Operating Systems on SoCs: A good idea?

Frank Engel Ihor Kuz Stefan M. Petters Sergio Ruocco
National ICT Australia Ltd.
Sydney
Australia
{firstname.lastname}@nicta.com.au

Abstract

System on a chip (SoC) designs penetrate in an increasing rate the embedded systems market. This affects the consumer market as well as other areas, like automotive or aerospace. As the computational power of CPU cores on such SoC increases, the use of operating systems (OS) as basis for the application becomes more and more an issue. This paper investigates the specific requirements of SoCs and analyses to what degree microkernel based OSes have any advantage over standard embedded real-time operating systems. It concludes that temporal and functional verification of OSes are of fundamental importance and that so far no embedded OS fulfills these criteria. However, it seems microkernels seem to support this better that other embedded OSes.

1 Introduction

Embedded systems have been widely deployed for many decades now. While older embedded systems incorporated processors memory and other peripherals all assembled on a PCB board, current progress in silicon technology and EDA tools has made it possible to implement such systems on a single die. These systems on a chip (SoC) allow functional modification to what were previously fixed components, thus enabling the tailoring of systems to application requirements (e.g. data throughput, power consumption, hardware functionality etc.). Typical application domains were SoC plays an important role are automotive (e.g. engine control), communication (mobile phones, modems, etc.), and multimedia (e.g. cameras, set-top boxes).

The implementation possibilities provided by SoC for specific applications are quite extensive. Thus, for example, an SoC implementation of a digital video camera on a single die (camera-on-a-chip) may include CMOS sensors, signal processing for audio/video, control processing for lens adjustment, etc. Designing the signal processing as custom logic hardware leads to a fast and small implementation [1]. Furthermore, using an embedded digital signal processor (DSP) allows algorithm updates, while introducing image manipulation specific instructions to this DSP improves its computation abilities [2], [3]. Deciding on which components to implement in hardware and which in software is the domain of *hardware/software co-design* [4].

An SoC typically consists of CPU cores (e.g. based on the ARM architecture [5]), or scalable processor architectures [6], combined with on-chip bus standards such as AMBA [5] and IP modules (including peripherals like USB, RS232). The design of an SoC is usually a network of modules, where each module provides a specific functionality and the connections between these modules consist of communication links typically provided by a bus or network. FPGAs capable of carrying whole SoC designs (called System on Programmable Chip (SoPC) solutions) now include DSP accelerators and soft or hard core CPUs (e.g. Altera [7]). SoPC makes manufacturing systems in small quantities affordable, while reconfiguration in hardware and in software further improve upgradability.

An operating systems that wishes to take (and provide applications with) full advantage of the multifaceted and modular nature of SoC must fulfill a number of specific requirements. First, the operating system should be modular, reflecting the modular nature

of the hardware. Also, knowledge of whether particular operating system functionality is implemented in hardware or software should be abstracted from the application. Furthermore, because the SoC design partitioning might change during the development process it must be possible to adapt the operating system to reflect these changes without redefining the abstraction presented to the application layer. This is important since system and application design are concurrent processes.

Next, the OS must not prevent an application from exploiting application specific functionality by defining too general an abstraction. An overly general abstraction may negate much of the benefit brought by application specific hardware in the first place. The OS must also provide protection and concurrency control, meaning that different parts of the OS and application (e.g., interrupt handlers, drivers, application code, etc.) cannot (expect to) interfere with each other.

Due to the application specific nature of SoC, it will be necessary to port the OS to many different platforms. The OS and application code should, therefore, be highly portable. This in turn requires the different functions of the OS to be well encapsulated, to avoid cross dependencies. Also, because most SoC systems have real-time and dependability requirements [8] verifiable correctness and temporal analysis of the OS are desirable.

The position taken in this paper is that a microkernel forms the ideal base for an operating system targeted at (reconfigurable) SoC. In the rest of this paper we will examine the above requirements in more detail and show how they are fulfilled by a microkernel-based operating system. We will also discuss other approaches to the design of operating systems for SoC and compare these to the microkernel-based approach.

2 Operating Systems for SoC

An operating system provides application software with an abstracted view of the hardware. As part of the abstraction it provides hardware initialisation and housekeeping, task management (including scheduling of tasks and jobs), general resource management (memory, peripherals, etc.), interprocess communication, and concurrency control (locks, semaphores, etc.).

Due to their application specific nature, not all SoCs [have to/need to/ necessarily] run an operating system. Instead the application software implements required OS functionality directly. The reasons for not running an OS range from performance concerns, to lack of an OS for the given SoC platform.

