# COS: A Configurable OS for Embedded SoC Systems

Hsin-hung Lin
Real-Time Systems Laboratory
Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi, Taiwan 621, R.O.C.
lsh@cs.ccu.edu.tw

Chih-Wen Hsueh
Embedded System and Wireless Networking Laboratory
Graduate Institute of Networking and Multimedia and
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan 106, R.O.C.
cwhsueh@csie.ntu.edu.tw

***Abstract*— As the increasing of system performance and computing power, embedded systems are more complicated and interactive. Therefore, operating system (OS) plays a more important role in embedded systems to utilize various hardware and software resources. However, no OS can meet all requirements of various embedded systems. Due to the advance of reconfigurable processors, system requirements can be more dynamic. Embedded OS has becoming more critical in the development of embedded systems and thus there is a strong need of configurable embedded OS to better and faster build up the target system. In this paper, we propose a configurable OS, called *COS*, based on SOA (Service-Oriented Architecture) for embedded SoC (System on a Chip) systems to build an application specific OS according to the system requirements. COS can be easily configured to better utilize the resources of the target embedded platform and have better support to the embedded application. Moreover, the COS can easily extend new features or functionalities of other OSes and even be adapted to the designs of other OSes. We implement COS on an ARM platform to prove its configurability and also evaluate its overhead. We believe that COS can be configured to meet the various requirements in embedded SoC systems and help to speed up the embedded system development process.**

**Keyword: Configurable OS, Service-Oriented Architecture, SoC, Embedded OS.**

## I. Introduction

In recent years, embedded systems have been rapidly and widely spread. The variability and fast-growing market makes time-to-market and product life cycle of embedded systems very short. With the increasing of system performance and computing power, embedded OS plays a more and more important role in embedded systems to better and fully utilize system resources for better resource control, task and memory management, real-time performance and even power-awareness. Due to the characteristics of various requirements of embedded products, porting OS to different hardware environments seems inevitable. Since reconfigurable processors have become a trend and can be more easily configured according to the target application, porting a suitable OS to a target platform becomes more critical in the development of embedded systems because each change in the target platform might result in modification or new porting of the embedded OS.

Therefore, there is a strong need in designing a fine-granular configurable embedded OS which allows the developers to create or modify kernel services according to the needs of their embedded systems. Configurability of embedded OS is important for optimizing systems with respect to specific characteristics. Although the kernel components of some general-purpose OSes can be customized for application specific purposes, implementing a new component may be very difficult. For example, it takes much time and effort to understand and modify a monolithic kernel. Configurable OS enables developers to build their own system-specific OS and makes it suitable for a wide range of embedded applications. Configurability also can ensure that the footprint is minimized as all unnecessary functionalities and features can be easily removed from the kernel.

In this paper, we introduce COS for embedded SoC systems. COS is the first to introduce the idea of applying SOA to embedded OSes. COS kernel is embedded with general interfaces for kernel components, including memory management, scheduler, interrupt handling interfaces, etc. A kernel component can be plugged or unplugged to its interface according to the application needs. Services of other OSes can be implemented as components of COS and can therefore perform the specific functionalities which are most preferable to the target embedded SoC system. This innovation will enable embedded system developers to work within a familiar and proven environment, facilitating code reuse, improved performance, and shorter development cycles. The main idea of COS is to allow system developers to impose their solutions of requirements on the kernel components, which includes application functionality and platform specification, whereas general-purpose OS has constrained this kind of implementation. The interfaces of COS are open and standardized as abstraction layers to the COS kernel and thus it is easy to implement new components. Nevertheless, by the open and standardized interfaces, the service components can be provided by open source community as well as proprietary contributors. As more and more developers work toward extending functionalities and contribute their components, COS

Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)
0-7695-2676-4/06 $20.00 © 2006 IEEE

can further meet more requirements of different application domains and thus helps to shorten the development time and time-to-market.

The rest of the paper is organized as follows. Section 2 presents previous works on OS configurability. The design and implementation of COS is presented in Section 3. Section 4 details an implementation of COS on an ARM evaluation board and analyzes its performance impacts. Finally, Section 5 is the conclusion of this paper.

## II. Previous Works

Paolo Gai and et al. proposed S.Ha.R.K., a dynamic configurable kernel architecture designed for integration and evaluation of scheduling algorithms [4]. However, the configurability of S.Ha.R.K is basically at scheduler level. Although it is a good platform to test new scheduling algorithms, it is not suitable in embedded systems due to other part of the kernel cannot be configurable as well.

