Operating Systems Portability: 8 bits and beyond *

Hugo Marcondes, Arliones Stevert Hoeller Junior,
Lucas Francisco Wanner and Antônio Augusto M. Fröhlich
Federal University of Santa Catarina
Laboratory for Software and Hardware Integration
UFSC/CTC/LISHA PO Box 476, 88049-900, Florianópolis, SC, Brazil
{hugom,arliones,lucas,guto}@lisha.ufsc.br

Abstract

Embedded software often needs to be ported from one system to another. This may happen for a number of reasons among which are the need for using less expensive hardware or the need for extra resources. Application portability can be achieved through an architecture-independent software/hardware interface. This is not a straight-forward task in the realm of embedded systems, since they often have very specific platforms. This work shows how an application-oriented component-based operating system was developed to allow system and application portability. Case studies present two embedded applications running in different platforms, showing that application source code is totally free of architecture-dependencies.

1. Introduction

When developing for embedded systems, application programmers often need to migrate their software from one system to another. This may happen for a number of reasons, among which are the need for using less expensive hardware, the need for extra resources such as more memory or specific components which are only present in different platforms, etc. However, modifying the applications to make it run in the new platform may become impracticable, for the recurrent engineering cost of the software re-design may overcome the benefits brought by the hardware modifications.

In order to guarantee application portability an architecture-independent software/hardware application programming interface must be defined. However, in the realm of embedded systems, this is not a straight forward task, since embedded systems hardware often have very specific characteristics.

Several strategies have been adopted to allow application portability, thus reducing recurrent engineering costs. An example is the use of system call interfaces (e.g. POSIX, WIN32, MOSI) [1]. Their use allow applications to run in any operating systems that implements them. However, most of these interfaces were defined focusing general purpose computing, making their use impracticable in deeply embedded systems.

Virtual Machines (VM) and Hardware Abstraction Layers (HAL) are two other strategies used to obtain portability for operating systems and, consequently, applications [1]. Although virtual machines offer a good level of portability, this strategy often incurs in excessive processing and memory overheads, restricting their use in embedded systems. A HAL is another option to obtain portability in operating systems and, indeed, is the most used strategy (e.g. ECOS, LINUX, WINDOWS). However, HAL implementations usually aggregate architectural characteristics which are common to a certain set of devices, restricting their portability only to similar hardware. This is not desired when dealing with embedded systems since hardware devices in such systems varies considerably.

This paper shows how an application-oriented, component-based operating system was developed to allow system and application portability. This was done through a careful domain engineering process allied to the use of advanced programming techniques such as aspect-oriented programming and static meta-programming in an object-oriented environment [2]. This paper also shows how the use of *hardware mediators* constructs [3] allowed the same system to run in very distinct architectures, ranging from 8-bits and beyond (e.g. H8, AVR, ARM, POWERPC, SPARCV8, IA32).

Case studies present two embedded applications running in different platforms. These show that application source code is totally free of architecture-dependent routines, thus porting between platforms are easier. The first case study is the implementation of an access control system using an AVR, 8-bits microcontroller (ATMEGA16). The second case study shows the same H.222 MPEG multiplexer application running on a POWERPC, 32-bits (IBM POWERPC 405) and an embedded IA32, 32-bits (AMD GEODE) platforms.

Section 2 presents the main architectural variations identified in the embedded systems domain along with an

*This work was partially supported by FINEP - Financiadora de Estudos e Projetos

analysis of the most used portability techniques for operating systems. Section 3 introduces the approach of this paper to enable portability on embedded systems. Section 4 describes the case studies. Finally, section 5 presents an overview of the paper and discusses the results.

2. Embedded Systems Portability

Embedded systems make use of a wide range of hardware architectures, ranging from simple 8-bit microcontrollers to sophisticated 32, 64 and 128-bits processors. The choice of a specific architecture depends on the requirements of the target application, and usually targets the reduction of production costs.

