Proceedings of the 5th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems, Madrid, Spain, February 15-17, 2006 (pp110-116)

Increasing the Determinism in Real-Time Operating Systems for ERC32 Architecture

A. Viana O.R. Polo P. Parra O.G. Población I.G. Tejedor S.S. Prieto
D. Meziat
Computer Engineering Department
University of Alcala, Alcala
Campus Universitario, 28871 Alcala
SPAIN

Abstract: To develop on-board software in embedded space applications, it is often a requirement to use a real-time operating system, since these systems must assure that the timing constraints associated to each task are guaranteed. Furthermore, it's desirable that the execution time of the real-time application in any scenario is as deterministic as possible (i.e., it takes always the same time to perform the task). Typical real-time operating systems focus on increasing performance by focusing on scheduler design, device driver writing, heavy machine-level code optimisations, etc. However, the main goal must be to behave in as a deterministic way as possible. The work related to this paper is centred in real-time, embedded operating systems research on the ERC32 architecture, an E.S.A. (European Space Agency) standard architecture for space applications. This architecture is SPARC V7 instruction set compliant and has a lot of features built-in for space applications, but some aspects of the architecture can, in some cases, decrease the system determinism. Our main research tries to obtain a real-time operating system that has a deterministic behaviour running on this architecture, by implementing new management paradigms over some architectural aspects, enhancing the time accuracy, allowing worst case execution time analysis and improving the context switch operations.

Key-Words : Real-time Operating Systems, Embedded Systems, On-Board Software, ERC32

1 Introduction

Currently, embedded software in space environments has increased its complexity and consequently needs real-time multitasking operating system (RTOS) presence. A good real-time operating system is not the fastest one but the more precise in its execution [1]. This means that consecutive executions of the embedded software over the RTOS must take the same time. Sometimes, this characteristic depends on hardware architecture and system developers must take into account many hardware aspects to make a deterministic system.

Our research is focused in the European Space Agency (E.S.A.) standard architecture for space applications, ERC32 [2, 3]. We are trying to develop an integrated development environment to perform design, modelling and automatic code generation for space applications on this architecture. The environment consists of the EDROOM [4] tool and ERCOS-RT real-time operating system. The integrated environment must provide a real-time solution with the main goal of obtaining deterministic operation.

This paper exposes what are the causes for indeterministic behaviour in real-time systems, what problems are derived from the ERC32 architecture, what are the solutions proposed and the improvement derived from our solution. Section 2 makes a brief explanation about the ERC32 architecture, section 3 describes the indeterminism problem in this archi-

tecture, the solution proposed for it and some timing measurements carried out over our real-time operating system ERCOS-RT (which implements the proposed solution), and other real-time operating systems. Finally, section 4 shows the conclusions of this work.

2 ERC32 architecture

The ERC32 implementation is based on the SPARC V7 architecture [5]. The work explained in this paper has been developed over the Tharsys SPARC RT Single Board Computer, built around the radiation tolerant ERC32 chip set (TSC691, TSC692 and TSC693) manufactured by Themic Semiconductors. This platform has a number of on-board peripherals developed by E.S.A., targeted at space applications such as:

- UART channels (A and B)
- General Purpose Timer
- Real-Time Clock
- Watchdog Timer
- Ethernet Interface
- VME interface
- Parallel interface

The SPARC architecture defines a set of registers and register windows visible to applications executing at any given time. This architecture defines 32 globally accessible floating-point registers that can be viewed as 32 single precision floating or integer registers (f0...f31), 16 double precision floating point registers (f0, f2, f4, ..., f30) or 8 extended precision floating point registers (f0, f4, f8, ..., f28). There's also another 8 globally accessible general purpose registers, and finally there's a set of sliding windows that at any given time allow access to a group of 24 registers divided into three sets of 8 registers each; input registers, output registers and local registers. Finally, it has special registers such as Processor State Register (PSR) and Window Invalid Mask (WIM). The first one has the Current Window Pointer (CWP) field that stores the current window and together with the WIM register, manages the SPARC register windows. Figure 1 shows the described register architecture.

