last feature is that interrupts can be employed by any process. Also this applies to more closed applications that have been thoroughly tested and allow no foreign code to be run on them, thus being able to benefit from decreased overheads by not necessarily invoking the operating system for every interrupt. Yet again for more complex modern embedded systems this approach is often too compromising for the overall system reliability.

Real-time operating systems (RTOS)

Real-time operating systems (RTOS) are a breed of operating systems, which are used in embedded systems with real-time constraints that are described in chapter 2.1. In DynOS SPUMONE the guest RTOS is TOPPERS/JSP, which is further described in chapter 3.4.

As real-time operating systems are almost inherently used in embedded systems, the common characteristics of embedded operating systems defined above in chapter 2.3 apply also for real time operating systems. These can then be further described by three key requirements:

- The timing behaviour of the OS must be predictable.
- The OS must manage the timing and scheduling of tasks.
- The OS must be fast. [31]

Predictable timing means that every service provided by the operating system must have a guaranteed maximum execution time. This is essential in designing real-time systems as there may be hard deadlines that are not to be missed. Obviously, the RTOS is also responsible for providing efficient scheduling services and also fine-grained timing services. The last requirement of the RTOS having to be fast is due to the nature of real-time systems that may have to be able to respond in only fractions of seconds thus producing deadlines that are doomed to be missed by slower systems.

For real-time systems, two types of approaches are commonly used. Either a fast real-time kernel or a standard OS with real-time extensions. [31] Both ways have their advantages. For traditional real-time systems which are dedicated to only a

handful of well-defined functions, the use of a fast and dedicated real-time kernel, which can be thoroughly tested, tend to be the best solution. However, due to embedded systems growing in complexity and starting to incorporate even dynamic code, the ease of development that is achieved by using a standard OS is also growing in importance. Recently ways to gain the benefits from both of the approaches have been introduced, including DynOS SPUMONE. The idea is to run a virtual machine monitor or a microkernel and let both the real-time and general-purpose operating systems run on it while preventing the guest OSes from colliding with each other. This approach has proven useful and powerful to be used for complex embedded systems and is used for example in modern handsets. [15]

Linux

Linux is used in the prototype implementation of DynOS SPUMONE as the guest general-purpose operating system. Linux is a general-purpose operating system which runs on a wide variety of hardware and is used throughout the world for all kinds of applications ranging from small embedded systems to massive server-cluster installations. Its development started as a hobby project by a Finnish computer science university student named Linus Torvalds. He published his project, which was originally a UNIX clone for the Intel 80386 architecture, as open source and for free to the Internet in 1991 and let other people to partake in the development process. Since then the scale of Linux development has exploded. Today the number of people contributing to the kernel development is measured in thousands, with Torvalds still controlling the overall progress. [47] All the information provided in here applies to Linux kernel series 2.6.

Linux is compatible with the IEEE "Portable Operating Systems based in Unix" (POSIX) standard. This makes porting most of the existing Unix-software to Linux an easy task, which adds greatly to applications available for the platform. The Linux kernel support symmetric multiprocessing with various memory models, making it attractive for also multi-core architectures. Linux's effective design and configurability have made it appealing for embedded systems with only restricted resources. A bootable kernel can be loaded into a 1.44 MB floppy disk and a network server can be implemented using Linux on an old 30386 computer with only 4 MB of memory. [7] In addition to simple embedded devices, Linux has also been adopted by complex embedded systems that provide rich functionalities, such as modern smartphones running on e.g. Google's "Android" or Nokia's and Intel's "MeeGo" -platforms. [34] [33]

Kernel modules

Although the Linux kernel, as almost all other Unix kernel variants, is by design monolithic, it does incorporate some advantageous features usually associated with modular design. The Linux kernel is structured as a collection of loadable modules

that can be loaded and unloaded at run-time. This makes the kernel very configurable and simplifies kernel development, as testing new features can be done simply by installing new kernel modules instead of compiling the whole kernel. Almost everything in the kernel can be implemented as a loadable module, examples being file systems and device drivers. Thanks to this modular design, users can also tailor their kernel for the specific needs of their systems. Unnecessary functionality can be left out from the custom kernel, which gives the user a better control of e.g. the kernel memory usage, which can thus be reduced without having to touch the original kernel distribution. The user interface of the DynOS SPUMONE prototype is implemented as a kernel module.

A module does not run as its own process but is instead ran on behalf of the current process in privileged mode. The module has thus access to all of the kernels functions but is restricted in other ways. It can not use system calls and it has to use kernel memory, which is very limited. Modules are generally not designed to be used as normal programs that are always on, but, instead as reactive programs that are summoned only when needed to provide interaction between user programs and some kernel internal functionality unreachable otherwise. For example, device drivers in Linux kernel are implemented as modules. They are invoked when a user program needs to communicate with some device, a task that usually requires access to interfaces only available for processes running in privileged mode.

2.4 Virtual machines

In the hearth of DynOS SPUMONE lies SPUMONE, a very light virtual machine that plays host for the various operating systems that are running user applications in the system. In this section we introduce the basic concept of a virtual machine (VM) and their implementation. We start by introducing general definitions concerning VMs. We continue by presenting the basic types of VMs, followed by implementation architectures of VMs. Last, we take a look at using VMs in server and embedded environments.

Overview

Virtual machines (VM) are software interfaces that virtualize parts of hardware to user programs while giving them an illusion of accessing real hardware. Recently, virtual machines have become an important everyday part of modern computing. Although used in mainframes and servers since the 1960s to make sharing resources between multiple users convenient, virtual machines never really gained much popularity in the personal computer domain that started in the 1980s. However due to advances in computer performance and also virtualization techniques, they have recently become feasible tools in solving various problems also in the modern personal computing domain and even in the embedded market.

The importance of isolation and security has increased significantly as multi-user systems and network connections have become commonplace. Even modern operating systems fail at providing sufficient security and reliability for many systems. By using virtual machines, a single computer can be shared between unrelated users with, in theory, perfect isolation and the possibility of utilizing different operating systems according to individual needs of the users. Virtual machines can also be used for developing software for different platforms. The software in development can be run on different operating systems inside a virtual machine. This may reduce development cost as it is faster and cheaper to maintain multiple operating systems on a virtual machine than it would be to have them on separate machines. Especially embedded systems development can benefit from this flexibility as running code in target hardware can be slow, inconvenient and lack in proper debugging tools and other assistance. In cases where hardware and software are developed simultaneously, using VMs for software development could even be the only possibility.

Definitions

The study of virtual machines started in 1960's when bigger mainframes started to emerge and a practical way of dividing computing time and other resources between multiple users was needed. In those early days, VMs were thought mainly as resource dividers that only duplicate existing hardware to be used by many users. During the years, many definitions have been proposed to define virtual machines. Popek and Goldberg gave the following often quoted definition for a VM in 1974.

> A virtual machine is to be taken as an *efficient, isolated duplicate* of the real machine. [36]

This widely spread definition addresses important implementation issues of efficiency and isolation, but is somewhat outdated, as today there are a lot of VMs that are not based on any real physical computer architectures, the Java Virtual Machine (JVM) being a common example. Another definition from 1973 by Buzen and Gagliardi is more general as it is based on the important abstraction of machine interfaces.

> A basic machine interface which is not supported directly on a bare machine but is instead supported in a manner similar to an extended machine interface is known as a virtual machine. [8]

Rosenblum sums up many definitions of VM in a simple way looking at the issue from the software point of view.

> Despite the variations, in all definitions the virtual machine is a target for a programmer or compilation system. In other words, software is written to run on the virtual machine. [40]

Basically, a VM can be thought of as a layer that gives the code running on it the feeling of accessing real hardware. The software that is responsible for creating this illusion is called a *Virtual Machine Monitor (VMM)* or sometimes a *hypervisor*. A VMM was sufficiently generally defined by Buzen and Gagliardi.

> The program which supports the additional basic machine interfaces is known as a virtual machine monitor or VMM. [8]

The VMM operates on bare *host hardware* or on a *host OS*, depending on the implementation. The VMM itself provides *host VMs* for the applications or OSes running on it. These are called *guests* from the VMs perspective.

Types of virtual machines

Modern computers are highly complex machines that are virtually impossible to master by one person, yet they continue to evolve and grow also in complexity each year. This has been possible largely due to the fact that computers are designed as hierarchies that have well-defined interfaces between separate levels of abstractions. This separation makes it possible to design higher level services and programs without having to take the details of the lower level functionality into consideration. All one has to do is to use the interface provided by the lower level. Virtual machines virtualize a computer or a part of it on some abstraction level by fulfilling and providing the appropriate interface. This way programs or services running on that interface can also run on the virtual machine. [44]

Figure 2.4 shows three computer interface layers that are commonly used by virtual machines. The "lowest" level interface is the *Instruction Set Architecture* (ISA) that is the border between software and hardware. Therefore a virtual machine that virtualizes ISA is equivalent to a complete computer system from the software point of view. ISA can further be divided into two interfaces, the user ISA that is freely used by user programs and the system ISA that in addition to the user ISA provides commands to manage the hardware resources. The system ISA can be accessed directly only by the operating system. Higher level programs that want to access resources provided by to system ISA have to do it through the operating system by utilizing the higher level interfaces; the *Application Binary Interface* (ABI) or the *Application Programming Interface* (API).

Programs use the ABI to access the computers hardware resources. ABI provides this access through the user ISA and the system call interface. The system calls invoke services in the operating systems that interact with the hardware through the system ISA. This way the operating system can evaluate the validity of the user programs requests before executing them. The API provides the programs with access to the hardware resources by using the user ISA and High-Level Language (HLL) libraries. Usually the programs invoke system calls through libraries. These three interfaces are the most important ones regarding virtual machines.

A VM that provides both the user and system ISA interfaces is called a *system virtual machine*. A user of a system VM has an illusion of having a complete physical computer system for himself. Multiple operating systems can also be run on a system VM. The VMM takes care that they will not collide although they are using the same physical hardware. System VMs can further be divided into two categories: classic system VMs and hosted VMs. A classic system VMM resides just above the hardware as the lowest level software whereas a hosted VM is run as a normal user mode program in an OS. The advantages of a hosted VM are that it can use device drivers and other services provided by the OS making the implementation easier and more flexible and that it can be installed and managed easily just like a normal program, possibly even without administrator rights to the system. [44]

A VM that provides API or ABI interfaces is called a *process virtual machine*. These kind of VMs are designed to run single processes and provide an execution environment for them. They provide a way to make applications transparent and more easily portable between different platforms. When an application is written for a process VM, it can be run on any platform that the VM runs on. This way, only the VMs need to be ported to different platforms, reducing development costs considerably. Also an operating system can be thought of as an VMM that provides process VMs for user processes, as each user process is isolated from other processes and sees a virtual memory space that only it can use.

