RTOS Acceleration Techniques – Review and Challenges

M. Sindhwani, T. F. Oliver, D. L. Maskell And T. Srikanthan
Centre for High Performance Embedded Systems (CHiPES)
Nanyang Technological University, Singapore
emohit@ntu.edu.sg, tim.oliver@pmail.ntu.edu.sg, {asdouglas, astsrikan}@ntu.edu.sg

Abstract

As embedded systems evolve, there is greater dependency on the real-time operating system (RTOS) to abstract the complex hardware. In return for the services provided, the RTOS consumes CPU cycles, thereby imposing a processing overhead on the CPU. In this paper, we review some of the techniques that have been proposed in literature for reducing the CPU utilisation by the RTOS primitives, including our work on two specific platforms – a multi-core processor and a soft-core processor. We clearly demonstrate that the benefits of adopting a suitable RTOS acceleration strategy are significant. We also look at some of the challenges that face the RTOS industry if they are to adopt these mechanisms. It is our belief that the self-maintaining nature of open source software makes it a very viable candidate for benefiting from these approaches.

1 Introduction

As embedded systems evolve, there is increasing reliance on run-time software, such as a Real-Time Operating System (RTOS), to abstract the underlying hardware and offer a consistent API to the application developer. However, this has resulted in the RTOS layer becoming more demanding, and consuming a greater percentage of CPU time. The CPU overheads imposed by the RTOS reduce the time available for processing user tasks. In a sample implementation, we found that the overheads imposed by the RTOS can be as high as 27% [1].

In this paper, we look at some of the methods proposed in literature for managing the CPU overheads imposed by the RTOS. To introduce the case, we begin by looking at the advanced processing options that are being made available in embedded systems. This is followed by a review of techniques that make use of these hardware features to reduce the RTOS overheads in these modern platforms.

The paper is organized as follows. The next section provides a brief introduction to hardware options in modern embedded systems. This is followed by a review of RTOS acceleration techniques proposed in literature, including our work with a multi-core processor and a soft-core processor. In Section 4, we discuss the problems and challenges that confront RTOS developers who wish to adopt these techniques for application-specific optimization of the RTOS. Finally, our conclusions are presented in the last section.

2 Embedded Hardware

Three main advancements have become common in the embedded space. This section lists these three advancements, and describes them briefly.

Advanced Embedded Processors: Owing to the advances in VLSI technology, more logic gates can be incorporated onto the same area of silicon. Modern process technology also results in systems that are more power-efficient and higher performing. Greater degrees of integration in embedded processors have resulted in multi-core processors that combine DSP and RISC functionality [2], CPU and I/O processors [3], or multiple CPUs. Such processors are becoming more common in embedded processing due to the advantages they provide.

FPGA Space: The availability of field-programmable gate array (FPGA) space in modern embedded systems allows designers to accelerate compute-intensive operations in hardware. Also, the re-programmable nature of the FPGA allows the hardware modules to be upgraded even after deployment. Common models of FPGA usage in modern embed-

ded systems typically incorporate an FPGA + CPU combination, either as separate chips, or as a single integrated chip [4].

Configurable Systems-on-Chip: The rapid increase in the gate counts of FPGA devices has enabled the creation and deployment of entire systems on FPGAs. The Configurable System-on-Chip often uses a soft-core programmable processor, running a real-time operating system, as the central processing element in the system. Typically, the instruction set of the processor can be extended to incorporate custom instructions that reduce a complex sequence of software instructions to a single instruction implemented in hardware. In fact, it has been predicted that by the end of the decade, nearly 80% of semiconductor products sold in the SoC class will belong to the configurable system-on-chip category [5].

These advancements in hardware offer new avenues for optimizing the RTOS and making it more efficient. In the subsequent sections of the paper, we see what options have been proposed for optimizing the RTOS in the modern embedded system.