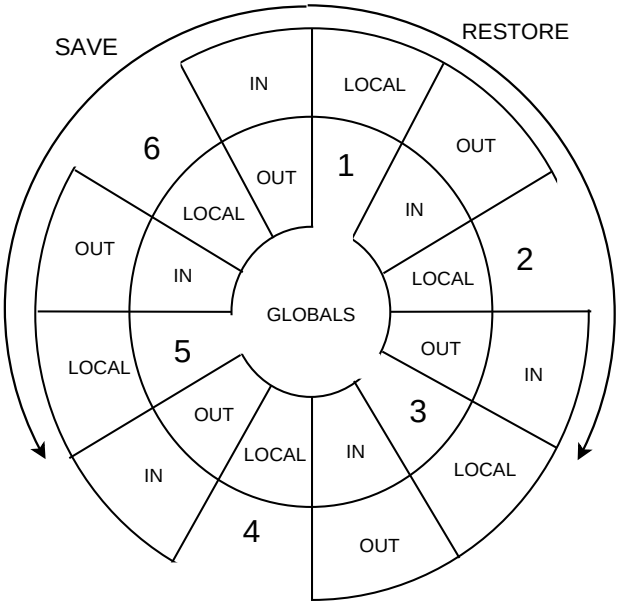

Figure 1: SPARC six register windows representation.

2.1 Register windows

The SPARC architecture implements the concept of sliding register windows. This concept tries to simulate that the architecture has and infinite set of fresh register sets available for each subroutine through some clever but also somewhat complicated register manipulation.
When performing subroutine calls, a `save` instruction is executed. This instruction decrements the CWP field, effectively sliding the current register window and providing a new set of registers ready to be used, where the subroutine stores its parameters and local variables. When returning from the subroutine, the `restore` instruction is executed, which increments the CWP field and makes the previous register window available again.
The ERC32 architecture has eight register windows, so having a subroutine call deeper than the number of register windows would result in window overwriting. This situation is called overflow, and it generates

a trap so that the appropriate trap handler can store the window register contents to be preserved in its window stack. When all subroutines return, it begins to perform `restore` operations. This operations could cause the need to restore the valid register information from the window stack to the register window. This is named as window underflow and it also generates a trap, so that the trap handler associated with it can restore the register contents from the window stack to the register window. Figure 2 shows the overflow and underflow conditions.

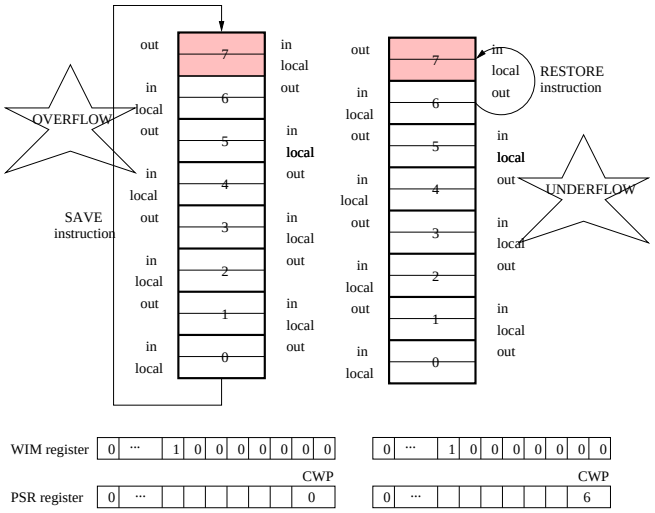

Figure 2: Overflow and Underflow Conditions.

The WIM register and CWP field are used together to manage this overflow and underflow situations. The CWP field contains the window that is currently in use and the `save` and `restore` instructions decrement and increment the CWP modulo the number of register windows. Each bit in the WIM register represents whether a register window contains valid information for another subroutine. A bit set to "1" represents valid information (invalid window) and a bit set to "0" represents a free register window. When a `save` instruction causes the CWP field point to a register window marked as invalid window, a overflow condition is triggered. The same mechanism occurs when the `restore` instruction triggers an underflow trap.