Some process VMs are not based on any existing “real” operating systems or hardware, but instead are designed to serve well some important features of a target high-level language. These *high-level language virtual machines* (HLL VM) circumvent the hard problems with emulating existing hardware or OSes in defining their own virtual hardware tailored for the needs of the high-level language. Again, also HLL VMs aim usually for platform independence through good portability between different hardware architectures. A good example of a HLL VM is the Java Virtual Machine (JVM) that was designed for the object-oriented Java language and is today spread throughout the computing domain. [43]

Implementation techniques

Some processors architectures are aimed at enabling system virtual machines to execute directly on the hardware. These kind of architectures are called *virtualizable*. Such an architecture must allow the virtual machine monitor to trap any attempts of accessing physical hardware resources by user mode programs. This way, the VMM can let other commands execute directly on the target hardware thus improving the overall system performance. Sadly, many popular architectures such as 80x86 or many RISC variants do not fulfill this and thus are not virtualizable hence complicating VMM design for such systems. For architectures that are not virtualizable, special techniques need to be used to run VMs safely. [17]

Two main branches of VM implementation techniques exists, these are called *full-*

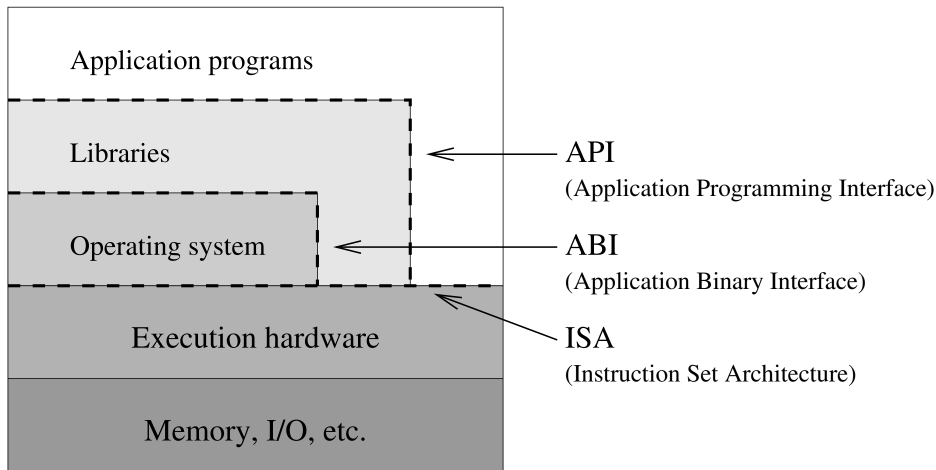


Figure 2.4: Main computer system interfaces. [44]

virtualization and *para-virtualization*. In *full virtualization* the whole underlying system is virtualized to the guest applications so that they can run without any modifications to their code. From the applications point of view it seems as it would be running on real hardware, timing aside. The original virtual machines used in mainframes were built with full-virtualization. The main benefit of this approach is clearly that no code modifications to the applications or even to the guest OSes are needed. If the VM implements the same interfaces as the real underlying hardware, the implementation can also be very effective as a lot of the code can be run directly on the hardware. However for architectures that are not originally designed for virtualization, as is for the popular x86-architecture, this is not easily attained and involves creative programming that has its effect also for the systems performance. However a wide variety of full-virtualization VMs are presently used also in the desktop computing domain, e.g. VirtualPC and VMWare.

In *para-virtualization* the VM implements an interface, which is similar, but not identical, to the "real" interface. In order to adapt to this custom interface, the guest applications or OSes have to be modified accordingly. The idea is to achieve better performance by letting the VMM and the guest OS to collaborate instead of the hard task of emulating difficult parts. Usually only the ISA interface is customized and other interfaces like the API and ABI are left as they are. This means that only the OS running on the VM needs to be modified and applications available on those OSes can run without any extra modifications, reducing the negative impact on development costs substantially. The downside of this approach is that updating the system will become difficult. New versions of both the guest operating systems and the VMMs need to be manually "fitted" together every time a new version appears. This can even stagnate system development and in the worst case lead to unreliable systems if no new security fixes can be adopted. Xen and Denali are popular examples of para-virtualized VMs. [11] [3]

Pre-virtualization is a fairly recent technique that aims to address the compatibility problems of para-virtualization without compromising system performance as much as full-virtualization. The idea is to use a method called *soft layering* to create a new layer, "in-place VMM" between the actual VMM and the guest OS. The interface towards the guest OS is kept neutral and the modifications normally made to the guest OS are instead made to the in-place VMM to provide the transparency needed to achieve better performance. The modifications are constrained by the following rules:

1. It must be possible to degrade to a neutral interface, by ignoring the co-design enhancements (thus permitting execution on raw hardware and hypervisors that lack support for the soft layering).
2. The interface must flexibly adapt to the algorithms that competitors may provide (thus supporting arbitrary hypervisor interfaces without prearrangement).

The first rule ensures that the guest OS can function on the system even if e.g. a new OS or VMM upgrade makes it impossible to use the performance modifications of the in-place VMM. In such cases a runtime fallback to a general "failsafe" interface is committed and the system can still function, albeit with a lower performance. The second rule requires the in-place VMM to be adaptable in respect to various kinds of VMMs and not to restrict itself to any certain interfaces. An implementation of the pre-virtualization approach has been made and evaluated with various VMMs and guest OSes. The tests indicate only minor performance drop in comparison to bare VMMs thus making pre-virtualization an interesting approach to combine the good sides of both full-virtualization and para-virtualization. [29]

Usage

For servers

The server domain has been using VMs since the early mainframe computers of the 60's and 70's. The evolution of VMs started when it became necessary to share computing resources and time between independent users so that their actions would not interfere with each other. Today, servers run VMMs to provide flexible use of the servers hardware resources. Large scale computing that before may have required multiple hardware platforms working together can now be implemented with cheap commodity hardware running VMs that provide the necessary services. As current OSes and server system-software architecture achieve only up to 10% utilization of the computers resources, significant gain is obtained through utilizing multiple VMs on the machines. [11]

The rise of a computing model called "cloud computing" in recent years has also boosted VMs popularity in the server domain. In the cloud computing model appli-

cations reside in the network instead of users local systems hard drives. Users access applications through the internet and use them in their browsers. The term “Cloud” itself refers to the hardware and systems software used in the service providers data centers. The Clouds architecture is more than often based on VMs that can be started, stopped, added and reduced routinely according to the current need. For example Amazon provides a service called “Amazon Web Services” where the client can build web sites that scale according to current traffic load. The service is implemented with standard x86 hardware running Xen VMMs that provide the needed VMs for the application. [2]

For embedded devices

As embedded devices have grown in computing capability and complexity, new challenges in providing applications with rich functionality in restricted environment have arisen. Embedded devices have only limited resources and must therefore use their energy and memory in much more efficiently than their desktop or server counterparts. Another important element for embedded devices, especially modern mobile handsets, is code mobility, that is the ability to run same code on different platforms without changing it. Capable of solving these challenges, virtualization techniques have been getting a foothold also in the embedded domain.

The demands for virtualization set by complex embedded systems differ quite remarkably from those set by server or desktop environments. Whereas for server applications it is usually desirable to maintain optimal load balance of the system, for embedded systems it is crucial to maintain real-time performance at all times while at the same time providing the user with a rich set of functionality. This is arguably best provided by employing multiple OSes in the system; a GPOS for the rich functionality and a RTOS for guaranteed real-time response. Such a dual-OS environment is obtained by virtualization.

A set of requirements for VMMs that are aimed at embedded devices is proposed in [16]. The VMMs have to be *real-time capable* in order to effectively deal with the RTOS. As is in conventional VMMs, *isolation* is of concern also in VMMs for embedded systems. New challenges are, however, posed by the need for the OSes running on different VMs to communicate with each other. This applies a new requirement of *efficient communication channels*, something rarely seen in the server virtualization domain. Also the *switching between VMs has to be fast.* Heiser also argues that for security, the size of the *trusted computing base* should be as small as possible, indicating a microkernel-like approach to the architecture of the VMM.

Multiprocessor virtualization

The recent shift in computing from single-core to shared-memory multiprocessor and multi-core architectures is presenting new challenges also for virtualization. As

seen in chapter 2.2, multiprocessors are not capable of providing significant instant performance gains but must instead be used in new effective ways. Virtualization provides a way to *partition* a multiprocessor system so that its resources can be divided effectively. Two main categories exist; *physical partitioning* and *logical partitioning*. In *physical partitioning* the VMs are physically separated, that is they run basically on their own cores or processors and use their own devices. This is easy to implement and provides great isolation between different systems. Even if one virtual systems hardware would crash it is unlikely to affect other virtual systems. Physical partitioning is, however, hardly optimal for resource allocation between different virtual systems. When optimal system utilization is needed over perfect isolation between virtual systems, *logical partitioning* is used. In *logical partitioning* the hardware resources are time-multiplexed between different VMs. By using this approach more elaborate schemes of load balancing can be used thus improving the overall systems performance. However, isolation between VMs will not be on the hardware level thus reducing its benefits. [43]

Chapter 3

Building blocks of the system

Chapter summary

DynOS SPUMONE is presented in this thesis work as a proof of concept implementation that makes use of several tools and techniques, both hardware and software. In this chapter the major building blocks of the DynOS SPUMONE implementation are introduced and a look is taken at some of their relevant functionalities in order to make the modifications presented in chapter 4 comprehensible to the reader.

The most important part of the DynOS SPUMONE is obviously SPUMONE, a lightweight virtualization layer for embedded systems, and its multi-core extension. SPUMONEs design philosophy and basic functionalities are introduced in chapter 3.1. SPUMONE is built on the SH-4A architecture and the multi-core extensions runs on an experimental SH-4A based quad-core multi-core processor called RP1. Hardware and software issues of this platform are introduced in chapter 3.2.

The two guest operating systems that are run on top of DynOS SPUMONE are a multi-core-patched SH-Linux that is further modified and the real-time operating system TOPPERS/JSP. SH-Linux is used also as the main user interface to the functionalities of DynOS SPUMONE through a kernel module. SH-Linux is introduced in chapter 3.3 and TOPPERS/JSP in chapter 3.4.

3.1 SPUMONE

DynOS SPUMONE uses a modified SPUMONE light-weight virtualization layer to run multiple operating systems on the RP1 platform described in chapter 3.2. DynOS SPUMONE is actually a derivate work of the main branch of SPUMONE. In this chapter SPUMONE and the design philosophy it is built on are introduced. Key implementation issues relevant to the discussion of DynOS SPUMONEs implementation in chapter 4 are presented.

Overview

SPUMONE (originally Software Processing Unit Multiplexer) is a light-weight virtualization layer for highly sophisticated embedded systems. It is developed by the Distributed and Ubiquitous Computing Laboratory of Waseda University and is currently part of a larger research project called DEOS that aims at developing a whole dependable operating system for future embedded multimedia systems. [21]

SPUMONE is designed to be used in embedded systems running advanced services that have real-time constraints. This is the case for example in modern mobile phones that are equipped with complex user interfaces and are used to run complicated applications, but where many of the devices, like the radio, have critical real-time requirements that must be fulfilled. RTOSes are designed to cope with real-time constraints but generally only possess low-level APIs badly suited for building complicated end-user software. On the other hand GPOSes provide a wealth of easy to use and powerful APIs but lack in ability to provide guaranteed hard real-time performance. SPUMONE exploits a para-virtualization scheme to enable running both a RTOS and a GPOS simultaneously on the same hardware. This makes it possible to use a hybrid operating system environment where tasks can always be run on the most suitable OS.

The first implementation of SPUMONE was built on a single-core platform using a Renesas SH-4A processor. Testing was done with a Linux 2.6 acting as the GPOS and TOPPERS/JSP 1.3 as the RTOS. [23] Consequently a multi-core extension of SPUMONE, on which this thesis-work builds on, was developed to run on a four-core SH-4A architecture RP1 platform described in chapter 3.2. [22]

Design philosophy

The design goal of SPUMONE is to enable building a hybrid operating system environment that fulfils the following three requirements:

1. Code modification of the guest OS should be minimal.
2. The virtualization layer should be as light as possible.
3. It should be possible to reboot the guest OSes independently from each other. [23]

Derived from these three requirements, the key goal of the SPUMONE virtualization scheme is to maximize both the efficiency of the implementation as well as the efficiency of developing a hybrid operating system environment through utilizing SPUMONE. To keep the virtualization layer as light as possible (requirement 2), a para-virtualization technique is used. To maintain low engineering costs, it is required that for getting a guest OS to run on SPUMONE, only a minimal set of code changes to it should be required (requirement 1), leading to light para-virtualization.