The arguments for running an OS on a SoC, on the other hand, are similar to those for running an OS on any system. Primarily, using an existing OS reduces the amount of time spent reimplementing OS functionality. Furthermore, using an OS with a well designed and documented API allows programmers to make efficient use of their time and skills. An OS also provides abstractions that make using the hardware easier and less error prone for the programmer. This abstraction also allows application code to be reused even if the details of the underlying hardware architecture change. Moreover, using a previously implemented and deployed OS reduces the number of bugs in the software. Finally, because there is a clear separation between the OS-like functionality (i.e. hardware access, task management, etc) and the application code, there is a cleaner separation of concerns, which leads to better structured and potentially less buggy code.

2.1 Modularity

The ability to implement traditional software components directly in hardware opens up new possibilities for SoC operating systems. In particular it becomes possible to implement OS functionality directly in hardware. For example, a TCP/IP stack may be implemented in hardware to improve performance and reliability, and reduce power consumption [9].

In order to support hardware implemented operating system functionality, the operating system must have a modular structure that allows parts to be added, removed, replaced, or modified. For example, if the TCP/IP stack is to be moved into hardware, it must be possible to remove the TCP/IP code from the operating system and replace it with an interface to the hardware implementation. Likewise if the SoC is originally built with a hardware TCP/IP stack but this is removed to make place for a different component, it must be possible to add a software implementation of the stack to the OS. In operating systems without a modular structure interdepen-

dencies between different parts of the OS make such modifications difficult and error prone.

Modularity is also required to achieve scalability of functionality. A scalable operating system is one that can be configured to include only the functionality that is required by the applications using it. Since SoCs used in embedded systems are typically resource constrained, scalability is a particularly important characteristic of SoC operating systems. Related to the scalability requirement is the requirement that the OS provide a lightweight interface (API) that provides only the functionality required by an application and no more. Heavyweight interfaces introduce unnecessary overhead, which negatively affects the overall system size and performance. For example, in [10] Colin and Puaut found that many of the parameters of the POSIX interface where not used and hence led to a considerable overestimation of the worst case execution time (WCET) estimate. Furthermore, heavyweight APIs detract from modularity by defining a large amount of core functionality that the OS must always provide.

2.2 Minimal Abstraction

As mentioned above, an operating system's main task is to provide an abstract view of the hardware that it runs on. In a traditional operating system the act of abstraction from the underlying hardware has three main objectives with respect to access to resources: protection and security, multiplexing and simplifying their usage. Virtual memory and multitasking are the chiefs examples of the first two objective. Device drivers are chief examples of the latter.

For SoC these three objectives are not necessarily desiderable. The software running on a SoC leverages application specific hardware present in the chip. Thus it should not be impeded the access in principle in the name of protection, or granted arbitrary access as privileged, in kernel code, but offered a finer, possibily hierarchical, tailoring of protection policies, instead of the binary choice kernel-space/user-space offered by standard operating systems.

In the end, the type of abstraction performed by the supporting software of an SoC should supports the hardware/software codesign of the system. This means that the operating system cannot abstract all the hardware, instead it should abstract only the generic (non application-specific) hardware.

In the case of a processor the OS reflects that the processor supports address space management, concurrent activities (e.g. programs running in parallel), information exchange between concurrent activities (inter process communication, IPC) and the management to serialize this virtual concurrency (scheduling). Since these characteristics are independent of the actual processor hardware architecture, such an OS abstracts particular hardware realizations. For example it abstracts how many general purpose registers a particular processor architecture provides, how a switch into supervisor mode is executed, etc. This is reflected by the amount of time or clock cycles needed to actually perform a certain duty.

2.3 Protection

SoC systems, especially those found in mobile phones and PDAs, are increasingly able to run downloaded, end-user code. When running such code it is important to prevent malicious or buggy code from overriding restrictions or modifying system properties. A user application that, for example, overrides a phone's lock to a particular network, or reprograms the radio to use different frequencies may have serious effects on revenue and liability. The ability to provide protection is, therefore, a main objective in selecting an OS for these types of systems.

The protection requirements can be summarised as memory protection (preventing unauthorised access to memory), failure confinement (preventing failure in one part of the OS from affecting other unrelated parts) and IP protection (). Note that, because failure confinement requires an understanding of the whole system's semantics, it can only be addressed to a limited degree within the operating system. Other aspects (such as,) may have to be addressed outside the OS, at the application level.

2.4 Portability

Since SoCs are application specific, every SoC design will provide a different hardware environment that the operating system must manage. This means that the operating system will effectively have to be ported to a new hardware platfrom for every new SoC that it is run on. An OS suited for SoC applications should, therefore, be highly portable, providing encapsulation for the different functions of the OS. This en-

capsulation is necessary to avoid cross dependencies, which complicate the porting process.