Another complicating factor is the sharing of registers between adjacent register windows. Each register window has local, input and output registers. The local registers are owned by the window, the input registers are shared with the output registers of the register window ((N + 1) modulo RW), with RW being the number of register windows, and the output registers are the same as the input registers of the register window ((N - 1) modulo RW). Figure 3 shows this situation.

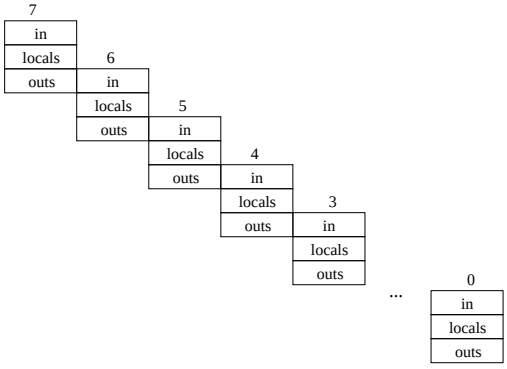

Figure 3: Sparc register sharing mechanism.

The register sharing mechanism improves parameter passing performance because all operations are register-based instead of stack-based. The caller loads into its output registers the callee's parameters. After the caller executes the `save` instruction, its output registers turn into the callee's input registers, thereby passing the parameter data. This is a very efficient mechanism to pass parameters because no data is actually moved by the `save` and `restore` instructions and no access from-to memory is performed. Only when the overflow-underflow conditions are triggered, the architecture mechanisms introduce a bit of time penalty to execute the associated trap handlers.

3 Real-Time Performance

There are some approaches that try to increase real-time performance by improving aspects of the operating system such as: specific scheduling policies, device driver optimisation, hand-made hardware-dependant code, etc. These aspects are very important in real-time operating systems development,

but they mainly focus in performance and low latency while probably the most important aspect of a real time system is determinism, which leads to predictability. Because in a real-time environment we have a lot of external events that appear asynchronously, the system execution becomes less deterministic and the real-time operating system should make this indeterminism even worse. The next section explains this problem in ERC32.

3.1 Problems with ERC32 to obtain a deterministic system

According to section 2, ERC32 is SPARC V7 compliant and this architecture, implementing the sliding windows mechanism. It has been shown how this mechanism decreases the time to perform inter-routine parameter passing, thus obtaining a very fast system, but on the other hand it decreases the determinism in multitask systems.

In a system with only one task, the sliding window mechanism does not affect determinism because the same task is being executed all the time, and since no context switch occurs, no other task can change the system state. All the register windows belong to the same task.

On the other hand, in real multitasking systems, external events would change the tasks execution flow, and a thread could stumble upon an overflow/underflow condition in one execution scenario but not in another execution of the same scenario. In this case the operating system should assure that the thread execution takes always the same time, so it must assure that the register window state will be the same in each system execution, in order to execute the same operations in all scenarios, thus obtaining a deterministic behaviour.

3.2 New sliding window management

The classic SPARC window management performed by several operating systems that support the ERC32 architecture, such as RTEMS [6], ORK [7], Linux, etc. configure the system so that there's one invalid window and all other are marked as valid. With this configuration it is not possible to know when a overflow/underflow condition will occur. As a result, the time to execute the real-time applications over the RTOS is indeterminate.

We propose a new way to manage the sliding window mechanism. With this new management technique, we configure only one window as valid, and all the rest as invalid. In the figure 4 the new window management is shown.