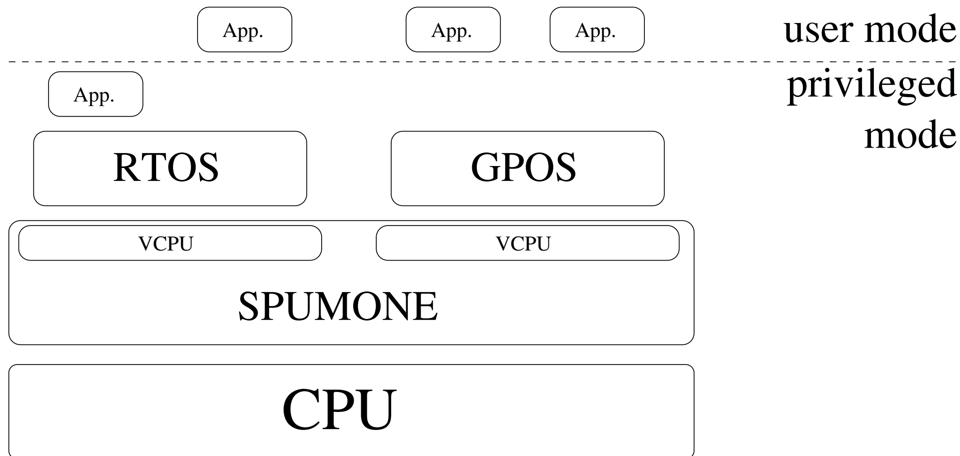


Figure 3.1: A system running on SPUMONE. [23]

The last requirement addresses the stability of the system by requiring the guest OSes to be able to reboot independently of each other.

Figure 3.1 illustrates the basic architecture of a system using SPUMONE to run simultaneously a RTOS and a GPOS. SPUMONE creates virtual CPUs (VCPU) and assigns them for the guest OSes. This way the OSes are running independently from each other and can be rebooted separately without affecting each other. No other hardware than the CPU is virtualized in order to keep the performance overhead posed by virtualization as low as possible. The guest OSes run are in the privileged mode of the real processor, providing them with an almost unlimited access to the whole system. Majority of the instructions executed by the guest OSes are run directly on the physical hardware without SPUMONE getting between. Only a handful of instructions are virtualized by SPUMONE thus the amount of required modifications to the guest OS kernels are kept to a minimum. This decreases both the virtualization performance overhead and engineering costs of the system.

Understandably, the use of SPUMONE incorporates trade-offs. The minimal virtualization overhead is obtained at the expense of isolation. As there are no memory protection mechanisms in SPUMONE, sloppy software design that ignores memory placement could lead to the guest OSes writing into each others reserved memory areas thus compromising the stability of the system. Also, the CPU aside, SPUMONE does not provide any device sharing mechanisms meaning that every guest OS has full access to all of the devices. Therefore if two guest OSes try to access the same device simultaneously without synchronizing their actions on higher level they will end up colliding and compromising the stability of the system.

Implementation details

SPUMONEs implementation is introduced in [23] and details can be found in the SPUMONE source code that is freely available for download in the project online repository. [46] SPUMONE is currently implemented for SH-4A, a RISC processor manufactured by Renesas. SPUMONE is entirely written in C and assembler. The virtualization technique used in SPUMONE is very light para-virtualization. Almost all of the code that is run by the guest OSes is directly executed at hardware. Only a minimal set of commands is virtualized and emulation techniques like binary translation are not used. The virtualized commands are the SLEEP command and interrupt transmit mechanisms. In addition, the build process of the guest OS has to be slightly adjusted.

The boot process of SPUMONE is fairly straight-forward. Upon system reset, SPUMONE is loaded to memory and started. SPUMONE then goes on to load the guest OSes to their respective memory regions and initializes VCPUs for them. The VCPUs are simply C structures that have fields for saving the guest OS's state, e.g. its registers. The VCPU structure also stores its priority and execution state. After the initialization is complete, SPUMONE launches the VCPU that has the highest priority.

SPUMONEs scheduling algorithm is very simple. It uses fixed-priority scheduling that basically ensures, that the VCPU with the highest priority can always run when it needs to. To ensure real-time performance, the VCPU assigned to the RTOS is given the highest priority so that it can execute without any external interruption from SPUMONE or other guest OSes. Even SPUMONE itself can only be invoked via the RTOS issuing a virtualized SLEEP call. This returns the execution to SPUMONE which can then schedule a new VCPU to dispatch. The VCPU which has the highest priority of VCPUs that are ready to execute, e.g. not in sleep mode, is always chosen. VCPUs that are in sleep mode will be dispatched when they receive interrupt, if at the time no VCPU with higher priority is executing.

SPUMONE needs to handle the interrupts in the system, otherwise the OS that would initialize last, would overwrite the vector handling pointers and would receive all the interrupts for itself only. In SPUMONE this is prevented by virtualizing the portions of the guest OSes that initialize interrupts. The interrupt handlers of the OS are replaced with SPUMONE API calls. The API calls return the execution to SPUMONE which decides if the interrupt will be sent to the appropriate guest OS or if it will be marked as pending in the OSes VCPU, as another VCPU with higher priority is still executing. Interrupt priorities can be set either statically at build time or dynamically by using the SPUMONE API.

SPUMONE provides no memory virtualization. Therefore the physical memory must be statically divided between SPUMONE and the guest OSes so they don't interfere with each other. This is done by modifying the build process of the guest OSes to use only their individually dedicated memory area. Normally, the GPOS

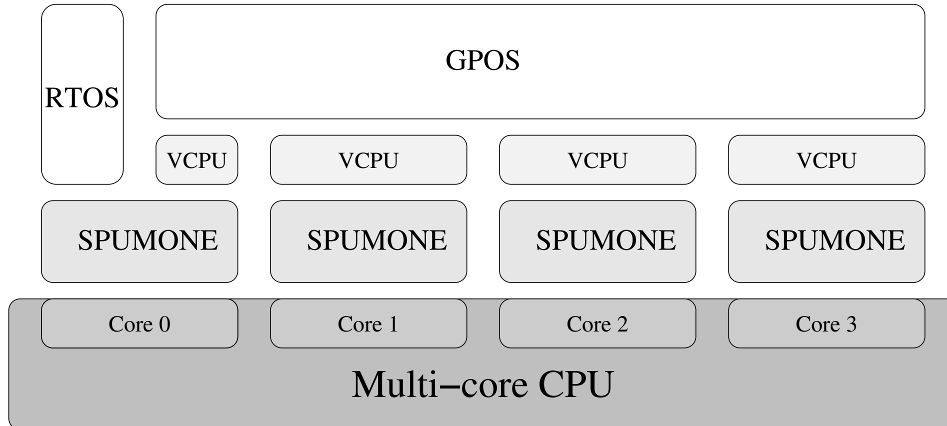


Figure 3.2: A system running on multi-core SPUMONE. [22]

is assigned the upper half of the physical memory, and the RTOS and SPUMONE are assigned smaller slices of the lower half. Unused memory areas can be used as shared memory between the OSes. Virtual inter-processor interrupts (IPI) between VCPUs can be used by the guest OSes to inform the others of new available data after writing to the shared memory region.

Multi-core SPUMONE

Multi-core SPUMONE is a branch of the main SPUMONE that enables running SPUMONE on multi-core architectures. It is currently implemented for the RP1 processor. With multi-core SPUMONE it is possible to let a multi-core capable OS, such as Linux, populate many cores and use a dedicated or a shared core to run simultaneously a RTOS on the same processor. An example of a system running on multi-core SPUMONE is illustrated in figure 3.2.

The implementation of multi-core SPUMONE differs from the main SPUMONE mainly in that it has separate SPUMONE instances on every core and a slightly more complicated booting process. Own of the SPUMONE instances serves as a master and the other ones are slaves. The master instance is the first one that is loaded to the processor in boot and it is responsible for booting the other cores with the slave instances. The synchronization between the instances during the boot is done so that after loading a slave to a core, the master instance waits for the slave to report that it has initialized itself by writing to a shared memory location. After this, the slaves enter an infinite loop where they wait for an interrupt. The master goes on to load its OSes, the RTOS and the GPOS which will then load itself to the rest of the cores. [22]

3.2 RP1 - multi-core processor

In this section we introduce the embedded multi-core processor RP1 on which the prototype implementation of DynOS SPUMONE is built to run on. After introducing the chip, its maker and its general design philosophy, we take a look at its architecture. Further on the hearth of the RP1, the SH-4A processor core and its basic components are presented. We go on by summarizing the memory and interrupt control of the RP1. Last we say a word about RP1s energy consumption and the platform on which the RP1 used in DynOS SPUMONE prototype is built on.

Overview

RP1 is a prototype quad-core System on Chip (SoC) processor manufactured by Renesas Technology, a semiconductor manufacturer established in 2003 by merging the chip operations of two Japanese chip makers, Hitachi Ltd. and Mitsubishi Electric Corporation. [10] In 2008 it was ranked by the market analyst company iSuppli Corporation as the sixth largest semiconductor manufacturer in the world.

RP1 was developed by Kasahara-Kimura laboratory (Waseda University), Renesas Technology Corporation and Hitachi, Ltd. as part of a Japanese New Energy and Industrial Technology Development Organization (NEDO) project entitled "Multicore Processor Technologies for Real Time Consumer Electronics". [14] Being a prototype, it has not been sold in the open market and the documentation concerning it is not publicly available from the manufacturer web page or otherwise. The design emphasis for the RP1 has been power-efficient high-performance embedded applications to be used in multimedia equipment, information appliances, networks, amusement equipment and car information systems (CIS). [52] [39]

RP1 is designed to meet three needs common for embedded applications:

1. Standard common on-chip-bus-based SoC integration.
2. Good adaptation to both symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP).
3. Dynamic power management, cooperating with software that can control the power of the processors, adapting to the processor load. [52]

Theoretical maximum performance of the RP1 is 4320 Millions of Instructions Per Second (MIPS) when all of the four cores run on the maximum frequency of 600 MHz. With the FPUs included in the cores enabled and all running at the maximum frequency, RP1 can achieve also the theoretical maximum of 16,8 GFLOPS (Floating point Operations Per Second). [14]

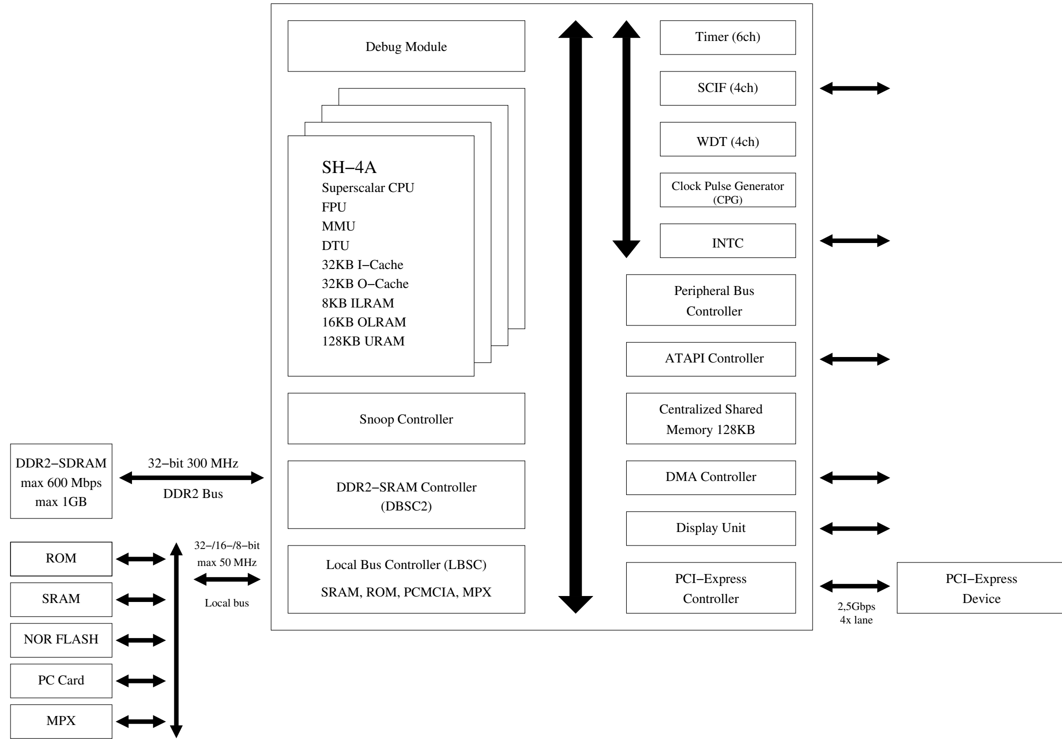


Figure 3.3: Block diagram of the RP1. [39]