Carsten Böke and et al. presented the use of the DREAMS component library for rather different classes of applications [1]. The DREAMS component library can be customized at source code level and configured in a very flexible way. This approach is flexible and configurable based on the proposed component library. However, the configurability of the proposed OS is thus limited and might be difficult to extend the proposed library because it does not emphasize on the abstraction of interfaces.

Michael Clarke and Geoff Coulson described the design of a dynamically extensible OS, DEIMOS [3]. DEIMOS does not define a kernel entity but specifies services as modules which can be loaded, configured and unloaded on demand by a Configuration Manager. However, some kernel services cannot be loaded at runtime, such as memory management for which using virtual memory or not has to be determined in compile-time as kernel itself needs to be loaded in the memory for execution. Therefore, its configurability is limited in the hardware platform dependent way.

FLUX OSkit leads developers to a faster start point with a set of libraries of hardware dependent and messy codes including bootstrap, management, drivers, etc, and thus save some development time [8]. However, developers still need to implement the other parts of the OS according to their needs and the implemented OS may have performance issues.

We propose COS, which obtains higher configurability in both software and hardware dependent considerations. COS enables developers to build their own system-specific OS by configuring OS service components via general kernel interfaces. Due to the abstraction of the interfaces, new services can be easily implemented and thus the growth in functionality can be unlimited, which makes COS very suitable for all kinds of embedded SoC systems.

## III. Configurable OS on SOA

As shown in Figure 1, in order to meet various needs of embedded applications, we introduce the idea of SOA to design general interfaces of OS components, including scheduler, memory management, interrupt handling, etc. Components of a specific OS service can be "plugged" to the interface. OS kernel and embedded application need no modification to the source code. COS can be easily configured to meet requirements of various embedded applications due to the idea of SOA. The components can be implemented in COS as monolithic kernel or in the user space as microkernel. COS design is able to be adapted to other OSes as different implementations of the COS.

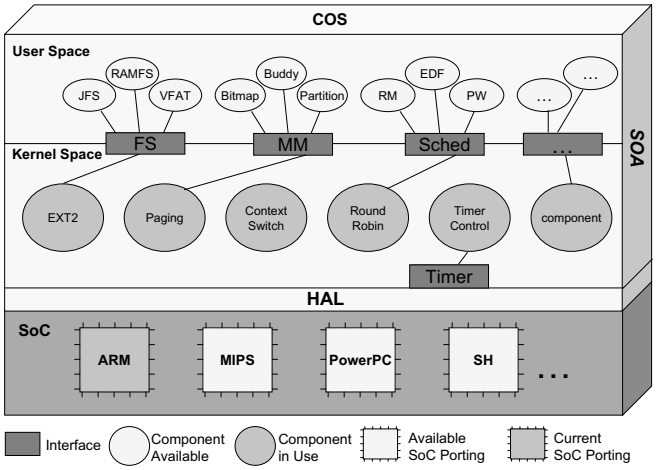


Fig. 1. COS Architecture

The proposed interfaces need to be general to support various implementations and transparent to both kernel and applications. From kernel's viewpoint, the interface is an abstraction layer between the kernel and some specific services. The configurability of COS also provides a standardized mechanism to extend its functionality and allows systems to be built from a rich set of optional configurable components. Developers are able to select components with particular implementations that satisfy the specific requirements of the system. For example, developers can configure the kernel with a real-time scheduler to meet the timing constraints of the system, or a MMU-less memory management is used for the target platform without MMU.

OS components using the COS interfaces can be configured in two different methods. Services that are not used at system boot time can be configured at runtime which would not cause the kernel size to be unnecessary large when the component is not needed and loaded, such as real-time or power-aware schedulers. On the other hand, compile time configured components allow developers to control the components at the earliest stage. Basically, these components are tightly coupled with the system operation and functionality and may affect the system performance a lot such as memory management and interrupt handling. Compile time configurability gives the best result in terms of code size because it can be done at the individual statement level in the kernel source code rather than at the function or object level. This makes compile time

Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)
0-7695-2676-4/06 $20.00 © 2006 IEEE

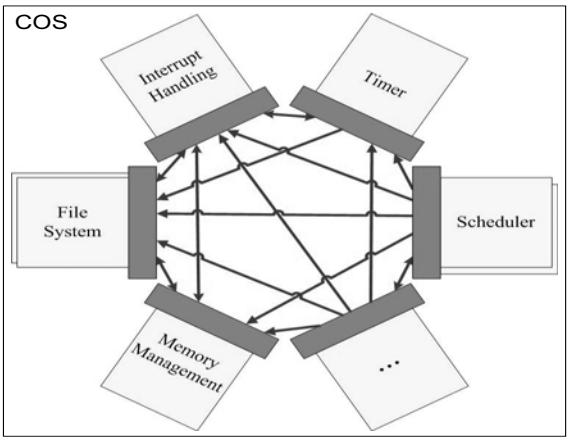


Fig. 2. COS Interfaces Example

configurability even more suitable for embedded systems and only the necessary components are included in the kernel image.