Deeply embedded applications usually do not require complex memory protection mechanisms, and may be built upon architectures without a Memory Managing Unit (MMU) hardware. Many microcontrollers are based on the Harvard architecture, with separate instruction and data buses, in contrast to the von Neumann architecture used in most general purpose microprocessors. Embedded microcontrollers may be based on either RISC or CISC architectures, trading off pipeline efficiency by code density [4].

Software compilation is also highly influenced by target architecture. When a microcontroller provides a large number of registers, these may be used for parameter-passing and function return values. Some architectures make use of complex structures such as the SPARC architecture register windows. Stack behavior is also determined by the architecture. POWERPC, for instance, make use of a dedicated register to allow the return address of a function to be pushed to the stack only when a new function is called.

Recent advances in Programmable Logic Devices (PLDS), have made FPGA (*Field Programmable Gate Array*) devices a viable alternative for embedded systems design. Although Hardware/Software Co-Design are still taking their first steps, it allow the software and hardware architecture design for embedded systems to be defined according to application requirements. Hardware elements may be recombined (through configurable IP components) and form architectures that are specialized for a given application.

Allowing efficient application portability in this scenario through common software/hardware interfaces is non-trivial, and requires efficient design and programming techniques. The usage of standard system-call interfaces (e.g. POSIX, WIN32) has allowed portability between operating systems and hardware architectures. However, for a standard to be effectively utilized, it must be based on well established and widely used technologies. This may be the main reason why some standard interfaces for embedded systems (e.g. MOSI - Microcontroller Operating System Interface) have failed to be widely accepted by the embedded systems industry.

The industry business model may also have contributed for the non-acceptance of standard system interfaces in embedded systems. In the general-purpose industry, hardware and software constitute separate, individual products. Embedded systems are usually single hardware/software products, which are in turn usually part of a larger system. Application portability through different operating systems thus takes a less visible role in embedded systems, as it is defined by the producer, and not the consumer.

Hardware Abstraction Layers (HALS) and Virtual Machines (VMS) also define means to attain application portability. Virtual Machines allow application portability through the definition of a hypothetical hardware architecture. Applications are designed for this architecture and then translated to the actual executing environment. Virtual Machines allow binary portability of applications allowing programs to execute in several platforms without any modification. However, most embedded systems cannot afford the overhead of a virtual machine. Binary portability is not always a requisite of embedded applications, and the costs associated with the computational resources necessary to support a virtual machine such as the JAVA VM would make most embedded systems economically unfeasible.

The usage of a Hardware Abstraction Layer (HAL) to abstract the underlying hardware platform seems to be the best option for attaining portability in an operating system, and it is in fact the most commonly used technique among operating systems (e.g. LINUX, WINDOWS, ECOS). However, HALS designed for these systems usually incorporate peculiarities from the original architecture for which they were conceived, allowing them to be ported only to similar architectures.

3. Designing Portable Software/Hardware Interface

There are several techniques which may be used to attain a good level of portability in a software/hardware interface. This section will present some of these techniques, including programming and design paradigms used to develop the interface proposed by this work.

Actual HALS for embedded systems do not satisfactorily explore the architectural variability presented by such systems. This occurs because most approaches do not go through a deep domain analysis to define their interfaces. The use of a careful domain engineering is essential to achieve the level of portability demanded by embedded systems. The domain engineering process consists of systematic development of a domain model and its implementation. A domain model is the representation of common and variant aspects of a number of representative systems of a domain and the rationale for variations [5].

Within this context, a lot of variabilities and commonalities are identified. Scheduling policies, synchronization handling (mutex, semaphore, condition variables), timing, memory management mechanisms (paging, segmentation, both, none), interrupt handling, I/O handling (memory

mapped, I/O ports, DMA, PIO) are examples of embedded systems variations. In order to sustain portability it is desirable that such characteristics are configurable in the operating system, thus allowing themselves adaptation to hardware and applications.

In order to produce an operating system with such characteristics, a domain analysis process was used, guided by the *Application-Oriented System Design* methodology (AOSD) [2]. This methodology proposes the use of several design and programming techniques to attain a high level of system configuration with minimum processing and memory overheads. These techniques include *Object-Oriented Design* [6], *Family-Based Design* [7] and *Aspect-Oriented Programming* [8].