RP1 architecture

The block diagram of the RP1 is shown in figure 3.3. The various hardware blocks included, like the four SH-4A cores, the peripheral and bus controllers are interconnected via a packet-based split-transaction system bus named SHwy. Processor core caches are interconnected via their own Snoop bus controlled by the Snoop Controller, separating the cache traffic from the more general SHwy bus and thus avoiding a traffic increase. The DDR2 memory is connected via a 32-bit DDR2 bus running at maximum 300 MHz. Other storage media like ROM, SRAM or PC cards are connected via the local bus that can be used in 8-, 16- or 32-bit mode and at maximum 50 MHz. The various peripheral controllers also have their according communication channels. [52] [39]

SH-4A CPU

The four SuperH architecture SH-4A 32-bit RISC (Reduced Instruction Set Computer) processor cores each include a Floating-point Unit (FPU), a Memory Management Unit (MMU) with a four-entry fully associative instruction Translation Lookaside Buffer (TLB) and a 64-entry fully associative unified TLB, an instruction cache and an operand cache in copy-back mode. The SH-4A cores are backwards

compatible on instruction set level to older models of the SuperH architecture (SH-1, SH-2, SH-3 and SH-4). [39] The cores all have a 7-stage pipeline and are superscalar having the ability to execute two commands simultaneously under certain constraints. [37]

The SH-4A cores have a RISC-based instruction set that is designed to be well suited for programming with the C language. Although the cores have an internal bus that is 32-bits wide, the instruction length is fixed to only 16-bits. Every core has sixteen 32-bit wide general-purpose registers, seven 32-bit control registers and four 32-bit system registers. [39] The cores have two operating modes, privileged mode and user mode. Shift from user mode to privileged mode happens when an interrupt is accepted or an exception occurs. [37]

The SH-4A cores included in the RP1 have some changes to the standalone SH-4A single-core processors, called SH-4A Extended Functions. The SH-4A Extended Functions are available for processors that have a value greater than or equal to H'40 in their Processor Version Register (PRV). The changes are mainly related to memory management and dealing with the other cores on the chip. [38]

Memory

RP1 has memory both dedicated for the individual cores as well as shared memory that can be uniformly accessed by all of the cores. Each core has their own MMU that supports 32-bit virtual addresses making the available virtual memory space 4 GBs big. The whole 4 GB is addressable from the privileged mode and 2 GBs are addressable from the user mode. There is also an 8-bit Address Space Identifier (ASID) that can be used to assign each simultaneously running process its own virtual address space. In privileged mode, the 4 GB virtual memory space is divided into five different areas called P0 to P4 that are distinguished by their cacheability and whether address translation is possible or not. [37]

Each SH-4A core has internal LRAM and URAM memory that can be accessed from the CPU and the FPU. The LRAM is high-speed memory that is divided into 8 KB ILRAM and 16 KB OLRAM. The URAM is 128 KB. The core internal memories and the chip internal data bus SuperHyway are connected through the cores Data Transfer Unit (DTU). The shared memories common to all the cores are 128 KB of Centralized Shared Memory (CSM), external DDR2 memory accessed through the DDR2-SRAM Controller (DBSC2) and various other external memory like SRAM, ROM or Flash cards that can be connected via the Local Bus Controller (LBSC). The CSM is accessed directly through the on-chip SuperHyway bus. Up to 1 GB of DDR2-memory can be connected via a dedicated bus that runs at maximum 300 MHz. External memory connected via the LBSC go through the local bus at maximum 50 MHz. [39]

Every SH-4A core has its own 32 KB instruction cache and 32 KB KB operand cache. All of the caches are connected to each other and to the Snoop Controller

(SNC) via a Snoop Bus to enable coherency check. The instruction cache coherency is ensured by software and the operand cache can be ensured by a MESI (copy-back) or an ESI (write-through) protocol. Operand cache coherency can be individually enabled and disabled and its operation protocol can be selected by each of the core individually. [38]

Interrupts

Interrupts of the RP1 are controlled by the Interrupt Controller (INCT). It controls the flow of interrupt requests to the SH-4A processor cores and keeps track of the interrupt source priorities. External interrupts can be set 15 levels of priority whereas on-chip peripheral modules can be set 30 levels of priority. INCT privileges interrupts according to their priorities.

The INCT has two modes of operation, Fixed and Automatic Distribution modes. In the Fixed Distribution Mode, interrupt mask registers for each CPU are set in a way that every one interrupt is distributed to one certain CPU. This mode suites applications that have regular interrupts that are best served by a dedicated CPU. In the Automatic Distribution Mode, the interrupts are distributed simultaneously to all of the CPUs. The first one to catch and acknowledge the interrupt masks the interrupt from the other CPUs and goes on to process the request. [39]

Interrupts among CPUs

The CPUs can also generate interrupts at each other CPU. Every CPU has a Register for Controlling Interrupt among CPUs (CnINTICI, n = 0..3) that can be written to from any other CPU. The registers are 32-bit wide and are divided in eight different 4-bit fields. Writing 1 to any of the bits generates an interrupt to the according CPU. The use of the fields is left to software and the can be exploited in two ways. The first way is to let every CPU write to a certain fixed field thus indicating the sender for the CPU that caught the interrupt. Another way is to let each of the fields have their own meaning and indicate the sender CPU by writing an individual code to the field. The fields can be set 16-level interrupt priorities in the Register for Setting Interrupt Priority Order among CPUs (CnICIPRI, n = 0 to 3). When received, the interrupt can be cleared by writing to the according field in the according Register for Clearing Interrupt among CPUs (CnINTICICLR, n = 0 to 3). [39]

Energy consumption

As RP1 is aimed at embedded systems with energy consumption constraints, the energy consumption of the RP1 can be adjusted in a couple of ways. There are three different power-down modes and the operating frequencies of the cores and buses can be individually adjusted.

The three power-down modes are *Sleep mode*, *Light sleep mode* and *Module standby mode*. Sleep mode and Light sleep mode both stop the CPU from operating, but in the Light sleep mode the on-chip memory is kept running whereas in the Sleep mode it is retained, saving even more energy. The sleep modes are entered by issuing a SLEEP machine command. The contents of the cores CPU Standby Control Register (CnSTRCR) define the sleep type. There are three ways to exit from both of the sleep modes, by obtaining an interrupt or by inflicting a power-on or manual reset.

The Module standby mode can be individually set for a range of on-chip modules, such as the Serial Communication Interface with FIFO (SCIF), individual Timer Unit (TMU) channels and Direct Memory Access Controller (DMAC). This stops the clock supply for the according module thus enabling software schemes for conservation of energy that would otherwise be wasted in maintaining a potentially unused module. The modules can be switched to and from module standby states individually by writing to the corresponding bits in the Standby Control Register 0 (MSTPCR0) or 1 (MSTPCR1).

The operating frequency of each of the four SH-4A cores can be individually set by selecting one of four available frequency ratios via a dedicated CPU Ick Frequency Setting Register (CnIFC, n = 0 to 3). As the available ratios range from x1 to x1/8, the frequency range with the maximum operating frequency of 600 MHz spans from the maximum 600 MHz to the minimum of 75 MHz. This makes it possible to save energy by running cores with less time critical computing with a lower clock frequency.

The Clock Pulse Generator (CPG) controls the frequencies of the internal and external buses of the RP1. The internally generated clock is run through a Phase Locked Loop (PLL) circuit that multiplies the clock rate by a factor of 72. After this the multiplied clock signal is routed through a divider circuit that is used to select the appropriate clock rates for the buses individually. The desired clock rate factors can be set individually to Frequency Control Register 1 (FRQRC1) and executed by writing a particular code to the Frequency Control Register 0 (FRQRC0). [39]

In [52] the total power consumption is estimated to drop as much as to 80% from the base case of 1.4 W by exploiting both the light-sleep mode and individual frequency adjustment in a test case simulation of a MPEG2 decoder benchmark in SMP mode using all the four cores.

MSRP1 baseboard and RP1 CPU board

The RP1 used in the prototype of DynOS SPUMONE was attached to a baseboard called MSRP1BASE02 produced by Hitachi ULSI Systems Co.,Ltd. The baseboard provides the necessary infrastructure for the RP1 processor to work and a lot of optional feature like communication channels to be used for various prototyping activities. The MSRP1 is powered by an external ATX power supply and provides electricity for the CPU-board on which the RP1 and the other parts of the base-

board are mounted. The CPU-board contains the external memories and four serial communication ports to connect the RP1 to a console on a host PC. The baseboard contains a 100BASE-TX Ethernet, a USB connection and some prototyping aids like user programmable LEDs, an 8-character display and switches. [18]

3.3 SH-Linux with CPU-hotplug

In this section SH-Linux, an implementation of Linux for the SH-architectures, implementation, is introduced. The GPOS in the DynOS SPUMONEs prototype is SH-Linux. The section starts with a brief overview of the SH-Linux and after that, two of the most important aspects of SH-Linux in regard to DynOS SPUMONE; the multi-core boot-sequence and the operation of the CPU-hotplug functions, are discussed in detail.

Overview

Linux for the SuperH family of processors is called SH-Linux. Ports are available in the main tree of Linux for a wide range of SuperH processors ranging from the early SH-2 to the modern SH-4 architectures. The SH-Linux used for the prototype implementation of DynOS SPUMONE was an experimental 2.6.16-sh kernel with RP1 baseboard support enabled. The version was not publicly available and was received from Renesas Technology with the RP1 board. Further an experimental patch was applied in order to enable the CPU-hotplug functionality to be used for the RP1. Also this patch was received exclusively from Renesas Technology and it is not publicly available.

The 2.6.16-sh Linux is compiled with the GNU Compiler Collection (GCC) version 3.4.5 using a patched version of the GNU C Library (GLIBC). The patch is based on the 2.3.3 version of GLIBC, but it does not include the Native POSIX Thread Library (NPTL). There is practically no documentation for the SH-specific parts available. Therefore the information presented here is extracted straight from the source code.

Multi-core boot-sequence

The multi-core boot-sequence of the SH-Linux is quite straight-forward and contains both architecture-independent parts from the Linux main branch as well as some architecture-dependent parts for interacting with the SH-hardware. SH-Linux always boots core 0 first and uses it as a master CPU to set up the other cores that are referred to as slaves here. There is a number of CPU-maps that control the assignment of the cores and the correct order of booting. They are bitmaps that have one bit corresponding to one CPU. These maps and their purposes are listed in table 3.1. Three of the maps are common to all Linux versions and four are specific to

Origin	Name	Purpose
general Linux	`cpu_possible_map`	populatable cores
	`cpu_present_map`	populated cores
	`cpu_online_map`	cores available for the scheduler
SH-Linux specific	`cpu_initialized_map`	initialized cores
	`cpu_callout_map`	core 0 calling a freshly started core
	`cpu_callin_map`	other cores replying to core 0
	`smp_commenced_mask`	cores that are preparing for online

Table 3.1: CPU-maps of SH-Linux.

the SH-architecture, although some of the SH-specific maps have their counterparts in other some architectures. The multi-core boot-sequence is implemented using the SH-specific map-declarations.