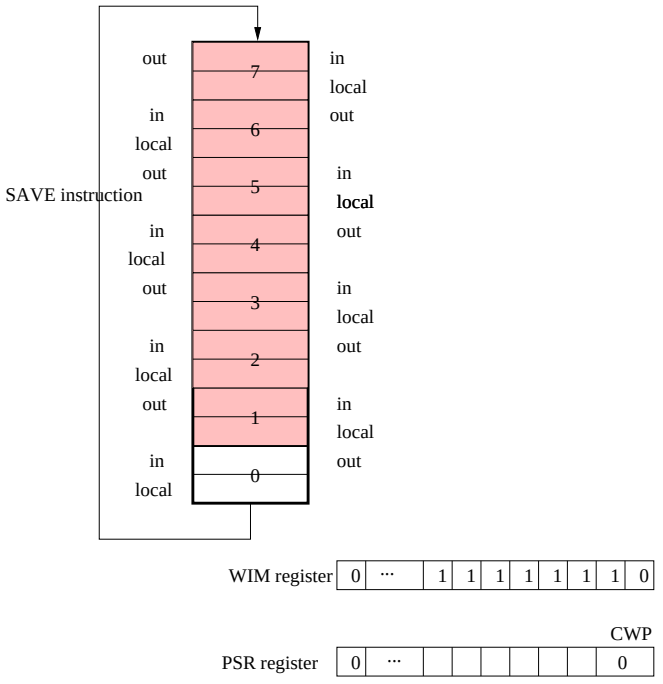

Figure 4: Proposed window management configuration.

The configuration explained above forces an overflow/underflow condition in every function call, making execution slower than using classic sliding window management; but on the other hand, the system always uses the same time in every routine execution and all the threads find the same register window situation, increasing system determinism.

Another important issue is that, with this configuration, the system always the worst case scenario, so the timing analysis would be performed with more accuracy.

3.3 Timing analysis

Nowadays, it is important to provide a mechanism for measuring the time spent by the operating system in performing some operations. To provide this mea-

surement capability, it is necessary to force the worst case execution scenario and it is not always possible.

The classic window management approach does not take into account this worst case scenario, and asynchronous events could change the execution flow. This fact implies that all the measurements carried out with this management could change with different executions.

The window management proposed places the architecture in the worst case execution scenario so, it would be easy to implement kernel routines to make timing analysis and all the results produced will be the same in every execution.

3.4 Timing accuracy

In real-time operating systems some of the most common operations performed by execution threads is to request a delay, to wait for some event, to synchronise with some other thread, to wait for some data, etc. The system must wake up the threads at the precise requested time, and the accuracy is crucial in most cases. Due to the fact that the classic window management does not guarantee the time to perform a routine execution, accuracy is lower and non-deterministic, and the system could wake up a thread before the time requested because the routine that wakes up the thread may cause window overflow or may not.

The window management proposed on this paper ensures that the time to perform all call operations are the same, increasing the wake up time accuracy and consequently the system determinism.

3.5 Fast Context Switch

The SPARC V7 architecture (and ERC32) [5] has a total of 167 user-allocatable registers, of which 128 are used for the sliding windows mechanism implementation explained in section 2.1. All the known kernels implemented for ERC32 architecture such as ORK [7], RTEMS [6], eCos [8], etc. uses the classic window management showed in figure 2.

With classic window management, when a thread is granted the CPU it can run for some time without performing any window overflow, and consequently without saving its context. What this means is that, since the time spent to perform a context switch varies depending on the number of windows that must be flushed, this time varies depending on the thread behaviour and each execution scenario, increasing the kernel indeterminism. When a context switch is performed it is necessary to flush all the non saved windows used by the thread onto the thread stack or else part of the complete thread context would be lost. The worst case execution time for the context switch happens when all the windows must be saved, in which case the time employed to perform the context switch can be quite high.

One of the distinctive requirements for real-time systems is that deadlines must be met. It is known that the caches exploit the locality of references to reduce memory access latencies [9]. When an operating system performs a context switch there is a cache interference cost that must be taken into account when calculating the worst case execution time and, in many cases, it is increasingly hard to determine it. In the same way, the classic window management makes the system faster, because not always an overflow/underflow conditions are triggered, but its main drawback is that it decreases the system determinism, because it is impossible to predict when this conditions will be triggered in a concurrent system.

With the new sliding window treatment proposed on this paper, all the `call` and `restore` operations cause overflow/underflow conditions and the associated interrupt handler stores/restores the register window into its stack. This mechanism makes the code execution slower than with classic window management, but the time spent is always the same and the kernel is more predictable [10]. Additionally, since each window is saved when it is left, it turns out that the worst case scenario to perform a context switch is better than with classic window management and moreover, this time is deterministic for any execution scenario. With this window management, in all context switchesthe kernel must only save the current thread window, avoiding the flushing of any other windows onto the stack because each window has already been saved when necessary.