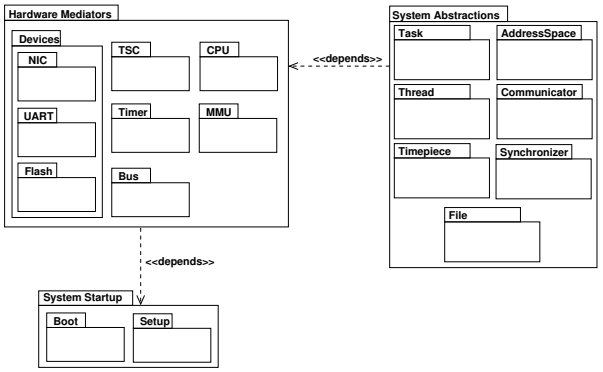

Figure 1. Domain Model

Figure 1 presents the organization of the component families in EPOS, an experimental application-oriented embedded operating system. Every architecture-dependent hardware unit was abstracted as a *hardware mediator* [3]. These constructs are responsible for exporting, through their interfaces, all the functionality needed by higher level abstractions. These abstractions are responsible for implementing traditional operating systems services such as memory management, process management, inter-process communication, etc.

3.1. Process Management

Processes are managed by the *Thread* and *Task abstractions*. *Task* abstractions corresponds to the activities specified in the program, while *Threads* are the entities that perform such activities. The main requisites and dependencies of these system abstractions are deeply related to central processor unit (CPU). For example, the execution context is defined by the registers presented inside the CPU, and stack manipulation is determined by the application binary interface standard of the CPU architecture.

Most architecture dependencies in process management are handled by the CPU *hardware mediator* (fig. 2). The CPU'S inner class *Context* defines all data that must be stored for an execution flow, and this way, each architecture defines their own *Context*. The *Context* object is always stored on each *Thread's stack*, so the memory address of *Thread Context* is always changing. The responsibility for maintaining the *Thread Context* pointer consistent is passed to the *CPU::switch_context* operation that receives a volatile pointer to the *Thread::_context* attribute (that is also a pointer to *Context* object). When the *Context* is saved on a new place on stack, this pointer is updated and the scheduled *Thread Context* is loaded in the CPU.

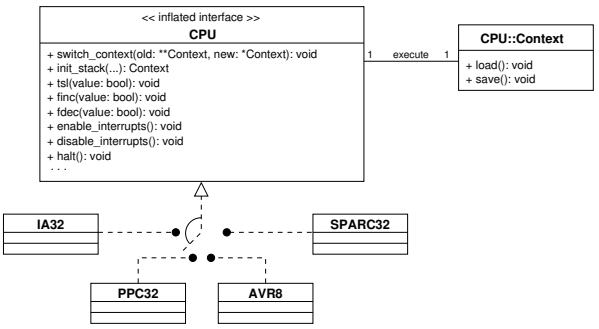

Figure 2. CPU Hardware Mediator

Another architecture dependency in process management is related to the stack initialization. The use of *meta-programed C++ templates* was essential to enable the creation of threads with flexible entry points without overhead. *Thread* constructors receives a pointer to a function with an arbitrary number of arguments of any type that is resolved by the meta-program in compile-time. Since compilers for different architectures handle function argument passing in different ways and *Thread* entry points work as if called like a normal function, the stack initialization on *Thread* creation must be done on CPU mediator through the *CPU::init_stack* method, ensuring that the function entry point arguments are placed according to each architecture application binary interface convention. Figure 3 illustrates a scenario where a *Thread* are created (steps 1, 2 and 3) and scheduled (steps 4 and 5). Notice that when the *preemptive* feature is enabled in system configuration, the step 5 occurs on *Thread* creation if it has a higher priority than the running *Thread*.