Not only is it necessary for the code to be portable, but modifications to the operating system code must not cause the API or the abstractions provided by the operating system to change. This is particularly important when changes is SoC design partitioning during the development cause related OS level changes. Such changes should not affect the application code unless they affect application functionality directly.

2.5 Verification and Analysability

Since many embedded systems have real-time and/or dependability requirements it is desirable that the OS can be shown to be correct and that it is possible to analyse it's temporal behaviour. While protection goes a long way towards fulfilling dependability requirements, only formal verification can provide certainty about the correctness and dependability of an application.

Verification of an application requires that the underlying OS and hardware can also be verified. In the safety critical application domains, OSes are trusted only on basis of code inspections. These inspections are underpinned by formal analysis of parts of the OS. While hardware verification has received wide attention [11], to our knowledge, no embedded operating system has yet been successfully verified.

Fulfilling an embedded application's real time requirements depends largely on knowing the timing properties of the underlying hardware and OS. In the area of real-time requirements, most systems will have soft real-time usability requirements. This ranges from end-user accepted delays between action initiation and the action being carried out in standard UIs, to image and sound quality of mpeg players or video cameras. While end-to-end measurements currently used for this kind of analysis provide a good idea of an operating system's average timing characteristics, only a complete WCET analysis can provide strong guarantees.

As with formal verification, to our knowledge no embedded operating system's timing characteristics have been succesfully analysed yet.

3 Embedded and Real-Time Operating Systems

Existing operating systems for embedded systems (and therefore SoC) can be divided into two categories. The first category, which we refer to as *embedded operating systems*, are traditional desktop and server operating systems that have been adapted to embedded systems. These include Unix and Windows derivates such as RT-Linux [12], RTAI [], and Windows CE.

The other category, which we refer to as *real-time operating systems (RTOS)* are operating systems that have been designed from scratch to be used in embedded systems. They typically have a small footprint, are scalable, and provide some support for real-time applications. There are hundreds of RTOS's on the market, with a wide range of functionalities [13]. Examples of widely used RTOS's include RTEMS [14], and uC/OSII [15]. Note that there are also a number of microkernel-based RTOS's (such as QNX Neutrino [16], VxWorks [17], etc.) which we will not discuss in this section.

3.1 Modularity

In the RTOS category of SoC operating systems, modularity is an important feature as it allows the OS to scale. Most RTOS's provide compile time scalability, allowing feature sets to be included or excluded as needed. The embedded operating systems, on the other hand, are generally less modular, and do not scale as well as the RTOS's. They also have a larger footprint and require more resources. Some hybrid systems combine a small RTOS with a large embedded OS. RT-Linux, for example, consists of an RTOS that runs Linux as its idle thread. Although the RTOS part is small, it does depend on Linux for functionality such as initialisation, memory management and drivers.

3.2 Minimal Abstraction

In terms of abstraction RTOS's provide a wide range of abstractions. The probably most common abstraction is the POSIX API. We argue that this is often too heavy as API for SoCs. As mentioned previously in this paper, it has been found, that the parameter

space provided is usually not balanced by requirements of an individual application.

Windows CE as embedded OS, for example, abstracts considerably away from the hardware. In RT-Linux the hardware may be directly accessed, due to the RT-threads running in kernel space. But this requires a very careful and robust application design since those applications could harm the whole system functionality. The alternative is to work through Linux device drivers, which once more have a very abstract interface. Thus the current minimal abstraction leaves it to the designer to choose between fast execution and reliability.

3.3 Protection

With regards to protection there are, once again, two categories of OSes: those that provide memory protection and those that do not. Some OSes like RTEMS provide only unprotected access to memory with the rationale that any software running on an RTOS is trusted. On the other hand, other OSes like favour a full protection model. The rationale behind this choice is that memory protection speeds up development, testing and bug tracking. There are also some OSes that offer both models, making the choice of protected or unprotected model a compile-time option.

3.4 Portability

Most commercial real-time and embedded OSes provide a wide range of hardware support. This hardware support is generally provided by the OS vendor and covers existing CPUs, common buses, etc. Only few commercial RTOS's (such as uC/OS-II) provide the means to adapt them to the very specific environment and capabilities of SoCs. Non-commercial OSes such as maRTE-OS [18] generally do not provide a wide range of hardware support, however, they are more accessible and therefore provide the means to take full advantage of the SoC environment.

