Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Programa de Maestría en Computación

Improving Operating Systems for Efficient Multimedia Handling in Embedded Systems

Thesis Proposal
Submitted in partial fulfillment of the requirements for the degree of

Magister Scientiæ en Computación

Author:
Diego Dompe

Advisor:
Ph.D. Francisco J. Torres-Rojas

May 2011

Resumen

Resumen en español

Abstract

CONTENTS

1. Introduction and background **1**
1.1. Introduction . . . 1
1.2. Background . . . 1
1.3. Problem statement . . . 5
1.4. Hypothesis . . . 5
1.5. Previous work . . . 5
1.5.1. Literature review . . . 5
1.6. Long-Range Consequences . . . 6

2. Objectives and Contributions **7**
2.1. General Objective . . . 7
2.2. Specific Objectives . . . 7
2.3. Contributions to the subject . . . 8
2.4. Scope and Limitations . . . 8

3. Methodology and Schedule **9**
3.1. Methodology . . . 9
3.2. Assumptions . . . 9
3.3. Procedure . . . 9
3.4. List of Deliveries . . . 10
3.5. Schedule and Work Breakdown . . . 10

Bibliography **11**

Acronyms **13**

LIST OF FIGURES

LIST OF TABLES

CHAPTER

ONE

INTRODUCTION AND BACKGROUND

> People in the embedded space don't do prototypes. They hack something until it works, then it's done.
>
> xav user on lwn.net

1.1. Introduction

1.2. Background

Embedded Systems are one of the fastest growing markets in the computer industry, with more than 10 billion embedded processors shipped in 2008 [2].

Embedded Software development is a discipline that is younger that software development for other environments like Personal Computers (PCs) or Mainframes. There are some particular issues that transform embedded software development into a complex eco-system:

- It usually requires some level of integration with the hardware that is non-common in other software development areas.

- Many of the hardware integration that is done is very particular and unique, which usually makes the code hard to re-use if isn't designed properly (or if the technology is new and isn't clear how to make implement a software solution in a generic way).
- The market for embedded system is so fast-paced that developers are in pressure to deliver working devices as soon as possible. Few companies care about the quality or re-usability of their embedded code, except by hardware vendors or efforts from single open-source minded developers who want to contribute back. For example recent articles regarding status of the contributions to the Linux™ kernel for the ARM architecture are described as a "mess" due many of different interests and forces behind it [4].
- Given the hardware integration requirements and the emergence of the field, many embedded developers may lack formal training on software development, but instead have an Electronic Engineer (EE) background.

These factors usually means that embedded developers will not likely find or apply the same best-practices that the rest of the software industry use. Is not rare to find embedded projects where each revision of the same product will have a complete new software stack or re-implementation from the previous iteration.

The fact that the code is tailored for specific hardware to the point where it can't be easily re-used goes against Operating Systems (OSs) theory which proposes that the OS should abstract all the software from the hardware implementation [5, p. 29]. However many embedded devices (especially ones with low memory footprint) may use a custom-made OS or use no OS at all (also called an OS-less solutions).

Linux has gradually gained acceptance as an OS for embedded systems due his open-source nature and support from embedded processors manufacturers, especially on the System on a Chip (SoC) market. SoCs are the main trend for new product development in recent years due the benefits it brings to new chip development [6].

The advantages that Linux provides for new embedded product development are clear:

- It's a probed OS which has been successfully used in different embedded markets.
- No royalty fees or licenses cost associated directly with it (although some embedded vendors may add royalty fees if you use their custom distributions).
- Wide knowledge base available and many developers are familiarized with it.
- Many hardware developers for peripherals provide drivers and support for Linux, simplifying the integration of hardware components. This is a critical element for minimizing the time to market of a product.

However Linux also presents some disadvantages for the embedded space:

- It's not designed to be a Real Time Operating System (RTOS), which make it not suitable for all type of embedded applications.
- There is a learning curve for RTOS developers, since Linux enforces some concepts that may be foreign to them:
  - Memory space protection: most RTOS do not provide memory space protection.
  - Separation between the kernel and user space applications: This is a side-effect of the memory space protection.
- There are several algorithms used in non-RTOSs like Linux that may affect the performance compared to RTOSs or OS-less solutions:
  - Context Switches: there is a penalty associated on several hardware architectures with the memory protection.
  - I/O Scheduling: the kernel page cache may defer I/O work. Linux offers full control to customized this behaviors for embedded systems, but many developers may be unaware of them.

- Interrupts prioritization: mainline Linux kernel doesn't offer a way to control priorities for interrupt handlers. There are patches to transform interrupt handlers into processes, but they aren't available for all kernel versions.

Despite the short-comings that Linux present versus a RTOS, just the fact of have widely available drivers and software stacks for peripherals on the embedded space outweighs these problems. Many SoCs may require complex software stacks to interact with integrated peripherals, such is the case of heterogenous SoCs using a General Purpose Processor (GPP) and a Digital Signal Processor (DSP) or other sort of hardware unit for accelerating specific operations (such as video coding, cryptographic algorithms, etc). Implementing or porting the functionality of these software stacks required to interact with the desired peripheral to an OS-less or other non-supported RTOS is usually non effective in terms of cost or time.

However many developers that are un-familiar with Linux and/or general purpose OSs design, approach the task of implementing the functionality for their embedded products bypassing Linux functionality. For example the author of this work have found several times kernel drivers for Linux designed just to specifically allow user-space access to the peripherals by any user space application in commercial Original Equipment Manufacturing (OEM)/Original Design Manufacturing (ODM) offerings for embedded development. There are several reasons for developers using this kind of approach:

- Some times the software stack was designed in OS-less environment and ported over to Linux. Developers just try to find a way to get the Linux hardware management out of the way of the software stack.
- Developers are simple un-familiar with Linux design.
- Linux may lack APIs or functionality to handle new scenarios posted by the hardware design. For example ability to realize zero-memory-copy operations.
- Developers try to use Linux's APIs but found that performance was un-acceptable for unknown reasons. They just assumed that is a problem with Linux being

bloated for desktop systems.