The boot process of SH-Linux is illustrated in figure 3.4. In the beginning, a boot loader loads the kernel to the memory. After it has finished, the first CPU is activated by running the `start_kernel` function. It sets up the cpu-maps, the kernel thread, interrupts, caches, memories, the scheduler, etc. After everything is set up, a kernel thread `init` is initialized. The scheduler is then started and the `cpu_idle` idle-function is called.

As soon as the scheduler wakes up, it schedules the `init` kernel thread to run. First, the kernel is locked by calling the `lock_kernel` function, also called The Big Kernel Lock or BKL. This ensures that the awakening of the other cores and the initialization of other kernel functions does not inflict any concurrency problems. After obtaining the lock, the boot of the other cores can start and architecture-dependent code is entered by calling the `smp_prepare_cpus` function.

In the architecture-dependent part, the actual booting of the other cores is done one by one by the function `do_boot_cpu`. First, the master core prepares the slave core by allocating an idle thread for it. The entry point for the core is set to a location that contains an initialization function. Then the master core issues a soft reset to the slave which starts executing. After this, the master core sets the `cpu_callout_map` for the slave core and starts a wait of maximum 5 seconds. If during this time the slave core has not replied to its location in the `cpu_callin_map`, the master core concludes that something went wrong and tries to bring the slave core to a halt. If the slave core comes up without problems and replies successfully to the `cpu_callin_map`, the master core goes on to repeat the process for the rest of the slave cores.

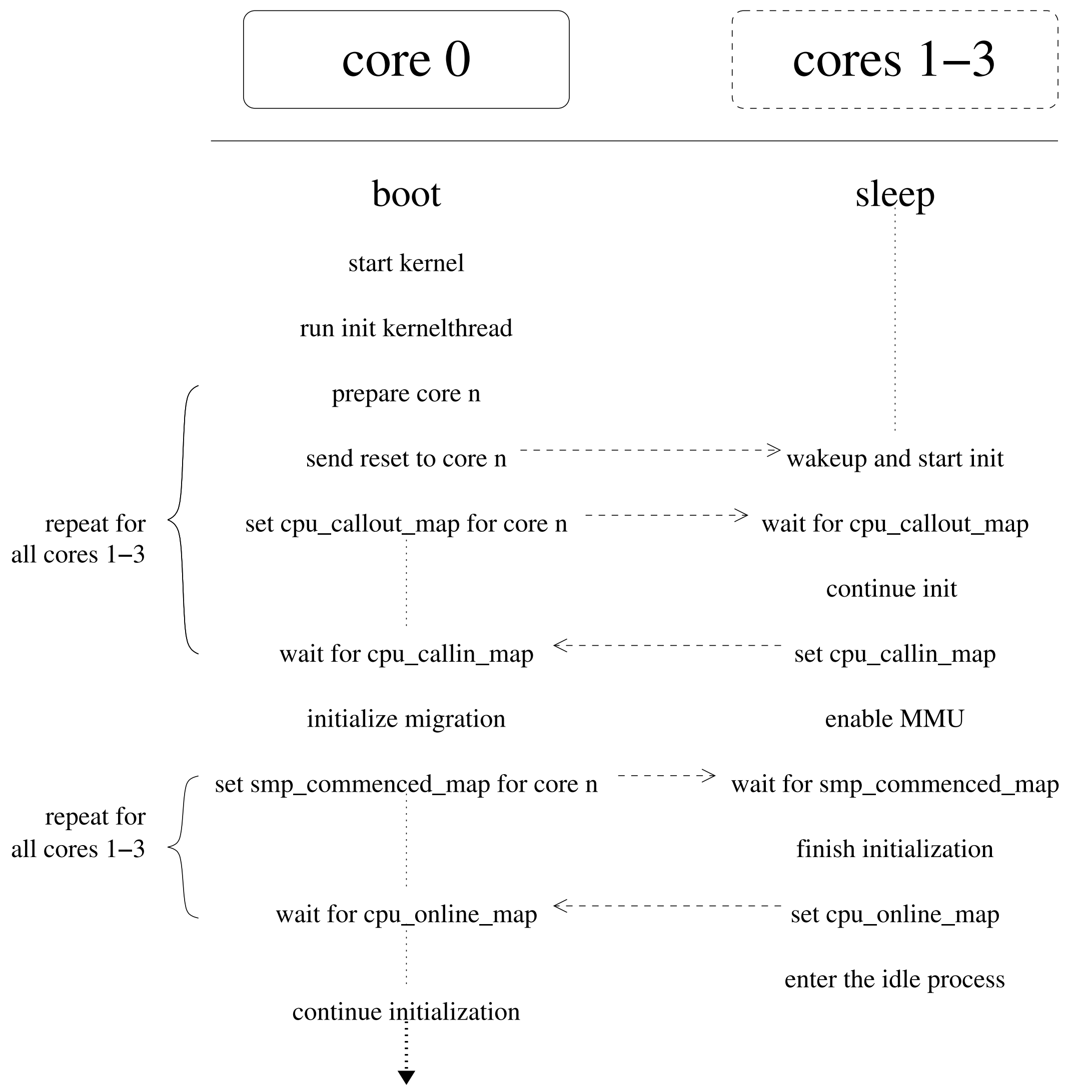


Figure 3.4: SH-Linux multi-core boot-sequence.

After the master core has finished waking up all of the slave cores, it initializes migration, a process that takes care of moving processes or threads from one core to another. The slave cores have meanwhile initialized themselves and enabled their individual MMUs and are already waiting for the master core to unleash them by writing to their locations in the `cpu_commenced_mask`. After the master core has done this, the slave cores finish up their initialization and start up their idle threads, finally changing their status in the `cpu_online_map` to online. The master core waits for all of the slave cores to become online and finishes the whole initialization after that.

CPU-hotplug

The CPU-hotplug functionality in SH-Linux, like in any other multi-core Linux, is accessible to the user through a device file. E.g. the device file for the core 1 is be located in `/sys/devices/system/cpu/cpu1/online`. The cores can be hot-removed or hot-added simply by writing a '0' or '1' to the corresponding device file. However in the RP1, core 0 can not be hot-removed. This is due to it having to always receive the non-maskable interrupt (NMI). Other cores can mask the interrupt and can thus be guaranteed to not wake up abruptly after being logically removed from the system.

The implementation of the hot-remove function is quite straight-forward. After receiving the hot-remove request through the device file, the kernel starts to prepare for removing the core. This includes notifying all parts of the kernel that the core is being removed so that it wont be interacted with anymore. Then a `take_cpu_down` process is run on the core to be hot-removed. This process sees to that all interrupts are migrated away from the core, local timer is stopped, the caches are flushed and the core is removed from the `cpu_online_map`. Upon finishing, the process requests the scheduler to schedule the idle task.

When the idle task runs on the core that is to be removed, it notices that it is running on a core that is declared offline and calls the `cpu_die` function. This function disables interrupts for the core, exits the idle task, removes itself from all of the CPU-maps, flushes the local cache, disables its MMU and cache and turns the core off by executing the processors SLEEP command.

The implementation of the hot-add function does not differ much from the multi-core boot procedure described above. Upon receiving the request to hot-add a core, the kernel sets up a task that will hot-add the core and schedules it. This task calls the same `do_boot_cpu` function that was used in the initial boot process to bring the up slave cores one by one. Similar as there, the core to be hot-added is allocated an idle thread and set up with an entry point to an initialization function before being issued a soft reset. Upon being reset, the initialization function is run. The core comes eventually up and starts running the idle thread, making it again available for the kernel scheduler.

3.4 TOPPERS/JSP

The RTOS used in the prototype implementation of DynOS SPUMONE is TOPPERS/JSP. As there was no need to be further modify TOPPERS/JSP for the prototype, discussion of its internal architecture and features here is omitted. Thus only a limited introduction to its background is given.

TOPPERS/JSP is a real-time operating system kernel designed for embedded systems. It is developed in the Toyohashi University of Technology and is used widely for various embedded designs ranging consumer electronics to cellular phones in Japan. The kernel is based on the μTRON 4.0 specification. This specification is part of the ITRON series of specifications for real-time kernels. The ITRON series is a de-facto standard for real-time kernel specifications in Japan. The TOPPERS/JSP is fully open source and is published under a TOPPERS licence, aimed at making the use of TOPPERS/JSP possible both in academia and industry. [49]

Chapter 4

Implementation

The hands-on contribution of this thesis work is the prototype implementation of DynOS SPUMONE. The implementation is not a single piece of software or hardware, but a bundle of patched software running on experimental hardware, working together to produce a proof of concept of the approach introduced in chapter 1.2. In this chapter we present the operating principles and usage of DynOS SPUMONE, the various modifications made to the building blocks of the system and the custom made parts. Verification and evaluation of the prototype are presented in chapter 5.

The core of the implementation is SPUMONE and its multi-core extension that are described in chapter 3.1. In order to get SPUMONE working with dynamically changeable OSes, some modifications and additions to the source code were necessary. Also the other guest OS, SH-Linux, was modified and a kernel module that acts as the run-time user interface towards SPUMONE was implemented.

The work was done during 2008-09 working as a research assistant and a SPUMONE team member in the Operating Systems Research Group of the Distributed and Ubiquitous Computing Laboratory in Waseda University.

4.1 DynOS SPUMONE

Basic operation flow

The basic operation flow of DynOS SPUMONE is very straight-forward. After the startup, Linux has populated all the cores. All cores except core 0 can be hot-removed at any time. When a core is removed from Linux, SPUMONE puts it to sleep. Now there are three possibilities to do with a sleeping core. The core can be left sleeping (for e.g. to conserve energy), it can be reassigned to Linux by a hot-add, or SPUMONE can be used to launch TOPPERS/JSP on it. When launched, TOPPERS/JSP starts its execution normally and operates undisturbed until it finishes all of its jobs. Then it signals SPUMONE about not being needed

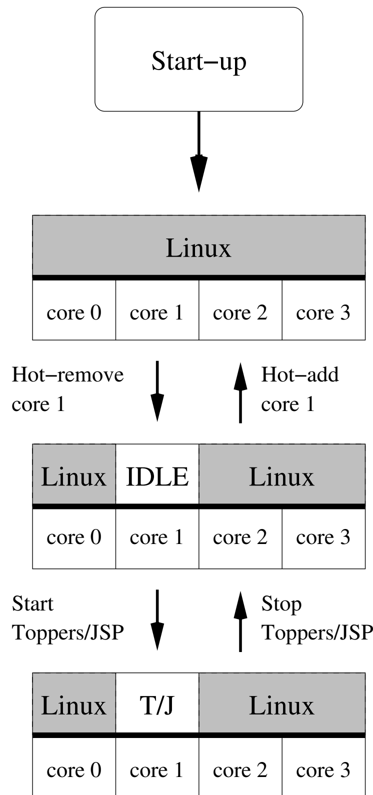


Figure 4.1: The basic operation flow of DynOS SPUMONE.

anymore and SPUMONE puts the core to sleep again. Now the core can be used to launch TOPPERS/JSP again, be reassigned to Linux or be left to sleep. The basic operation flow is illustrated in figure 4.1.