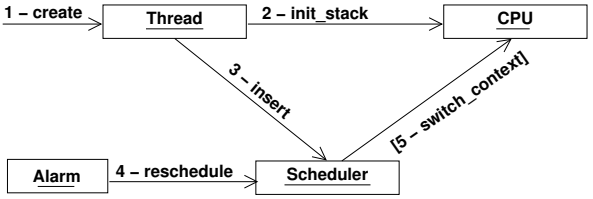

Figure 3. Thread's Creation and Scheduling

CPU *hardware mediators* also implement some functionality for other systems abstractions like bus-locked read-and-write transactions (*Test and Set Lock*) required by the *Synchronizer* family of abstractions and endianness conversion functions (e.g. Host to Network and CPU to Little Endian) used by *Communicators* and I/O Devices (PCI Devices). The process scheduling algorithm is handled by the *Timepiece* family of *abstractions*.

3.2. Timepiece

The notion of time is essential in any multi-thread or process system. Time in EPOS is handled by the *Timepiece* family of *abstractions*. *Timepiece abstractions* are supported through *hardware mediators* such as *Timers*, *Timestamp Counters* (TSC) and *Real-Time Clocks*.

The *Clock abstraction* is responsible for keeping track of current time and is available only on systems that feature a real-time clock device. The *Alarm abstraction* can be used to generate events that can wake-up a *Thread* or call a function. *Alarms* also have a master event with high priority that is associated with a certain period of time. This master event is used to call the process scheduling algorithm at each quantum of time, when the active scheduler feature is configured on the system. The *Chronometer abstraction* is used to perform time measurements, helpful in system performance benchmarks and time statistics.

Timer hardware in embedded systems may present a large number of functions and may be configured in many distinct ways. A timer can serve as a Pulse Width Modulator controlling a analog circuitry, Watchdog Timer, Programmable Interval Timer or as a Fixed Interval Timer. Each one of these have their own configuration peculiarities. To preserve an interface contract with the *Alarm* abstraction, the *Timer hardware mediator* presents a simplified view of the hardware which is seen as a periodic interrupt generator. The programmer can simply enable or disable the occurrence of interrupt and set its frequency. Other high-level abstractions might be defined to satisfy other usages of timers (e.g Pulse-Width Modulator abstraction), creating other specific purpose contract interfaces.

Often, deeply embedded architectures do not provides a timestamp counter inside the CPU. When this happens, the architecture Timestamp Counter mediator (TSC) must count the time using the hardware timer presented on the system. Generally these provide only low-precision measurements, but don't invalidate the use of *Chronometer abstraction*.

3.3. Synchronizers

Processes and Threads usually cooperate and share resources in the execution of the application. This cooperation is done using inter-process communications mechanisms or through shared data. Concurrent access to shared data or resources may result in data inconsistency. *Synchronizers* (fig 4) are mechanisms to ensure data consistency in a concurrent process environment.

The *Mutex* member implements a simple mutual exclusion mechanism that supplies two atomic operation: *lock* and *unlock*. The *Semaphore* member realizes a semaphore variable, that is a integer variable whose value can only be manipulated indirectly through the atomic operations *p* and *v*. The *Condition* member realizes a system abstraction inspired on the condition variable language concept, which allows a thread to wait for a predicate on shared data to become true.

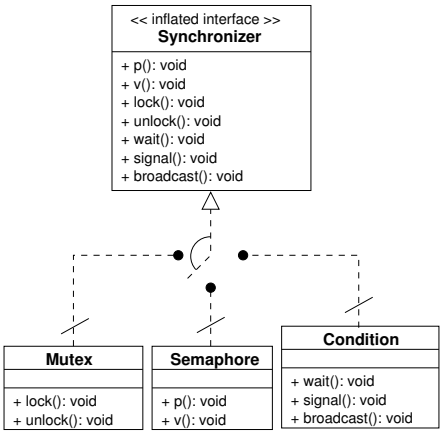

Figure 4. Synchronizer Abstractions

In order to implement these mechanisms, the hardware must provide ways to access data through atomic operations. Some architectures provide specific instruction for these operations (e.g. *tsl* instruction on IA32). Architectures that don't provide direct hardware support for these atomic operations must emulate the atomicity by disabling the occurrence of interrupts on CPU. As seen before these atomic operations are implemented in the CPU hardware mediator.