We have developed a real-time operating system, named ERCOS-RT, over which this new window management has been implemented. The context

Operating System	Platform Configuration	usecs
RTEMS	ERC32 16 Mhz	128
ORK	ERC32 10 Mhz	85
ERCOS-RT	ERC32 16 Mhz	31

Table 1: Context Switch Latency

switch latency has been measured by implementing the LMBENCH [11] test suite over the real-time operating system. ERCOS-RT takes 31 microseconds to perform a context switch in the hardware platform described in section 2, without saving the floating point registers. The context switch kernel routine takes only 14 microseconds to perform the task switch, but the benchmark implemented also measures the time the kernel needs to propagate the event that performs the task switch. The same tests have been executed in the RTEMS [6] kernel and it takes 128 microseconds. The routine that performs the RTEMS task switching takes 10 microseconds, but this value does not take into account the time to perform all the needed window flushing. The worst case execution time for the tasks switch in RTEMS would be 10 microseconds plus the time spent to flush the maximum number of windows, which could be up to seven in the case of the ERC32 architecture. The table 1 shows some ERCOS-RT comparisons with other real-time kernels for the ERC32 architecture, such as the Open Ravenscar Kernel [7].

4 Conclusions

We have determined that one of the most important things in a real-time operating systems is not just performance or low latency, but its predictability. A deterministic system allows much better constraint evaluation, and leads to more robust and precise systems. Chasing this goal, we have proposed a method to enhance predictability in the ERC32 architecture that guarantees a more predictable task behaviour in a real time environment, and that at the same time obtains better performance and a deterministic behaviour in context switch routines. This method is implemented in the ERCOS-RT operating system and is the basis for time-constrain analysis tools under way.

References

[1] Stankovic J.A. Misconceptions about Real-Time Computing, a Serious Problem for Next-Generation Systems. *IEEE*, October 1998, pp. 10–18.

[2] Gillier L., Via A.D., and Dherbecourt A. *VME Sparc RT Single Board Computer*. European Space Agency.

[3] Stachetti V., Gaisler J., Goller G., and Gargasson C.L. 32-bit processing unit for embedded space flight applications. *IEEE Transactions*, vol. 43, June 1996, pp. 873–878.

[4] Polo O.R., la Cruz J. M. D., J.M. G.S., and S. E. EDROOM. Automatic C++ Code Generator for Real-Time Systems Modelled with ROOM. In *NTCC2001 IFAC Conference*, November 2001.

[5] Sparc International INC. *Sparc Architecture Manual Version 7*.

[6] On-Line Application Research Corporation. RTEMS SPARC Applications Supplement, september 2000.

[7] de la Puente J.A., Zamorano J., Ruiz J., Fernández R., and García R. The design and implementation of the open Ravenscar kernel. In *IRTAW '00: Proceedings of the 10th international workshop on Real-time Ada workshop*. ACM Press, New York, NY, USA, 2001.

[8] Redhat. eCos Reference Manual, September 2000.

[9] Stärner J. and Asplund L. Measuring the cache interference cost in preemptive real-time systems. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. ACM Press, New York, NY, USA, 2004. ISBN 1-58113-806-7.

[10] Halang W.A. Contemporary Research on real-time schedulin considered obsolete. In *27th* IFAC/IFIP/IEEE *Workshop on real-time programming, WRTP'03*, May 2003.

[11] McVoy L. and Staelin C. LMBench: Portable tools for performance analysis. In USENIX *Annual Technical Conference*, January 1996.

[12] Selic, B., Gulleckson, G., and P.T W. *Real-Time Object Oriented Modelling.* John Wiley and Sons, 1994.

[13] Harel and David. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, (8), 1987, pp. 231–274.

[14] Viana A., Polo O.R., Lopez O., Knoblauch M., Prieto S.S., and Meziat D. EDROOM: a free tool for the UML2 component based design and automatic code generation of tiny embedded real-time systems. In *3 European Congress in Embedded Real-Time Systems*, January 2006.

[15] Polo O.R., S. E., Grau A., and de la Cruz J. Control Code Generator used for Control Experiments in Ship Scale Model. In *CAMS2001 IFAC Conference*, July 2001.

[16] UML 2.0. www.u2-partners.org/uml2wg.htm.