Focusing on building standardized interfaces first rather than implementation does provide a right way to reach configurability. Although the COS interfaces are designed in a generic way as much as possible, developers might want to change the interfaces because of performance or other new concerns. It is possible to extend the interfaces or define sub-interfaces within an interface to extend the configurability of COS.

## IV. Implementation of COS

COS kernel calls the functions provided in the proposed general interfaces when requesting for OS services. The interfaces are function pointers that point to the actual functions of the configured OS components performing the services, such as scheduling or disabling interrupts. The actual functions of each services are registered when kernel initializing for compile time configured components and when loading into kernel for run-time configured components. The COS design can be applied to both existing microkernel and monolithic kernel or even to build a specialized new OS. When implementing the COS design to an existing OS, modification of the kernel source code is inevitable. We need to patch the target kernel with the proposed interfaces and then substitute with the actual functions of selected components. The implementation of COS based on monolithic ARMLinux kernel is shown in Figure 2. The kernel and components communicate through the general COS interfaces. For example, kernel obtains the next executing task from the scheduler via the scheduler interface. The scheduler interface has the actual scheduling function of the selected scheduler in the system to perform scheduling. The scheduler will set the timer for the next re-scheduling time via the timer interface.

### A. Interrupt Handling

We divide the interrupt interface into two sub-interfaces, top half and bottom half. Top half functions can be considered as part of a hardware abstraction layer between kernel and interrupt controller. Interrupt handling using simple mechanism needs only the top half sub-interface, such as $\mu$C/OSII. Developers can register/remove an ISR of a specified IRQ (Interrupt ReQuest line) and also execute all ISRs (Interrupt Service Routines) when necessary via the interface. Other functions are used to initialize, enable/disable, mask, clear the IRQs on the system.

### B. Timer

The COS timer interface acts as part of hardware abstraction layer between kernel and system hardware timers. Developers can initialize, start/stop, set interval and operating mode of a timer. Time unit for the specified timer can also be set via the timer interface which help developers to specify time variables transparent to hardware timer.

### C. Scheduler

The COS scheduler interface includes function pointers and parameters of a scheduler component. The COS kernel can register/unregister tasks to a scheduler, start/stop scheduling, obtain the next executing task, etc through the scheduler interface. Therefore, on-line and off-line scheduling algorithms can be independently implemented as a scheduler component without detail knowledge and modification to the OS kernel. Since schedulers are run-time configured components, we also implement corresponding system calls for developers to select a scheduler, register a task to a specified scheduler, etc.

### D. Memory Management

The COS memory management interface defines initialization, allocate/free memory operations. It also maintains the number of total, used, and free memory and pointers to free and used memory pool for kernel usage.

### E. File System

Linux uses VFS (Virtual File System) as an abstraction layer for file systems to enable kernel to perform operations on various underlying file systems through same interface. In COS, we also adapt VFS as file system interface. VFS defines general file operations including *open()*, *read()*, *write()*, *llseek()*, etc [2].

## V. Case Study

We apply the COS design to ARMLinux [6] and port it on a Creator ARM9 evaluation board [7]. Components are implemented for important OS services to prove the configurability of COS.

### A. Implementation

We modify the ARMLinux kernel to accommodate the COS interfaces and construct service components. When OS services are needed, the COS interfaces are invoked instead of the original ARMLinux kernel calls. Since COS provides general interfaces, the most effort is to redirect original kernel calls to the COS interfaces and re-construct parameters to pass to the components. Services of other OSes or new components

Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)
0-7695-2676-4/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

TABLE I
COMPONENTS OF THE COS IMPLEMENTATION

| Service | Component | Config. Method |
|---|---|---|
| Interrupt Handling | Top/Bottom Half<br>Top/Bottom Half w/ tasklet | Compile Time |
| Scheduler | Rate Monotonic<br>Earliest Deadline First<br>Pinwheel<br>Round Robin | Run-Time |
| Memory Management | Paging with MMU<br>Paging without MMU<br>Partition and Block<br>Bitmap | Compile Time |
| File System | EXT2<br>RAMFS<br>VFAT | Run-Time |