3.4. Memory Management

For most operating systems the presence of a memory management unit (MMU) is an unbreakable barrier that forces them to either be portable across platforms that feature an specific kind of MMU (e.g. Paging) or portable across platforms without memory management hardware. However, careful design of abstractions and hardware mediators may enable the memory management components to be ported across virtually any platform.

The encapsulation in the MMU family of hardware mediators of details pertaining to address space protection and translation, as well as memory allocation, was essential for achieving the high degree of portability in EPOS. The *Address_Space* abstraction is a container for chunks of physical memory called segments. It does not implement any protection, translation or allocation duties, handing then over to the MMU mediator. This design is depicted in figure 5, which additionally illustrates the message flow for a segment creation (1 and 2) and attachment (3 and 4).

The *Flat* Address Space (fig 5) defines a memory model in which logical and physical addresses match, thus eliminating the need for a MMU hardware. This ensures the preservation of the interface contract between other components and the memory subsystem in platforms that do not feature a MMU. The MMU mediator for a platform that does not feature the corresponding hardware components is a rather simple artifact. Configuration rules guarantee that this simple artifact can not be used without a *Flat* Address Space model. Methods concerning the

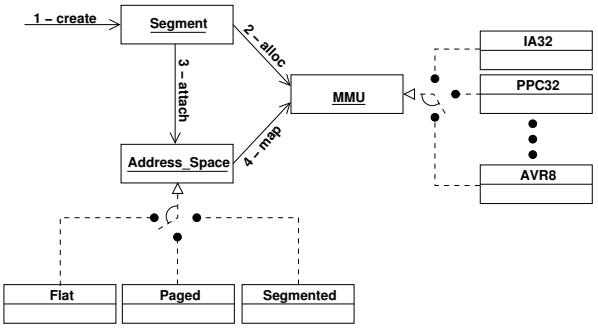

Figure 5. Memory Management

attachment of memory segments into the single flat address space become empty, with segments being attached at their physical address. Methods concerning memory allocation operate on bytes in a way that is similar to libc's traditional malloc function.

Conceptually the memory model defined by the *Flat* Address Space may be viewed as a degeneration of the paged memory model in which the page size equals to one byte and the page tables implicitly map physical addresses as logical ones.

3.5. Input/Output Devices

Controlling computer system I/O (input/output) devices is one of the main functions of an operating system. Deeply embedded system does not usually provides traditional I/O interfaces (e.g. keyboards, display, mouse) they usually interface with the environment through sensor and actuators [9]. These devices present a huge variability and the way they are access may be different according to the architecture.

An additional phenomenon which is typical of low-level programming regards the mediation of the same hardware device in different architectures. For example, suppose that a given device is part of two hardware platforms, one that uses programed I/O and another that uses memory mapped I/O. Being the same device, it is very likely that the procedures for access, configuring and interacting with the hardware will be the same in both platforms, thus turning the corresponding device driver a portable component. Nevertheless, the difference regarding I/O access mode will lead traditional operating systems to set-up two distinct, non-portable drivers. A meta-programmed hardware mediator can solve this kind of problem by introducing an *IO_Register* component that solves the I/O access mode at compile-time (using template specialization and operator overloading of C++). Memory-mapped devices may also bring another problem, PCI based devices treats words with little-endian byte ordering, thus a device driver for this hardware must concern the CPU byte-ordering in its implementation. This is handled in CPU hardware mediator by methods responsible for changing byte-ordering if necessary.

Interrupt Handling is another major concern in I/O management,since it avoids CPU polling of hardware registers. Interrupt management is done by *Machine hardware mediator* with the Interrupt Controller (*IC*) hardware mediator. The *IC* hardware mediator abstracts the process of enabling and disabling the occurrence of interrupts, while the interrupt vector table is management by the *Machine hardware mediator*.