3 RTOS Acceleration Techniques

In this section, we review the four main approaches that have been proposed for RTOS acceleration. These include co-processor assisted acceleration, hardware-assisted acceleration, hardware RTOS and instruction set customization.

3.1 Co-processor approach

There are a few approaches based on the use of a co-processor. In [6], the idea of a task-scheduler co-processor for hard real-time systems is presented. The proposed design uses an external 8032 microcontroller as the task scheduling co-processor and can handle up to 32 tasks. In this design, all interrupts are routed to the co-processor so that full scheduling management can be performed by the coprocessor. The coprocessor is designed on a separate board that connects to the main CPU board over the systems bus. Communication between the coprocessor and the target is by using interrupts.

Our own work [7] is based on a similar concept, but relies on an on-chip programmable I/O processor. The Infineon TriCore TC10GP microcontroller [8] was used. The MicroC/OS-II [9] tick scheduler was ported to run on the Peripheral Control Processor (PCP), while dispatch operations continued to be carried out by the CPU. The PCP communicated with the main CPU using interrupts.

The RTOS overheads on the main CPU are reduced because of two factors:

- The number of times the main CPU is interrupted is reduced from the system timer tick interrupt frequency (e.g. 100 times a second) to the frequency at which tasks are made ready in the system – this results in fewer interrupts to the main CPU, resulting in better CPU utilization by user tasks.

- Also, the time of the tick ISR on the main CPU was reduced by about 1600 cycles, thereby reducing the cost associated with each invocation of the interrupt service routine.

Results: Due to the manner in which the RTOS was split and the timer interrupts were separated from the main CPU, it was found that the overheads (in clock cycles) on the main CPU depended only on the number of tasks that were made free every second and were not directly affected by the number of tasks in the system, the frequency of the system timer tick, the cost of executing the scheduling algorithm or the frequency at which the CPU was operating (this has implications in power management).

Such an approach is readily applicable to any multi-core processor in which two or more independent processing units exist.

Finally, efforts to split some of the overheads of message passing in massively parallel processors are presented in [10]. Communications co-processors provide dedicated hardware support for fast communication that can be exploited for executing user-level message handlers, thus freeing the main processor for computational work. Since the CPU is freed from the tasks required for message handling, this mechanism allows overlap of computation and communication to a greater extent. Results in the paper show improvements as high as about 3 times in the execution of some benchmark programs.

3.2 RTOS Primitives in Hardware

A number of efforts to port portions of the RTOS to hardware have been presented in literature. The main motivation for these efforts is to:

1. Reduce RTOS overheads on the CPU.

2. Implement more complex and comprehensive algorithms for RTOS tasks.

F-Timer: In [11] there is the proposition of using an external FPGA that provides dedicated hardware units for maintaining a 32-task list, organized by time priority. It provides a time resolution of 100μS and the various interrupt modes and tasks are programmable.

Spring Scheduling Co-processor: The Spring scheduling coprocessor [12] is a coprocessor to accelerate scheduling. Many different scheduling policies and their combinations can be used. The architecture has been designed for multiprocessor systems and it has been shown that the main portion of the scheduling operation can be improved by over three orders of magnitude [13]. Performance issues related to the co-processor are presented in [14].

For system-on-chip environments, a number of propositions have been presented. It is claimed that the hardware-assisted interprocessor communication (IPC) mechanism proposed in [15] can reduce the communication overhead of embedded microcontrollers by a factor of 30 or more. In [16], an efficient, small and simple hardware unit is proposed for reducing synchronization overheads in multiprocessor system-on-chip implementations. In [17], a hardware-assisted memory management scheme called Two-Level Memory Management is proposed for multi-processor implementations. In the paper, it is also shown how to modify an existing RTOS to support the proposed hardware.

In addition to the above, [18] discusses hardware support for distributed real-time systems and in [19], a novel deadlock detection algorithm and architecture is presented to support the RTOS.