TABLE II
OVERHEAD OF COS INTERFACES

| Interface Name | w/o ($\mu$s) | w/ ($\mu$s) | Overhead of Interface ($\mu$s) |
|---|---|---|---|
| do_irq() | 0.886 | 1.107 | 0.221 |
| mask_irq() | 0 | 0 | 0 |
| unmask_irq() | 0 | 0.526 | 0.526 |
| do_softirq() | 0.479 | 0.625 | 0.146 |
| mem_init() | 85170 | 85280 | 110 |
| kmalloc() | 10 | 10 | 0 |
| kfree() | 0 | 0 | 0 |

need to conform to the COS interface. Table I shows components of current COS implementation. We implement several components for each service [10], [9] and test COS with CPU bound and I/O bound tasks including a MP3 decoder using various combinations of implemented components and thus proves the configurability and feasibility of COS.

### B. Performance Evaluation

In order to evaluate the performance of COS, we measure the overhead of proposed interfaces. Table II shows the overhead of COS interfaces in average and we can see the overhead is manageable. The zero executing time is because actual value is less than a timer tick and cannot be measured. *mem_init()* performs considerable *kmalloc()* and *kfree()* operations during initialization and thus the overhead is much larger. The overhead of interfaces without parameters reconstructing, such as $schedule()$ of the scheduler interface, can be neglected due to the implementation of function pointer. In order to reduce the overhead of COS interfaces, macros and inline assembly can be used to further reduce the extra function calls and parameters reconstructing of current implementation.

Kernel components need to be carefully implemented according to specific requirements of target embedded system. In our implementation of the paging without MMU memory management component, we disable the MMU of the ARM9 core on the Creator board to test the MMU-less memory management service. Due to the hardware limitation of the ARM9 processor, disabling the MMU also disables the data cache [5], which results in about 7 times degradation in performance (Table III). It proves that each service component needs to be carefully selected or implemented according to its application and platform requirements. It also shows the benefit from configurability of COS to easily meet various requirements of embedded systems.

TABLE III
PERFORMANCE IMPACT OF MMU AND DCACHE

| Mem Manage. Component | Allocate ($\mu$s) | Access ($\mu$s) |
|---|---|---|
| Paging with MMU | 316 | 137 |
| Paging without MMU | 208 | 994 |
| Paging with MMU (DCache disabled) | 742 | 876 |

## VI. CONCLUSION

In this paper, we introduce the strong need of configurability in embedded OS and present the idea and architecture of the configurable OS design (COS) for embedded SoC systems. We show how to apply service-oriented architecture (SOA) on OS design and separate implementation from interfaces. Implementation issues of COS are also discussed. An implementation of COS based on ARMLinux on an ARM9 evaluation board is presented and performance is analyzed to prove its feasibility and configurability. We show that functionalities of other OSes or new OS services can be easily implemented as COS components by conforming to the COS interface design. Moreover, COS can accommodate to other existing OS implementations as well. Since "COS" is a reverse of "SOC", we believe COS is very configurable, can meet specific SoC design requirements, and can fully utilize the embedded hardware and software resources.

## REFERENCES


[1] Carsten Böke, Marcelo Götz, Tales Heimfarth, Dania Adnan El-Kebbe, Franz Josef Rammig, and Sabina Rips. (re-) configurable real-time operating systems and their applications. In *the IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, January 2003.

[2] M. Cesati and D. P. Bovet. *Understanding the Linux Kernel, 2nd Edition*. O'Reilly, second edition, 2003.

[3] Michael Clarke and Geoff Coulson. An architecture for dynamically extensible operating systems. In *the 4th International Conference on Configurable Distributed Systems*, 1998.

[4] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. In *the 13th IEEE Euromicro Conference on Real-Time Systems*, pages 199–206, June 2001.

[5] ARM. Internet homepage. http://www.arm.com/. 2006.

[6] ARM-Linux. Internet homepage. http://www.arm.linux.org.uk/. 2005.

[7] Creator Development Board. Micortime Internet homepage. http://www.microtime.com.tw/product/product.htm. 2004.

[8] Flux Operating System Toolkit. Internet homepage. http://www.cs.utah.edu/flux/index.html/. 2005.

[9] Chih-wen Hsueh and Kwei-Jay Lin. Scheduling real-time systems with end-to-end timing constraints using the distributed pinwheel model. *IEEE Transactions on Computers (SCI)*, 50(1), January 2001.

[10] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 10(1):46–61, 1973.



Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)
0-7695-2676-4/06 $20.00 © 2006 IEEE