User interface

The user interface to DynOS SPUMONE consists of five basic functions: *initialization*, *CPU-hotremove/-hotadd* and *start/stop RTOS*. The first four functions are accessed in Linux and the last one is executed from the RTOS. A summary of the functions, their operation and execution is in table 4.1.

The *initialization* that has to be done in user mode consists only of inserting the DynOS kernel module to the running Linux kernel and creating a device node file for it, if not already present, to the file system. These are done by using the common Linux commands `insmod` and `mknod`. The commands to issue are `insmod dynos.ko` to insert the kernel module and mknod `/dev/dynos1 c 254 1` (/dev/dynos1 for the file name, c for character class, 254 for the major number and 1 for the minor number of the device).

Function	**Operation**	**Execution**
Initialize	Insert Dynos kernel module and create a DynOS device file	Linux-commands `insmod` & `mknod`
CPU-hotremove	Remove a core from Linux	Write '0' to CPU's online-file
CPU-hotadd	Add a core to Linux	Write '1' to CPU's online-file
Start RTOS	Start the RTOS on a core	Write a command to the dynos-node-file
Stop RTOS	Stop the RTOS on a core	Call a SPUMONE system call from the RTOS

Table 4.1: DynOS user functions.

The *CPU-hotplug*-functions are executed by writing to the corresponding CPU's online file. For example for core 1 the online file is `/sys/devices/system/cpu/cpu1/online`. Writing a '0' to the file will remove the core from Linux, making it logically offline. Writing a '1' to the file will add the core to Linux, making it logically online. This mechanism is not specific to DynOS. It is included in all Linux's with CPU-hotplug support.

The *start RTOS* command is issued through the DynOS kernel module. The user sends a command to the kernel module by writing to the DynOS device node file that is initialized to `/dev/dynos1`. The user must specify an inter-processor interrupt (IPI) number 6 to be sent and the core it will be sent to, that is the core on which the RTOS should be started. The command for starting TOPPERS/JSP on core 1 would thus be `send core 1 ipi 6`. Upon receiving interrupt number 6, the SPUMONE instance on the target core starts up the RTOS.

The *stop RTOS* function is implemented only to TOPPERS/JSP. It is issued by simply sending an ASCII 'i' through the serial bus to the RTOS. This results in the RTOS stopping its timers and SPUMONE putting the core to sleep. This could be implemented in various ways depending on the application.

Implementation architecture

The architectural hierarchy of the prototype implementation is shown in figure 4.2. The hardware used is the RP1 four-core processor. SPUMONE resides directly over the hardware where it controls the execution of the guest operating systems. The slightly modified guest operating systems, TOPPERS/JSP as the RTOS and Linux as the GPOS, lay on SPUMONE. Real-time applications are run on TOPPERS/JSP and normal applications are run on Linux. Applications running on Linux can control SPUMONE by issuing commands via the DynOS SPUMONE custom kernel

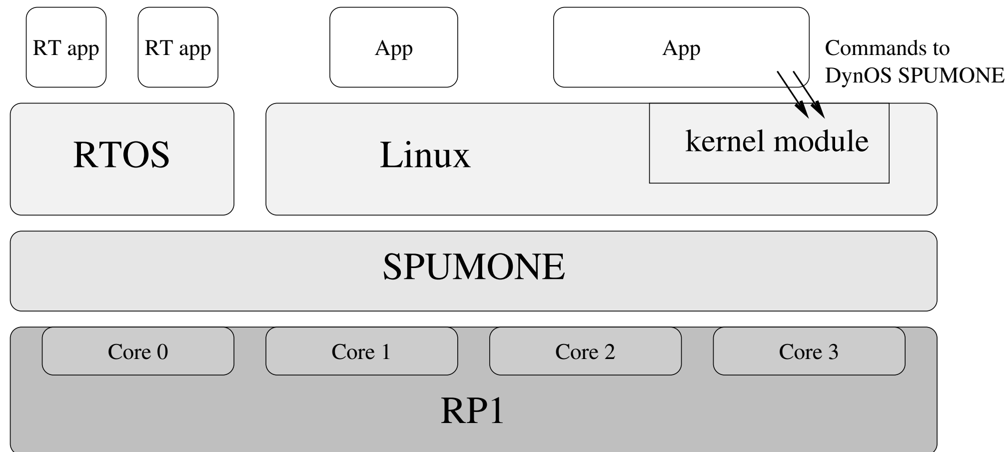


Figure 4.2: Architectural hierarchy of the prototype DynOS SPUMONE.

module. Virtual CPUs normally used by SPUMONE are not visible in the figure, as they are not used in the same sense in the prototype DynOS SPUMONE where one core is used to run only one OS at any given time. Virtual CPUs however exist inside SPUMONE on each core for controlling their state and for "storing" the currently unused OSes.

Boot process

The boot process of DynOS SPUMONE is very simple. When the processor is switched on, the boot loader loads master SPUMONE to core 0. It initializes itself and loads the slave SPUMONEs to the rest of the cores as described in chapter 3.1. SPUMONEs then initialize themselves, load Linux and TOPPERS/JSP to the memory and starts up the VCPUs on each core. On core 0, the VCPU for Linux is started. On the other cores an idle VCPU is started, so they stay waiting. The Linux boot process on core 0, then loads Linux to the other cores. Instead of the hard reset that unmodified Linux uses, the master SPUMONE on core 0 is then used to send interrupts to the other cores. Upon receiving the interrupts the slave SPUMONEs start the VCPUs for Linux and the Linux boot process goes on normally. When the Linux boot is finished, DynOS kernel module is added to the kernel. Last, the `/dev/dynos1` device file, the interface between the user and the DynOS module, is created, if not already present in the root file system.

In the current implementation, TOPPERS/JSP can be started only on core 1. This is not caused by architectural limitations but due to project time constraints that left no time for modifying the build process to support running it on other cores. In order to make TOPPERS/JSP available for all the cores, separate instances of it should be loaded to the memory for each core. This is necessary because TOPPERS/JSP uses hard coded physical addresses for some of its internal data.

4.2 Changes to SPUMONE

SPUMONE was originally built to enable running two operating system statically on one processor. The multi-core extension allows this to be done on multiple cores and enables running operating systems in their own multi-core configurations. What is not possible with the original versions is to assign and reassign operating systems to cores at run-time. In order to make this possible, some modifications to the source code of SPUMONE and the multi-core extension were needed.

As the concept of DynOS SPUMONE is to assign whole cores at need to run an operating system, the original feature in SPUMONE of running two operating systems simultaneously on one core was dropped to make the design simpler and more robust. However, there are no architectural barriers of using such a scheme also with DynOS SPUMONE.

Modifications

Three major modifications to multi-core SPUMONE were needed in order to enable dynamic assignment of the cores. The initial boot sequence had to be modified, an interrupt transmit mechanism between the guest OSes and SPUMONE had to be built and a way to record Linux timer-state had to be implemented.

The initial boot sequence of the original multi-core SPUMONE left the slave cores in an idle loop after initialization, from where they were transferred to run guest OSes on VCPUs upon receiving an interrupt. For DynOS SPUMONE this was rewritten so that the slave cores dispatch the idle VCPUs upon finishing initialization. This modification was necessary because the state of the idle VCPU was not sufficiently well initialized if not properly run first. This caused no problems when the guest OS allocation was static and the idle VCPU was not summoned, but for the DynOS SPUMONE case of dynamic allocation, where the idle VCPU is summoned after removing an OS from the core, the system functioned incorrectly. The reason for the incorrect function was tracked down to a bug in the VCPU handling mechanism of SPUMONE and was corrected by SPUMONEs main developer.

In order to enable fast signalling between the OSes and SPUMONE, a simple IPI transmit mechanism between cores was implemented. A guest OS can issue a SPUMONE API call to send IPI to other cores. Upon receiving an API request to send an IPI to another core, SPUMONE uses the processors standard instructions to translate the request in to a real IPI to the proper core. Upon receiving an IPI, the SPUMONE at the receiving end decodes the IPI and decides if it is forwarded to the guest OSes or masked from them. TOPPERS/JSP is launched on a core using this mechanism. The user issues a SPUMONE API call through the DynOS kernel module, which is translated to a IPI that stands for launching the RTOS and is sent to the core defined by the user. Upon receiving the interrupt, the target SPUMONE decodes the IPI concluding that it was meant for it instead of

the guest OSes and launches TOPPERS/JSP on the core.

Linux's timer-state changes with different combinations of cores that are online in the kernel. As running TOPPERS/JSP between Linux's CPU hot-remove and hot-add changes the timer settings without Linux knowing about it, Linux will behave unpredictably after the core has been hot-added back to it. Thus a mechanism to preserve and restore the timer state of Linux had to be implemented and deployed between the CPU hotplug operations and running the TOPPERS/JSP. When SPUMONE receives an IPI that commands it to start TOPPERS/JSP it first automatically saves the state of the timers. A variable in every SPUMONE instance keeps track of the execution status of TOPPERS/JSP on that core. When dispatching Linux, SPUMONE first checks from the variable if TOPPERS/JSP has been run. If true, it loads the timer state that was recorded before TOPPERS/JSP was run, and if not true, nothing is done and Linux is dispatched as normally. The check is important, as during the initial boot sensible timer state data is naturally not available and issuing a random timer-state would of course make very little sense.

4.3 Changes to Linux

As SH-Linux is already para-virtualized for multi-core SPUMONE, only the CPU hotplug functionality was left to be modified for DynOS SPUMONE. In addition, a normally unused interrupt was forced to function as a normally assignable interrupt in the Linux kernel. This was then used for receiving interrupts from SPUMONE or from TOPPERS/JSP to the Linux kernel. The functionality was used during development to enable TOPPERS/JSP to request a boot for itself via sending an interrupt to Linux. In order to make this work, the interrupt table of Linux and the interrupt setup process had to be slightly modified.

For DynOS SPUMONE to function correctly, the CPU-hotplug functions had to be modified so that they do not execute instructions that would affect other OSes running on the same core. For the hot-add function, nothing had to be modified, as the only core-specific function it calls is the same `do_boot_cpu`-function as the initial multi-core boot process, which was already modified to use SPUMONEs virtual reset function instead of the native `sh_send_reset`-function that issues a software reset for the whole core.

The CPU hot-remove function was modified so that it will leave the core in such a state that SPUMONE can continue operating on it after Linux has finished removing the core from itself. In order to achieve this the `cpu_die`-function had to be modified not to disable interrupts, MMU and cache, and not to issue a software reset to the core that would set the core in to sleep mode. Instead, the function was modified to last call SPUMONEs `spumone_vcpu_suspend`-function that puts the core in to idle mode without disabling interrupts by suspending the current virtual CPU.

4.4 DynOS kernel module for Linux

The DynOS kernel module for Linux was implemented from scratch. It is a standard character class device that is operated through writing ASCII data to the device-file that is created to the `/dev` directory. As device drivers normally, also DynOS kernel module is not running all the time in the background but is run only when reads or writes to the device-file happen. If the file is read, DynOS only returns the contents of its read-buffer, that is nothing. If something is written to the file, a very simple parser is invoked. This parser is implemented simply as a chain of if-else-statements that check if the input is a valid command for DynOS, as defined above in this chapter. If the command is valid, it is executed, otherwise an error message is returned to the user.

During initialization, the kernel module attempts to claim a "major number", which is used to identify devices inside the kernel, and reserves itself the needed amount of memory. After this it sets up a normally unused interrupt number to receive interrupts from SPUMONE as described in the above section.