3.3 Hardware RTOS

Hardware RTOS approaches replicate most of the software RTOS as a complete hardware entity. There are two main projects in this area.

FASTCHART and Related Projects: FASTCHART [20] is a system that consists of a fast time deterministic CPU and hardware based real-time kernel that implements the entire RTOS in hardware. An analysis of the FASTCHART approach is presented in [21]. FASTHARD [22] is a modified version of FASTCHART that can be used with any general CPU. It is connected to the system bus, and in addition, needs an interrupt line to the CPU. It can cater to 256 tasks and 8 priorities. The interface is created as a set of service calls from the CPU to the real-time unit. Similarly, the Sierra Operating System Accelerator [23] is a commercial product for RTOS acceleration that can manage 16 tasks with 8 priorities, 16 resources and 8 interrupts. It is from a company that was set up by the authors of FASTCHART and FASTHARD.

Silicon TRON: A similar approach for RTOS speed-up is presented in [24]. The authors have ported the most basic system calls to hardware and show that the hardware implementation is about 130 to 1880 times faster than the software implementations. This hardware was designed to support the TRON Operating System [25], and is similar to the other options discussed this far.

3.4 Instruction Set Customization

In [1] and [26], we proposed instruction set customization of soft-core processors as the means to contain RTOS-imposed CPU overheads. By using custom instructions to implement parts of the RTOS kernel, we reduced a complex sequence of software instructions to simpler single-cycle (combinatorial) and multi-cycle (sequential) operations, supported by hardware. To evaluate our approach and designs, we extended the instruction set of the Altera NIOS processor [27] and the open-source OpenRISC processor [28] to aid scheduling, event management and time management of the MicroC/OS-II RTOS.

The task scheduler, event control block and the timer management routines were supported by custom instructions, and the resulting performance was measured. The task scheduler module resulted in a ROM saving of about 1KByte, which would result in improved energy dissipation of the system. The maximum critical-section length was reduced by around 3% to 5%, translating into an improvement in the interrupt response time.

Individual routines showed performance improvements in the range of 50%–90%. Frequently used RTOS primitives (based on the Rhealstone benchmark [29]) showed an improvement of 10%–35% and the Dhrystone mark [30] of the system improved by as much as 13%. Although the MicroC/OS-II RTOS was used, our methods are equally applicable to other operating systems and soft-core processors. Detailed results are available in [1] and [26].

3.5 Summary

Clearly, there are a number of approaches that can be used to accelerate the RTOS and reduce the CPU overheads imposed by the RTOS.

Co-processor approaches introduce problems with data inconsistency, communication and synchronization between the multiple processors, and verification. A similar problem affects hardware approaches where the real-time kernel runs in parallel with the main CPU.

Instruction set customization offers a simple yet effective technique for improving the efficiency of the

RTOS. Since the RTOS is still a software entity, there are no concerns about synchronization or communication. Also, custom instruction verification is relatively simple.

Most of the methods above claim excellent scalability in the face of increasing number of tasks, a feature that is crucial as embedded systems become more complex.

4 Challenges

Typically, the RTOS has been treated as a full-software entity with the above techniques being offered by academia as options for making the RTOS more efficient. The actual solution that should be used depends on the requirements of the system being designed. The selection is influenced by the needs of the embedded system and the capability of the embedded system. For example, a system should use a hardware scheduler if scheduling overheads are significant in the system. However, this can be done only if there is hardware space available in the target. The actual scheduler that can be used and its performance will then depend on the amount of available hardware space.

This brings up certain challenges for RTOS designers:

1. The RTOS has typically been perceived as a software entity. The techniques highlighted in this paper rely on the availability of hardware accelerators. This would require a commercial RTOS vendor to have a team of qualified digital hardware engineers to design, integrate, and test the accelerators for different platforms.

2. Even though programmable logic space is available in modern embedded systems, the amount of space available for RTOS activities will be influenced by the specific application. It is difficult for RTOS developers to cater to this kind of an unspecified target.