One common believe in the embedded community is that any software design highly specialized will provide better performance than a implementing it over a generic solution. One example of this is multimedia software architectures on top of Linux, where the amount of data being processed requires to use data paths with zero memory copy operations across the whole software stack.

1.3. Problem statement

Embedded developers tend to create software designs and architectures that aren't re-usable or maintainable under the premise that generic software architectures penalize performance. The particular example this work will focus on is multimedia software stacks due the high amount of data that it moves around the system and the soft real-time requirements of handling audio and video streams.

1.4. Hypothesis

It it's possible to optimize embedded Linux multimedia software stacks to provide good performance with negligible penalty compared to custom OS-less multimedia solutions, if architecture-specific optimizations are made on the Linux kernel, and the software stack is optimized with efficient algorithms.

1.5. Previous work

1.5.1. Literature review

It wasn't possible to find existing literature on the specific subject of this work, however we found related literature that may apply.

Liedtke [3] proved with his work that using the right algorithms and analysis it is possible to implement software stacks that manage hardware access with minimal performance penalties. He faced common belief that μ-kernels suffer inherit design performance penalties and demonstrated that rather than a design issue it was an implementation issue the root cause of the performance problems. Liedtke also showed that architecture-specific optimizations were required without loosing generality in the software architecture design.

If we characterize the main differences between Linux and RTOSs or OS-less solutions, we can identify previous work that addresses the performance and optimizations for these areas in embedded processors.

Memory Protection and Context Switching

Some paper found here: TODO [1]), instrumented performance penalties associated with context-switching in ARM processor architecture.

I/O Handling

TODO

Process Scheduling

TODO

1.6. Long-Range Consequences

This work will provide a foundation for analysis of the factors involved on adapting the Linux kernel for his use on embedded systems dealing with multimedia software stacks.

CHAPTER

TWO

OBJECTIVES AND CONTRIBUTIONS

> Don't find fault, find a remedy.
>
> Henry Ford

2.1. General Objective

- Prove that it is possible to improve multimedia software frameworks for embedded Linux to provide performance comparable with RTOS or custom OS-less solutions by tuning the system with architecture-specific improvements.

2.2. Specific Objectives

- Specific Objective 1
- Specific Objective 2

2.3. Contributions to the subject

This work will provide documentation regarding the details of the specific tradeoffs for the implementation approaches of multimedia software on the target platforms, and will serve as basics for future work regarding improving performance (power consumption).

2.4. Scope and Limitations

We will provide a reference implementation working on at least two different platforms. The reference implementation may lack features compared with existing solutions, but focused on providing a more maintainable design and performance equal or better.

CHAPTER

THREE

METHODOLOGY AND SCHEDULE

3.1. Methodology

Describe in technical language your research perspective and your past, present, or possible future points of view. List three research methodologies you could use, and describe why each might be appropriate and feasible. Select the most viable method.

3.2. Assumptions

Describe untested and un-testable positions, basic values, world views, or beliefs that are assumed in your study. Your examination should extend to your methodological assumptions, such as the attitude you have toward different analytic approaches and datat-gathering methods. Make the reader aware of your own biases.

3.3. Procedure

Describe in detail all the steps you will carry out to choose subjects, construct variables, develop hypotheses, gather and present data, such that another researcher

could replicate your work. Remember the presentation of data never speaks for itself, it must be interpreted.

3.4. List of Deliveries

1. Reference implementation of the overall general architecture implementing at least one audio and video format on two different hardware architectures.

3.5. Schedule and Work Breakdown

BIBLIOGRAPHY

[1] Gilles Chanteperdrix and Richard Cochran. The ARM Fast Context Switch Extension. In *Proceedings from the Real Time Linux Workshop*, 2009.

[2] Peter Clarke. Embedded processor market to resist financial crisis, say VDC. *EE Times*, 2009. URL http://www.eetimes.com/electronics-news/4194731/Embedded-processor-market-to-resist-financial-crisis-say-VDC.

[3] Jochen Liedtke. On μ-kernel construction. In *15th ACM symposium on Operating Systems Principles*, 1995.

[4] Brian Proffitt. Linux ARM support: A hot mess, an ugly clean-up. *ITworld*, 2011. URL http://www.itworld.com/mobile-wireless/175829/arm-and-linux-major-construction-ahead.

[5] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts Essentials*. Wiley Publishing, 2010. ISBN 04701287209.

[6] Deepak Somaya and Greg Linden. System-on-a-Chip Integration in the Semiconductor Industry: Industry Structure and Firm Strategies. *SSRN eLibrary*, 2000. doi: 10.2139/ssrn.259878.

ACRONYMS

DSP Digital Signal Processor is a special type of micro processor.

EE Electronic Engineer

GPP General Purpose Processor is a processor suited for standard computational operations, in contrast of a DSP for example.

OEM Original Equipment Manufacturing refers to a company or a firm that is responsible for designing and building a product according to its own specifications, and then selling the product to another company or firm, which is responsible for its distribution. The one company produces products on behalf of another company, after which the purchasing company markets the product under its own brand name. TODO (taken from)

ODM Original Design Manufacturing is a company or firm that is responsible for designing and building a product as per another company's specifications. TODO (taken from)

OS Operating System

PC Personal Computer

RTOS Real Time Operating System is an operating system that is designed to meet with Real Time deadlines.

SoC System on a Chip is a technology that consist on integrating several functional blocks of re-usable electronics logic (even from different vendors) into single die.