Chapter 5

Verification and evaluation

In this chapter we present the verification of the prototype DynOS SPUMONE implementation and present an evaluation of its performance and engineering costs. The verification was made by executing a test run with a fully functional system on real RP1 hardware. The evaluation comprises of two parts: performance and engineering costs. Performance evaluation of the system is further divided down into three parts: switching time between OSes, Linux schedulers performance on multi-core SPUMONE and the delay overhead in interrupt handling for the TOPPERS/JSP. Engineering costs were rated by calculating the amount of modifications needed to build the system. For evaluation also results from earlier SPUMONE research are used. The results presented in this chapter are further discussed in chapter 6.

5.1 Verification

The test setup used for the verification is summarized in figure 5.1. It consisted of the RP1 processor mounted on the MSRP1-board and a host-PC. The board was connected to the host-PC through Ethernet and two serial buses. Ethernet was used to provide a network boot to the RP1 and to provide a root file system for the system on the host-PC via the Network File System (NFS) protocol. The serial buses were used to provide a user interface to the RP1 through terminal emulation from the host-PC. Serial bus 1 was assigned to the master SPUMONE running on core 0 and Linux. Serial bus 2 was assigned to slave SPUMONEs on cores 1 to 3 and the TOPPERS/JSP on core 1. On the host-PC two instances of a serial communication program called *Minicom* were used for communication through the serial buses. The communications from both terminal connections were saved in to log files.

Linux was run as the root and commands to SPUMONE were given manually in the login shell through predefined shell scripts for hot-removing and hot-adding cores and for booting TOPPERS/JSP on core 1. TOPPERS/JSP ran always a simple

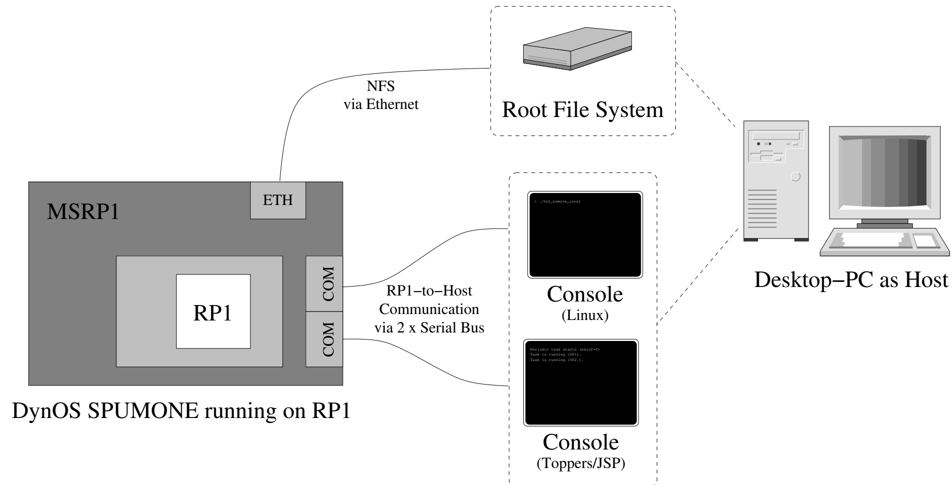


Figure 5.1: Verification test setup.

test program that looped periodically and reported to the serial port. The user could send commands to the program through the serial connection. Sending the command 'i' stopped the execution of TOPPERS/JSP and the test program and returned control of the core to SPUMONE.

The verification run were conducted manually. First, the MSRP1-board was manually reset. The system was booted through the network, with all files residing in the NFS on the host-PC. First SPUMONE was loaded to core 0 and it went on to boot the other cores and started Linux on core 0. Linux again loaded itself to the remaining cores and upon finishing initialization opened up a login shell. The tests were manually driven from the login shell through invoking the predefined shell scripts. Testing consisted of hot-removing cores, hot-adding them after hot-removes and running TOPPERS/JSP on core 1 after hot-removing it from Linux. The commands were run multiple times and in arbitrary order to make sure that the system works in various situations and is not dependable on execution order of DynOS SPUMONE functions. The online-status of the cores from Linux's viewpoint was ensured by polling Linux's system information. Upon launching TOPPERS/JSP and the test program, it was let to run for a while before sending the 'i' command through the serial connection.

During the verification run no defects were found and the system executed flawlessly.

5.2 Performance

The switch time between Linux and the TOPPERS/JSP test program was measured from the verification run. The switch time consists of two phases, the Linux hot-add or hot-remove and starting or stopping the real-time task on TOPPERS/JSP.

Function	Average execution time	Standard deviation
hot-add	0.43 s	0.02 s
hot-remove	0.50 s	0.27 s

Table 5.1: Linux core-hotplug execution times on RP1.

The switch cases were not combined into one command because DynOS SPUMONE supports hot-removing cores both from Linux and from TOPPERS/JSP without starting the other guest OS. This provides a simple but effective way to do power management by shutting down unnecessary cores.

The execution times for hot-add to Linux and hot-remove from Linux were directly calculated from the time labels of Linux kernels printk-messages that are used to print debug-information. There are printk-messages at the start and end of the hotplug-processes so the measurement is quite accurate. Both hot-add and hot-remove were executed 6 times during the verification run. Average and standard deviation were calculated from the results. The results are shown in table 5.1. The startup and shutdown times of the test program running on TOPPERS/JSP were not measured as both of them were observed to be extremely short. The startup takes some time as the kernel is booted every time, but the shutdown takes virtually no time at all, as all it does is to disable the interrupt that is waking the test process up thus releasing the control of the core back to SPUMONE.

The impact multi-core SPUMONE has on SH-Linux's multi-core performance on RP1 is analyzed in [22]. The measurements were conducted with a program called *hackbench*. It was developed to measure Linux schedulers' scalability for multi-core processors and works by creating multiple processes that send and receive data from each other. [42] In [22] several measurement cases of multi-core SPUMONE using *hackbench* are described. There are two cases that are relevant for DynOS SPUMONE. The first one is running Linux on all four cores and the second one is running Linux on three cores and TOPPERS/JSP on the fourth, all OSes above multi-core SPUMONE, respectively. The first one is equivalent to the situation in DynOS SPUMONE when Linux controls all of the four cores, e.g. the state after start-up. The latter case is equivalent to the situation where one core has been hot-removed from Linux and TOPPERS/JSP has been started on it. The rest of the cases have one core simultaneously shared between Linux and TOPPERS/JSP and are thus not relevant for DynOS SPUMONE.

The tests were executed simply by timing the execution of the *hackbench* program with different amounts of processes. The results in [22] show an almost linear trend in execution time growth in relation to growth of the amount of processes ran by *hackbench* for both cases. The difference between the execution time of the two cases also grows linearly. All of the tests execute faster in the case where Linux controls

Configuration	Average (clock cycles)	Worst (clock cycles)
Native	102	102
Running on SPUMONE	367	1582

Table 5.2: Timer interrupt handling delay in TOPPERS/JSP on a SH-4A. [24]

all of the four cores.

SPUMONEs impact on the real-time responsiveness of TOPPERS/JSP has been evaluated in [24]. The delay in handling timer interrupts was measured for TOPPERS/JSP running both natively and with SPUMONE and the overhead was approximated by calculating the difference. The test was made on a single-core SH-4A processor that has the same architecture as the RP1. 20000 interrupts were measured. The results are summarized in table 5.2. It can be seen, that the overhead in using SPUMONE is on average 265 clock cycles and in the worst case 1480 clock cycles. The article states that *"The worst case overhead shows the time required to save the state of Linux and restore the state of TOPPERS."*, making the worst case scenario irrelevant for the DynOS SPUMONE use case. The relevant average delay for the DynOS SPUMONE use case will thus also be even lower than the figure presented because TOPPERS/JSP is executed on its own core thus eliminating the need to record Linux's state between TOPPERS/JSP timer interrupts.

5.3 Engineering costs

Two cases are considered for evaluating the engineering costs. The engineering cost for implementing DynOS SPUMONE itself and the engineering cost for implementing a complete system running on DynOS SPUMONE. A rough but adequate approximation of required engineering costs was made by counting the number of lines of code (LOC) that were modified, added or removed. The count includes source files, header files and makefiles. Engineering costs for implementing DynOS SPUMONE can be thought to consist of the size of the SPUMONE code base and the required modifications to the SPUMONE code base and the guest OSes. When considering the engineering costs for implementing a system on DynOS SPUMONE with new OSes, only the required modifications to the guest OSes are relevant.

Multi-core SPUMONE is the foundation on that DynOS SPUMONE is built. The engineering costs for modifying the guest OSes Linux and TOPPERS/JSP to run on multi-core SPUMONE are calculated in [22] and shown in table 5.3.

For DynOS SPUMONE, modifications were made to the multi-core SPUMONE and to the guest OS Linux. For TOPPERS/JSP, no modifications specific to DynOS SPUMONE were needed. In addition, the DynOS kernel module for Linux was built.

Component	Lines changed
SH-Linux	26
TOPPERS/JSP	38

Table 5.3: Code changes per guest OS for multi-core SPUMONE. [22]

Component	Lines removed	Lines inserted
SPUMONE	242	599
SH-Linux	5	10
TOPPERS/JSP	-	-
DynOS Kernel module	-	249

Table 5.4: Code changes per component for DynOS SPUMONE.

All the code changes needed for the prototype DynOS SPUMONE implementation are summarized in table 5.4. For SH-Linux only code changes additional to the changes required by the multi-core SPUMONE were counted. TOPPERS/JSP was not modified. The DynOS kernel module was built from scratch so no lines were removed.

Chapter 6

Conclusions

In this thesis DynOS SPUMONE, a new concept of utilizing the parallel processing power of embedded multi-core processors in a new and effective way, was presented. In this chapter we discuss the concept, the prototype implementation and its verification and evaluation. Last, suggestions for future work building on the DynOS SPUMONE concept and/or implementation are given.

6.1 Discussion

The need for a new approach to embedded system design stems from the fact that the evolution of processors has reached a state where the design paradigm is shifting from optimizing single-threaded systems to providing more and more parallel computing power. The reason for this is that it is not anymore feasible to increase frequencies of the processors, because the increasing heat dissipation is becoming too expensive to accommodate to. As application level software is still being mainly developed with old paradigms suitable for single-threaded machines and no major shift is seen in the near future, there is a genuine need for system-level solutions that would make better use of the constantly growing parallel computing power in embedded systems.

The DynOS SPUMONE concept was compared with related commercial and scientific work done on the field in chapter 1.3. Although similar concepts are being researched, DynOS SPUMONEs concept proved to be unique in a number of ways. Its main focus is not in isolation-provided security and reliability like e.g. OKL4 but instead in low engineering costs and in absolute real-time responsiveness, making the approach directed at closed embedded systems with hard real-time constraints.

To provide a proof of concept, a prototype implementation of the DynOS SPUMONE was built, verified and evaluated. All of the phases are described in chapter 4. The prototype was built on RP1, an embedded multi-core processor, and was based on multi-core SPUMONE, a lightweight virtualization layer. Despite being an early prototype, no defects were found in the verification. Already this implies, if only lightly,

that implementing a well-functioning system based on the DynOS SPUMONE concept can be done with moderate engineering costs.

Further, evaluation of the engineering costs in 5.3 indicate that porting a guest OS for the DynOS SPUMONE has also an extremely low cost, the whole process consisting of less than 50 lines of code changes. Compared with other platforms with similar goals e.g. OKL4, RTLinux or RTAI, porting Oses to DynOS SPUMONE is roughly two magnitudes cheaper measured in lines of code [24]). This comes however at the expense of decreased isolation between the guest OSes. DynOS SPUMONE masks only an absolute minimum set of necessary commands from the guest OS. Other VMMs mask all the commands that could potentially lead to interferance between OSes running on the system. This lack of isolation between guest OSes in DynOS SPUMONE indicates that it is best suited for closed devices, where the system designer has all the control over the implementation of the system. The extremely small size of the DynOS SPUMONE code base indicates a smaller possibility for unnoticed design defects, as the amount of bugs in a program is generally assumed to be proportional to the size of the code base ([9]).