3.5 Verification and Analysability

We are not aware of any successful or ongoing attempts to formally prove the functional correctness of any operating system discussed in this section. Likewise, we are not aware of any successful real-time analysis of these operating systems. With regards to real-time analysis, there have been attempts to analyse RTEMS and RT-Linux [10, 19], however, none of these performed a complete analysis, nor did they provided results that would be usable in an industrial setting. There have also been attempts to formally analyse an RTOS specification [20] as well as analyse individual parts of an OS, such as an analysis of the scheduler performed by Cofer and Rangarajah [21]

Linux-based embedded systems face additional challenges. Proving any temporal property is complicated by the fact that any device driver, just as any other component of the operating system, can arbitrarily enable or disable interrupts. While it may doing this without violating the functional correctness of the systems, it will definitely affect the scheduling of real-time processes. Thus, a minimal update to any device driver will require a complete analysis of the system.

4 Microkernels

In the late 1980's microkernels were born from the necessity of taming the complexity of operating systems development. In a traditional operating system different subsystems, such as device drivers, file systems, network stacks, etc. coexist in a single address space and protection domain. Having all its subsystems operate in the same protection domain like that makes the kernel brittle and vulnerable to bugs. While the effects of a buffer overflow or following a null pointer in an application remain confined within the application's address space, the same problem in a driver or other OS subsystem can crash the kernel and bring the entire system down.

Furthermore, sharing data structures (e.g. having global variables), introduces subtle interdependencies that muddle boundaries between subsystems and hinder evolution. Small modifications in one subsystem may have unexpected results for other subsystems that access its functionality. For example a new organization for the page table affects all device drivers that access it directly to map I/O buffers from and to user virtual address spaces.

The microkernel itself retains only the control of the privileged aspects of the CPU, that is memory protection and address space management, scheduling, and communication between threads. However,

while a traditional operating system tends to offer the richest and most complete set of APIs for this purpose, a microkernel strives to expose only the most basic and minimum set of *mechanisms* that allow the implementation of these functionalities without constraints.

In the following subsections we will illustrate how the microkernel approach to operating systems design fulfills the requirements for SoC discussed earlier.

4.1 Modularity

A properly designed microkernel-based system is intrinsically modular. Functionality is provided by server modules that are isolated from each other through memory protection. Clients make requests and obtain services through IPC. Servers can fulfill requests directly, or decompose them into lower-level requests and submit them to other servers, delegating the work to servers providing the most appropriate functionality.

With a microkernel-based design the modifications are confined in the various servers. Not only can the implementation details of a server be changed without affecting any other server, but it can also be substituted on the fly with another server without side effects, provided that the interface stays the same and a proper hand-over protocol is established and followed.

Because all communication is done through well defined interfaces the subtle interdependencies caused by data structures implicitly shared between different modules do not arise. Note that microkernel do allow memory to be shared between separate address spaces. This sharing is explicit, and is not prone to the problems caused when modules share memory implicitly. Notwithstanding this, microkernels are sometimes used to guest full-fledged, monolithic operating systems running as a single task. This is the case of Apple's MacOS X. Darwin is a NetBSD operating systems adapted to run on top of the Mach 2.5/3.0 microkernel as a single Mach task.

4.2 Minimal Abstraction

The key design principle of second-generation microkernels is that the microkernel does not contain anything that other parts of the system may want or need to modify as the system requirements change. In particular, the only abstractions of the hardware that a microkernel provides are a platform-independent interface to virtual address spaces, threads, scheduling, and IPC.

4.3 Protection

The microkernel approach to protection is to run OS subsystems as specialized servers executing in separate and protected address spaces. Each subsystem provides its services by running one or more threads that communicate with user applications and other subsystems, exchanging messages according to well defined protocols. While slower than manipulating directly shared data, this IPC-based interaction is considerably safer and more flexible. The server's key data structures are protected and the server is free to change its internals as long as it mantains compatibility with the published interface.

4.4 Portability

The portability of the microkernel ultimately depends on the architectural details of the platform it manipulates on behalf of its clients. Virtual memory page table management and context-switch code are definitely examples of processor-specific features of a microkernel and are inherently not portable. While these technical and performance issues mean that a microkernel itself may be not fully portable, as long as the microkernel interface is platform independent the rest of software built on top of it will be portable. Furthermore, because the microkernel only provides a minimal abstraction of the hardware, all other OS services will be fully portable as long as they rely only on the microkernel interface.

4.5 Verification and Analysability

The feasibility of formal verification of software depends chiefly on its size. A second generation microkernel like L4 [22] is composed by about 10,000 lines of C++ and assembly code. A first generation microkernel like Mach or a monolithic kernel like Linux are, respectively, roughly one and two orders of magnitude larger.