The interrupt vector table can be handled by many different ways, accordingly to the architecture. POWERPC for instance, implement two levels of interrupt vector table, one for internal CPU exceptions with distinct offsets for each exception. Another vector table may be defined for external interrupts generated by external hardware. *Avr* based microcontrollers usually implement an interrupt handler table with pre-defined size and location. *Epos* system handles the interrupt vector table in an uniform way. This is achieved through a special boot-up utility called *Setup*. The *Setup* is a non-portable tool that executes before the operating system to build an elementary execution context for it. This way, one the Setup's responsibilities is to build complementary interrupt vector structures when needed to support the uniform treatment of interrupts on *Machine* and *IC* hardware mediators.

4. Case Studies

This section presents two case studies using proposed the software/hardware interface: An access control system and a H.222 MPEG multiplexer.

These applications were developed by using the inflated interfaces of EPOS components. The resulting application code was submitted to a tool that performs syntactical and data flow analysis to extract a blueprint for the operating system to be generated. When the system is implemented in a programmable logic device, the high-level description for the hardware platform to be synthesized is also generated by this tool. This blueprint is then refined by the application programmer through a dependency analysis process guided by information about the execution scenario acquired from the user through visual tools. This process generates a series of selection keys that will drive the compilation of the operating system and the synthesis of hardware components into a programmable logic device, if required.

4.1. Access Control System

This case study consists of an access control system (fig. 6) developed using contactless smart cards. The embedded system reads the data from a smart card reader and performs a search over a database implemented in an internal EEPROM. If the data is found in the database, the system liberates a locking system through a GPIO port (e.g. GPIO switch on a triac connected on a electromechanical lock).

The contactless smart card reader performs the reading and sends the data through a serial interface. We used an AVR ATMEGA16 microcontroller to perform the system management. The ATMEGA16 has an internal AVR

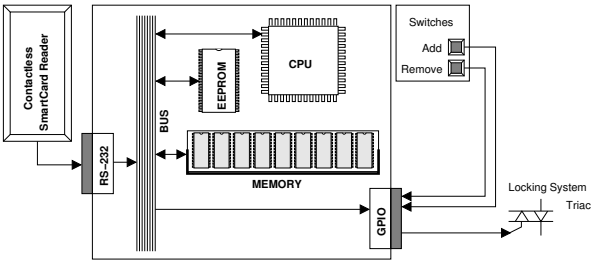

Figure 6. Access Control System

8-bits processor with a 16-bit address space range. It has 16Kb of program memory and 1Kb of RAM memory. The database was implemented inside an EEPROM memory that holds 512 bytes. The ATMEGA16 includes a set of devices, such as programmable timers, serial controllers, ADC converters and I/O interfaces for external devices interaction (GPIO).

The system abstractions used by this case study were the *Serial Communicator*, for the reception of the data read from smart card reader, *Storage*, to build the database on EEPROM memory, and GPIO, used interact with external systems and users. A *Thread* implements the the main application execution flow. A second *Thread* handles the process for insertion and deletion of new data in EEPROM database. Since both threads share the EEPROM resource, synchronizers are used to ensure data integrity.

The application was compiled and linked with EPOS operating system using GCC 4.0.2 for AVR. The resulting object code used 241 bytes of data memory (.data and .bss segments) and 13.484 bytes of code memory (.text segment).

4.2. H.222 MPEG Multiplexer

The H.222 MPEG multiplexer (fig. 7) was developed in the context of the Brazilian Digital Television System (SBTVD) and consists of an embedded system responsible for assembling a MPEG-2 transport stream, receiving the elementary streams of audio, video and data. The transport stream is sent to the modulation system in order to be transmitted to the user's reception unit.

Since audio and video de-synchronization and delay in this multimedia application are not acceptable, the system has real-time requirements. The use of high definition image brings a high data flow in the system, making the use of simple microcontrollers impossible.

We used a ML310 development board from Xilinx which contains a VirtexII-Pro XC2V30 FPGA (*Field Programmable Gate Array*) to implement the system. This FPGA has two IBM POWERPC 405 32-bits processor *hardcores*, but only one was used for this project. All others functionalities needed by the system were synthesized on the FPGA, following the system requirements. An UART was used to debug the system, and a PCI Bridge and an Interrupt Controller was instantiated on FPGA to connect the processor to the external I/O devices (e.g. networks cards), responsible for receiving and sending the MPEG data.