DynOS SPUMONEs suitability for systems with hard real-time constraints is considered by the interrupt handling delays experienced by the RTOS, evaluated in chapter 5.2. The results from [24] are already good and adequate for a wide range of real-time applications, but for DynOS SPUMONE the real delays can be expected to be even shorter, as there is no core-sharing between the GPOS and RTOS but instead when running real-time tasks one core is reserved wholly for the RTOS until it turns over the control of the core itself. There is practically no interference whatsoever while the RTOS is running. This makes DynOS SPUMONE suitable for any real-time tasks that a RTOS running natively on the same hardware could execute.

Another metric of interest is the switching time from one OS to another, that is how long delay there is in starting the RTOS from the GPOS and vice versa. From the results in chapter 5.2 it can be seen that for the prototype virtually the whole delay is consisted of Linux core-hotplugs' delay. Even in its current state the delays are acceptable for a wide range of applications, e.g. for the mobile phone in the example application concept from chapter 1.2, a less then one seconds delay in starting the camera is nothing unusual in todays mobile phones with cameras. However there are applications and platforms for which even such a delay in starting up would be intolerable. If real-time applications are run frequently and they last only a short amount of time, the startup delay costs might be unreasonable. In the current implementation no effort has been made to optimize the startup time thus this could be addressed in future work.

From the above discussion it can be concluded that the DynOS SPUMONE concept is a promising way to make better use of the parallel cores appearing lately also in embedded systems. While it is certainly not suitable for all systems due to having no isolation between the OSes, it outperforms other systems in real-time capabilities and can be deployed where hard real-time performance matters the

most and where there is also a need to take advantage of the rich set of features provided by a modern GPOS. Using DynOS SPUMONE in embedded devices could aid in reducing manufacturing costs. Real-time tasks that are running on special chips could be ported to run on one core in a multi-core chip thus eliminating the need to place costly special chips on the design at all. Many systems could also benefit from the flexibility to choose the best OS for any given job. Getting legacy real-time applications running on legacy OSes running on newer platforms could also be simplified by eliminating the need to port the application to a new OS and instead just run it on a dedicated core when needed.

6.2 Proposals for future work

DynOS SPUMONE is a novel concept that aims to make better use of the parallel processing power provided by the increasingly available multi-core processors for embedded systems. What distinguishes it from other approaches summarized in chapter 1.3 is its really small footprint coupled with its minimal intrusion to the guest OSes. Emphasis has also been put to requiring as little engineering effort as possible for deploying DynOS SPUMONE to use in real embedded systems. These principles should guide also future work building on DynOS SPUMONE.

The concept could be further explored by introducing a memory-hotplug functionality either to SPUMONE or to the guest OSes, improving the usage of the physical memory. Virtual memory would also be useful for many tasks, e.g. loading guest OSes from arbitrary memory locations without modifications to the build process, but might result in significant drop in real-time performance. An efficient and easy to use scheme of conveying data between the guest OSes through the shared memory would make the idea of using RTOSes for controlling data-intensive devices, such as cameras and radios, very feasible.

Bibliography

[1] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference* (New York, NY, USA, 1967), ACM, pp. 483–485.

[2] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., and Zaharia, M. Above the Clouds: A Berkeley View of Cloud Computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM, pp. 164–177.

[4] Barr, M., and Massa, A. *Programming Embedded Systems: With C and GNU Development Tools.* O'Reilly Media, Inc., 2006.

[5] Betti, E., Bovet, D. P., Cesati, M., and Gioiosa, R. Hard Real-Time Performances in Multiprocessor-Embedded Systems Using ASMP-Linux. *EURASIP J. Embedded Syst. 2008* (2008), 1–16.

[6] Borkar, S. Thousand core chips: a technology perspective. In *DAC '07: Proceedings of the 44th annual Design Automation Conference* (New York, NY, USA, 2007), ACM, pp. 746–749.

[7] Bovet, D. P. *Understanding the Linux Kernel.* Safari Tech Books Online,, Boston, MA :, 2005. Safari Tech Books Online.

[8] Buzen, J. P., and Gagliardi, U. O. The evolution of virtual machine architecture. In *AFIPS '73: Proceedings of the June 4-8, 1973, national computer conference and exposition* (New York, NY, USA, 1973), ACM, pp. 291–299.

[9] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An empirical study of operating systems errors. In *SOSP '01: Proceedings of the eighteenth*

ACM symposium on Operating systems principles (New York, NY, USA, 2001), ACM, pp. 73–88.

[10] Clarke, P. Hitachi, Mitsubishi to merge chip businesses. *Embedded Systems Design* (March 2002). Online http://www.embedded.com/news/embeddedindustry/9900805?_requestid=480424, Retrieved January 12, 2010.

[11] Crosby, S., and Brown, D. The Virtualization Reality. *Queue 4*, 10 (2007), 34–41.

[12] Flynn, M. Very high-speed computing systems. *Proceedings of the IEEE 54*, 12 (Dec. 1966), 1901–1909.

[13] Gustafson, J. L. Reevaluating Amdahl's law. *Commun. ACM 31*, 5 (1988), 532–533.

[14] Hanawa, T., Sato, M., Lee, J., Imada, T., Kimura, H., and Boku, T. Evaluation of Multicore Processors for Embedded Systems by Parallel Benchmark Program Using OpenMP. In *IWOMP '09: Proceedings of the 5th International Workshop on OpenMP* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 15–27.

[15] Heiser, G. Do Microkernels Suck? In *9th Linux.Conf.Au* (Melbourne, Jan 2008).

[16] Heiser, G. Hypervisors for Consumer Electronics. In *Proceedings of the 6th IEEE Consumer Communications and Networking Conference* (Las Vegas, NV, USA, Jan 2009).

[17] Hennessy, J. L. *Computer architecture: A quantitative approach.* Morgan Kaufmann,, Amsterdam, 2007.

[18] Hitachi ULSI Systems Co., Ltd. *MSRP1BASE02 Users manual*, January 2008. Rev.1.1.

[19] Holgado-Terriza, J. A., and Viúdez-Aivar, J. A flexible Java framework for embedded systems. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems* (New York, NY, USA, 2009), ACM, pp. 21–30.

[20] Hwang, J.-Y., Suh, S.-B., Heo, S.-K., Park, C.-J., Ryu, J.-M., Park, S.-Y., and Kim, C.-R. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE* (January 2008), pp. 257 –261.

[21] Ishikawa, Y., Fujita, H., Maeda, T., Matsuda, M., Sugaya, M., Sato, M., Hanawa, T., Miura, S., Boku, T., Kinebuchi, Y., Sun, L., Nakajima, T., Nakazawa, J., and Tokuda, H. Towards an Open Dependable Operating System. In *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC '09. IEEE International Symposium on* (March 2009), pp. 20–27.

[22] Kanda, W. Design and Implementation of a Multi-core CPU Virtualization Technique for Embedded systems. Master's thesis, Waseda University, Tokyo, Japan, February 2009.

[23] Kanda, W., Yumura, Y., Kinebuchi, Y., Makijima, K., and Nakajima, T. Spumone: Lightweight cpu virtualization layer for embedded systems. In *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on* (Dec. 2008), vol. 1, pp. 144–151.

[24] Kinebuchi, Y., Morita, T., Makijima, K., Sugaya, M., and Nakajima, T. Constructing a Multi-OS Platform with Minimal Engineering Cost. In *Analysis, Architectures and Modelling of Embedded Systems* (2009).

[25] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, MT, USA, Oct 2009), ACM, pp. 207–220.

[26] Laplante, P. A. *Real-Time Systems Design and Analysis*, third ed. John Wiley & Sons, Inc., 2004.

[27] Lee, E. A. Embedded Software. *Advances in Computers 56* (2002), 56–97.

[28] Lee, I. *Handbook of real-time and embedded systems.* Chapman & Hall/CRC,, Boca Raton :, cop. 2008.

[29] LeVasseur, J., Uhlig, V., Yang, Y., Chapman, M., Chubb, P., Leslie, B., and Heiser, G. Pre-virtualization: soft layering for virtual machines. In *Proceedings of the 13th IEEE Asia-Pacific Computer Systems Architecture Conference* (Hsinchu, Taiwan, August 2008), Y.-C. Chung and J. Morris, Eds., IEEE Computer Society Press, pp. 1–9.

[30] Levis, P., and Culler, D. Mate: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2002), ACM, pp. 85–95.

[31] Marwedel, P. *Embedded System Design*, first ed. Kluwer Academic Publishers, 2003.

[32] McDougall, R. Extreme Software Scaling. *Queue 3*, 7 (2005), 36–46.

[33] Mokhoff, N. Intel, Nokia launch open software 'MeeGo' platform. *Embedded Systems Design* (February 2010). Online `http://www.embedded.com/products/softwaretools/222900416?_requestid=215778`, Retrieved May 4, 2010.

[34] Oliver, E. A survey of platforms for mobile networks research. *SIGMOBILE Mob. Comput. Commun. Rev. 12*, 4 (2008), 56–63.

[35] Olukotun, K., and Hammond, L. The Future of Microprocessors. *Queue 3*, 7 (2005), 26–29.

[36] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (1974), 412–421.

[37] Renesas Technology. *SH-4A, Software Manual*, October 2004. Rev.1.50.

[38] Renesas Technology. *SH-4A Multiprocessor-capable Functions, Software Manual*, September 2006. Rev.0.50 - Preliminary.

[39] Renesas Technology. *RP1 Hardware manual*, June 2007. Rev.0.51.1 - Preliminary.

[40] Rosenblum, M. The Reincarnation of Virtual Machines. *Queue 2*, 5 (2004), 34–40.

[41] Ruocco, S. A real-time programmer's tour of general-purpose L4 microkernels. *EURASIP J. Embedded Syst. 2008* (2008), 1–14.

[42] Russel, R. hackbench, 2003. Online `http://devresources.linux-foundation.org/craiger/hackbench/`, Retrieved May 2, 2010.

[43] Smith, J., and Nair, R. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[44] Smith, J. E., and Nair, R. The Architecture of Virtual Machines. *Computer 38*, 5 (2005), 32–38.

[45] Snol, L. More Smartphones Than Desktop PCs by 2011. *PC World* (September 2009). Online `http://www.pcworld.com/article/171380/`, Retrieved December 3, 2009.

[46] SPUMONE. Source code repository. Online `https://repos.dcl.info.waseda.ac.jp/spumone/trac`, Retrieved May 5, 2010.

[47] Stallings, W. *Operating systems: internals and design principles.* Prentice Hall ;, Harlow :, cop. 2008.

[48] Stanley-Marbell, P., and Iftode, L. Scylla: a smart virtual machine for mobile embedded systems. In *WMCSA '00: Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'00)* (Washington, DC, USA, 2000), IEEE Computer Society, p. 41.

[49] Takada, H. Introduction to the TOPPERS Project " Open Source RTOS for Embedded Systems. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (Washington, DC, USA, 2003), IEEE Computer Society, p. 44.

[50] Turley, J. The Two Percent Solution. *Embedded Systems Design* (December 2002). Online `http://www.embedded.com/shared/printableArticle.jhtml?articleID=9900861`, Retrieved November 18, 2009.

[51] VirtualLogix. Meeting the Challenges of Connected Device Design Through Real-Time Virtualization. White paper, June 2006.

[52] Yoshida, Y., Kamei, T., Hayase, K., Shibahara, S., Nishii, O., Hattori, T., Hasegawa, A., Takada, M., Irie, N., Uchiyama, K., Odaka, T., Takada, K., Kimura, K., and Kasahara, H. A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International* (Feb. 2007), pp. 100–590.