3. Configurable processors allow the addition of custom instructions. The number, nature and complexity of these instructions depend on the amount of available logic space in the target hardware. Therefore, it is not possible for RTOS developers to anticipate or assume the availability of specific instructions in the target processor's instruction set.

4. System software is typically supplied by independent vendors - this means that the hardware and software teams work independently, rather than synergistically [31]. This is especially the case if the processor hardware is customized by the project team, but the RTOS is licensed from a third-party vendor.

Due to the above reasons, RTOS vendors are forced to provide more generic and non-optimal solutions. On the other hand, designers of constraint-driven embedded systems will value the ability to tweak the RTOS and carry out application-specific optimization for their specific target. This problem is likely to intensify in the future as systems developers move towards greater levels of hardware customization in their designs.

It is therefore our proposition that these techniques are best suited for the open source domain. Due to the vast number of people involved in developing, verifying and maintaining open source software, open source operating systems are good candidates to benefit from the techniques listed in this paper. Due to the open source community's willingness to share, there is a much greater chance that such techniques can be implemented, documented, maintained and effectively used. Open source software and hardware components can be developed and made available to the community. Developers can select and integrate the components they need for their system.

We feel that the most sensible way to tackle this problem is to define a framework [32] to customize the RTOS before embedding it in the system. This implies that the RTOS be customized exclusively for the application. For this to be possible and feasible, software tools are required that can automatically (or with a little help) analyze the RTOS requirements and the resource availability in the target embedded system, and customize the RTOS to optimally meet the needs. In this case, the RTOS will be auto-generated by the software tool as a set of files, some of which can be compiled to execute on the target architecture, while others can be synthesized into a hardware model.

Such a framework would need extensive information about the performance parameters, and software and hardware modules that implement portions of the RTOS. It is far more believable that the open source community would be willing to share such information.

5 Conclusion

In this paper, we have looked at the various techniques offered as solutions to the problem of reducing the CPU overheads imposed by the RTOS. Most of these techniques rely on the availability of hardware accelerators. We have identified the problems that

commercial RTOS vendors would face if they were to support these options. We have also suggested that the open source community is in the best position to develop, verify, and maintain these options. It is, therefore, our conclusion that these techniques are best suited for the open source domain.

References

[1] Z Jin, M Sindhwani and T Srikanthan, 2004, *RTOS Acceleration on Soft-core Processors Using Instruction Set Customization*, International Conference on Field Programmable Technology (FPT 2004), Australia.

[2] Texas Instruments, 2002, *TMS320VC5470 Fixed-Point Digital Signal Processor Data Manual*.

[3] Infineon Technologies, 2001, *TC1775 User's Manual System Units*. see: http://www.infineon.com/tricore/

[4] Xilinx, 2002, *Virtex-II Pro Platform FPGAs* see: http://www.xilinx.com/

[5] Balough, C, 2000, *Picking Winners in the Configurable System-on-Chip Space* see: http://www.techonline.com/

[6] Cooling J. and Tweedale P, 1997, *Task scheduler co-processor for hard real-time systems* Microprocessors and Microsystems, 20 (1997), pp. 553–566.

[7] Ramakrishnan N, 2002, *H37/01 – Towards an Independent On-Chip RTOS Manager*. Honors Year Project Report. Nanyang Technological University (2002).

[8] Infineon Technologies, 2000, *Infineon Tricore TC10GP Users Manual*.

[9] Labrosse J J, 1999, *MicroC/OS-II: the real-time kernel*, Kansas R&D Publication.

[10] Schauser K E, Scheiman C J, Ferguson J M, Kolano P Z, 1996, *Exploiting the capabilities of communications co-processors*, Proceedings of the 10th International Parallel Processing Symposium, pp. 109–115.