While comparably small, L4 is towards the upper bound of the typical size of programs being verified.

Still, a research project being undertaken at NICTA is tackling the verification of the L4 virtual memory subsystem and, eventually, the full kernel.

Similarly, the relatively small size of the microkernels, and the design style they support that induces a systematic, modular decomposition in the rest of the system, makes them more amenable to temporal analysis than a full fledged monolithic kernel.

5 Conclusion

In this paper we have investigated the suitability of different operating system classes for SoCs. We have provided five criteria to compare the available operating systems: Modularity, abstraction, protection, portability and analysability. The term analysability is currently a weak point for all types of operating systems. The monolithic real-time operating systems as a group provided support for almost all of the required properties, however, individual OS's showed a lack in individual points. This is often reflected in either a lack of protection or a non minimal abstraction. This comes quite natural, as these are subject to trade offs for each individual application. In terms of effectively facilitating the features of SoC hardware, when the RTOS is ported, the monolithic RTOS's are not doing as well as microkernels. Furthermore we see the advantage of the low abstraction of a microkernel, which supports an protection level to be introduced as needed between the application and the kernel.

Current and Future work within NICTA Ltd. addresses the issue of providing formal verification of functional properties and timing analysis of the L4 microkernel and thus is closing the gap remaining in fulfilling all properties described above.

References

[1] U. Ramacher, I. Koren, H. Geib, C. Heer, T. Kodytek, J. Werner, J. Dohndorf, J.-U. Schlüssler, J. Poidevin, and S. Kirmser, "Single-Chip Video Camera With Multiple Integrated Functions," in *Proc. of IEEE International Solid-State Circuits Conference (ISSCC)*, (San Francisco, CA), pp. 306–307, 1999.

[2] W. Caarls, P. P. Jonker, and H. Corporaal, "SmartCam: Devices for Embedded Intelligent Cameras," in *Proc. of 3rd PROGRESS Workshop on Embedded Systems* (M. Schweizer, ed.), October 2002.

[3] R. Kleinhorst, H. Broers, A. Abbo, H. Ebrahimmalek, H. Fatemi, H. Corporaal, and P. Jonker in *Proc. of ProRISC*, (Veldhove, The Netherlands), November 2003.

[4] R. Ernst, "Codesign of Embedded Systems: Status and Trends," *IEEE Design & Test of Computers*, vol. 15, pp. 45–54, April–June 1998.

[5] ARM Ltd. Homepage. www.arm.com.

[6] Tensilica Inc. Homepage. www.tensilica.com.

[7] Altera Inc. Homepage. www.altera.com.

[8] D. D. Gajski and F. Vahid, "Specification and Design of Embedded Hardware-Software Systems," *IEEE Design & Test of Computers*, vol. 12, Spring 1995.

[9] Ipsil Inc. FlowStack. Homepage. www.ipsil.com.

[10] A. Colin and I. Puaut, "Worst case execution time analysis of the RTEMS real–time operating system," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, (Delft, Netherlands), pp. 191–198, June 13–15 2001.

[11] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Transactions on Design Automation of E. Systems*, vol. 4, pp. 123–193, Apr. 1999.

[12] FSMLabs Inc. Homepage. www.fsmlabs.com.

[13] iApplianceWeb. Webpage. www.iapplianceweb.\ com/apptable/iaw_operating_systems.htm.

[14] RTEMS Project. Homepage. www.rtems.com.

[15] Micrium Inc. Homepage. www.micrium.com.

[16] QNX Software Systems Inc. Homepage. www.qnx.com/products/rtos/.

[17] Wind River Inc. Homepage. www.micrium.com.

[18] M. A. Rivas and M. G. Harbour, "Evaluation of new posix real-time operating systems services for small embedded platforms," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, (Porto, Portugal), July 2-4 2003.

[19] S. M. Petters, *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Institute for Real–Time Computer Systems, Technische Universität München, Munich, Germany, Sept. 2002.

[20] S. Fowler and A. Wellings, "Formal analysis of a real-time kernel specification," in *Proceedings of the Fourth International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, (Uppsala, Sweden), pp. 440–458, Springer–Verlag, 1996.

[21] D. C. A1 and M. Rangarajan, "Formal modeling and analysis of advanced scheduling features in an avionics rtos," in *Proceedings of the Second International Conference (EMSOFT 2002)*, Lecture Notes in Computer Science, (Grenoble, France), Springer–Verlag, Oct. 7-9 2002.

[22] disy L4. Homepage. www.disy.cse.unsw.edu.au/L4/.