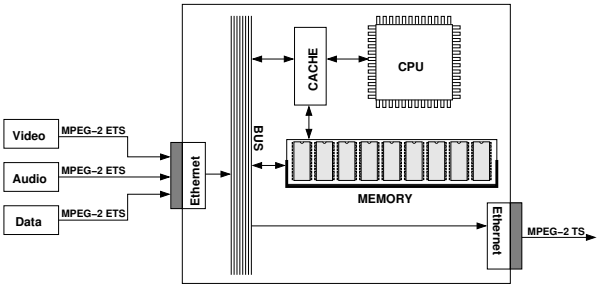

Figure 7. H.222 MPEG Multiplexer Platform

The application uses an arbitrary number of threads to handle elementary stream reception (ES). These threads executes in order to avoid hardware reception buffer overflows. Two control threads provide stream timing information (T) and packet synchronization (S). The multiplexer thread (MUX) gathers data from the ES, T and S threads and outputs a transport stream through an output (OUT) thread. In order to guarantee the consistency of shared data, Synchronizers are used by all threads.

The MUX application was also developed over a industrial PC Geode GX1 platform with a SC2100 Geode processor. The main reason for this was the execution time restriction of SBTVD project. Since this platform is based on the IA32 architecture, a well established port of EPOS system, the developers could test the application on a ordinary PC computer while the EPOS was ported to ML310 platform.

This shows the portability of application through the design proposed. The application code is running on both systems, without any modifications, and although the two systems are 32-bit architectures, both have many differences between them. The IA32 is a little endian machine with a restricted set of registers and a MMU hardware imposed. PowerPC architecture presents a big endian machine with 32 registers (some used for passing functions arguments), and the MMU hardware was disabled since the proposed application doesn't require a multi-task environment.

The application was compiled and linked with EPOS operating system using GCC 4.0.2 for POWERPC. The resulting object code used 1.055.708 bytes of data memory (.data and .bss segments) and 66.820 bytes of code memory (.text segment).

5. Conclusion

This work presented the design of a highly portable operating system that may be executed on a variety of hardware architectures, from simple microcontrollers to sophisticated processors, accordingly to application requirements. This was achieved by a careful domain engineering process that yield a component and family based design and modern programming techniques such as static metaprogramming and aspect-oriented programming. These

techniques enable the design of a contract interface between portable system abstractions and hardware mediators in such way that when the hardware does not feature an essential requisite, this is implemented by software in the hardware mediator artifact.

The use of an unique software/hardware interface facilitates the development process. Through system abstractions reusability it is possible to decrease time-to-market and minimize recurrent engineering costs when applications must be ported to another hardware architecture.

References

[1] J. D. Mooney, "Strategies for Supporting Application Portability.", *IEEE Computer*, vol. 23, no. 11, pp. 59–70, 1990.

[2] A. A. M. Fröhlich, *Application-Oriented Operating Systems*, GMD - Forschungszentrum Informationstechnik, 1 edition, 2001.

[3] F. V. Polpeta and A. A. Fröhlich, "Hardware Mediators: A Portability Artifact for Component-Based Systems.", in L. T. Yang, M. Guo, G. R. Gao, and N. K. Jha, editors, *EUC*, volume 3207 of *Lecture Notes in Computer Science*, 2004, pp. 271–280. Springer.

[4] D. Bhandarkar and D. W. Clark, "Performance From Architecture: Comparing a RISC and CISC with Similar Hardware Organization.", in *ASPLOS*, 1991, pp. 310–319.

[5] K. Czarnecki, "Beyond Objects: Generative Programming", 1997.

[6] G. Booch, *Object-Oriented Analysis and Design with Applications*, Addison-Wesley, 2 edition, 1994.

[7] D. L. Parnas, "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, pp. 1–9, Mar. 1976.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming", in *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, June 1997, pp. 220–242, Jyväskylä, Finland. Springer.

[9] P. Marwedel, *Embedded System Design*, Kluwer Academic Publishers, 2003.