[11] Parisoto A, Souza A Jr, Carro L, Pontremoli M, Pereira C, Suzim A, 1997, *F-Timer: dedicated FPGA to real-time systems design support*, Proceedings of the Ninth Euromicro Workshop on Real-Time Systems, 1997, pp. 35–40.

[12] Burleson W, Ko J, Niehaus D, Ramamritham K, Stankovic J A, Wallace G, Weems C, 1993, *The spring scheduling co-processor: a scheduling accelerator*, IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 140 – 144

[13] Burleson W, Ko J, Niehaus D, Ramamritham K, Stankovic J A, Wallace G, Weems C, 1999, *The spring scheduling coprocessor: a scheduling accelerator*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume: 7 Issue: 1 pp. 38–47.

[14] Niehaus D, Ramamritham K, Stankovic J A, Wallace G, Weems C, Burleson W, Ko J, 1993, *The Spring scheduling co-processor: Design, use, and performance*, Real-Time Systems Symposium, 1993. pp. 106–111.

[15] Srinivasan S, Stewart D B, 2000, *High speed hardware-assisted real-time interprocess communication for embedded microcontrollers*, 21st IEEE Real-Time Systems Symposium, pp. 269–279.

[16] Saglam B E, Mooney V J III, 2001, *System-on-a-chip processor synchronization support in hardware*, Design, Automation and Test Conference and Exhibition in Europe, 2001., pp. 633–639.

[17] Shalan M, Mooney V J III, 2002, *Hardware support for real-time embedded multiprocessor system-on-a-chip memory management*, Tenth International Symposium on Hardware/Software Codesign, pp. 79–84.

[18] Pontremoli M M B, Pereira C E, 1997, *Hardware Support for distributed Real-Time Operating Systems*, Control Eng. Practice, Vol 5, No. 10, pp. 1435–1442.

[19] Shiu P H, Yudong Tan, Mooney V J III, 2001, *A novel parallel deadlock detection algorithm and architecture* Ninth International Symposium on Hardware/Software Codesign, pp. 73–78.

[20] Lindh L, 1991, *Fastchart – a fast time deterministic CPU and hardware based real-time-kernel* Workshop on Real Time Systems, 1991. Euromicro '91, pp. 36–40.

[21] Stanischewski F, 1993, *FASTCHART – Performance, Benefits and Disadvantages of the Architecture* Fifth Euromicro Workshop on Real-Time Systems, pp. 246–250.

[22] Lindh L, 1992, *FASTHARD – A Fast Time Deterministic HARDware Based Real-time Kernel*, Fourth Euromicro workshop on Real-Time Systems, pp. 21–25.

[23] Realfast, 2002, *Realfast Sierra Operating System Data Sheet.*

[24] Nakano T, Komatsudaira Y, Shiomi A, Imai M, 1997, *VLSI implementation of a real-time operating system*, Asia and South Pacific Design Automation Conference, 1997, pp. 679–680.

[25] The TRON Project. See: http://www.sakamura-lab.org/TRON/

[26] Timothy F Oliver, Douglas L. Maskell, 2004, *Accelerating an Embedded RTOS in a SOPC Platform*, Annual Technical Conference of the IEEE Region 10.

[27] ALTERA NIOS CPU Data Sheet, March 2003.

[28] Lampret D, 2004, *Open RISC 1000 Project*, see: http://www.opencores.org/

[29] Rabindra P Kar, 1990, *Implementing the Rhealstone Real-Time Benchmark*, April 1990 issue of Dr. Dobb's Journal.

[30] Reinhold P Weicker, 1984, *Dhrystone: a synthetic systems programming benchmark*, Communications of the ACM Volume 27, Issue 10.

[31] Harbison S P, 1999, *System-level hardware/software trade-offs*, 36th Design Automation Conference, pp. 258–259.

[32] Mohit Sindhwani, 2002, *A Framework for a Portable, Scalable and Extensible Real-Time Operating System*, Nanyang Technological University, Master in Engineering, Year 1 Progress Report.