

Real Time Operating Systems Implemented in Hardware

Jake Swart

School of Information Technology

University of Ottawa

Ottawa, Ontario


Email: jswar070@uottawa.ca

Abstract—This paper describes several state-of-the-art real-time operating systems (RTOS) implemented in hardware that implement task scheduling, synchronization, multiprocessor dispatching, time management and event management. When designers develop RTOS support in hardware different levels of hardware usage are used. Many architects developing RTOS supporting hardware are opting for designs focused around determinism and avoid using Commercial Off The Shelf (COTS) chips in building their RTOS supporting hardware. No matter the level of hardware support that was implemented for RTOS execution, real-time operating systems implemented in hardware provide significant speed-ups over those implemented in software.

I. INTRODUCTION

A real-time operating system is an operating system dedicated to supporting real-time operations and are popularly implemented in embedded systems. Embedded systems that are used today are a complex combination of hardware and software designed to perform a dedicated function. In this report, embedded systems are reviewed for their use of hardware in implementing support for RTOS execution. Real-time operating systems implemented in hardware provide significant speed-ups over those implemented in software. Because of this fact, many designs for implementing RTOS support in hardware have been proposed and are reviewed in this report. Reviewed designs in this paper include the following: system on chip RTOS, RTM unit, HW-RTOS using SMP, ARPA-MT Embedded SMT Processor RTOS Hardware Accelerator and the MERASA Project.

II. MODERN HARDWARE DESIGNS

System on Chip RTOS  In [1] a framework was created to provide developers with the ability to configure a target system with or without specialized hardware for an RTOS. They experiment between three configurations, with increasing levels of hardware usage in implementing an RTOS created using the framework. The first level being a purely software RTOS implementation (ie. no hardware designed to support an RTOS). The second level uses a System-on-Chip Lock Cache (SoCLC). The last level of implementation uses a real-time management unit (RTU) for RTOS support. For the purpose of this report, focus will be emphasized on the increasing levels of hardware implementation and not the reconfiguration framework.

In the first level, the purely software RTOS is implemented using Atlanta RTOS 0.4 for heterogeneous multiprocessor shared-memory RTOS. The architecture used a SoC with minimal hardware support and software semaphores to provide RTOS synchronization. The second level uses the SoCLC, seen in Fig. 1, which provides a set of lock arrays. Each array holds single bits represented by P_i processors within the SoC and the remaining bits contain the locked variable data represented by LV_k . When a processor bit is set within an array, it indicates that it is waiting for access to that variable. Upon release of the lock, all waiting processors are sent an interrupt so that a new processor may acquire the lock. The RTOS also receives notification on what locks have been released for task scheduling purposes.

The RTU is the third configuration with the greatest hardware support seen in Fig. 2. The Interrupt request module (IRQ) provides intelligent interrupt handling, MsgQLib provides message handling, semaphores and temporal scheduling and the RTC provides real-time control. Two bus interfaces are used to access the RTU, which are the generic bus interface (GBI) and the technology dependant bus interface (TDBI). The TDBI allows for the module to be adapted easily to different architectures using buses. Acceleration registers provide a fast means of communication with the RTU. Communication software is used to provide the handshaking between the CPU and RTU.

Each level of hardware implementation was tested using different task synchronization situations, where atomic actions with variables stored in shared memory would be used during execution.

Results show that the RTU that provides the greatest level of hardware support for RTOS operation performed the best with a 50% overall speedup over a pure software solution. SoCLC also had shown good performance with 41% overall speedup over the pure software solution. As the number of tasks increase both speedups become lower, but the RTU still has a 36% overall speedup twice is that of the SoCLC implementation. The reason for the reduction in speedup with increased workload is due to increased synchronization and context switching. Synchronization causes delays between tasks and may even cause a task to switch from running to waiting. The switch is a context switch in which all the processor registers are pushed to the stack, the next task is

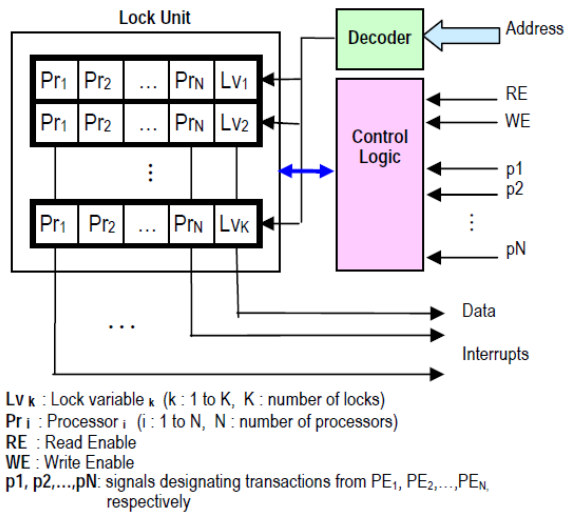


Fig. 1. A diagram of the SoCLC lock unit [1]

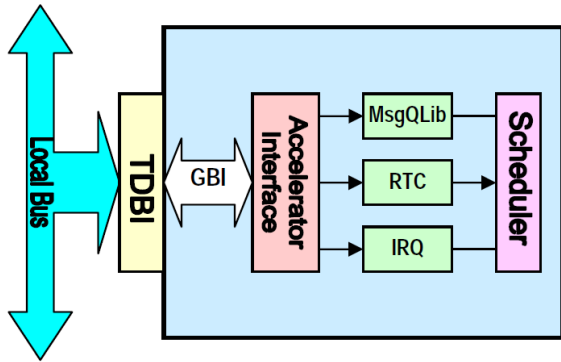


Fig. 2. A simplified diagram of the RTU [1]

selected to run and the new tasks registers are retrieved for execution.

RTM Unit [2] proposes a similar unit to that of the highest level of hardware implemented in [1]. In [2] use a Real-Time Task Manager (RTM) that removes expensive operations performed by a RTOS from the main CPU. By performing the expensive operations in the RTM such as: task management, event management, and scheduling, significant performance bottlenecks are reduced.

The previously listed operations are executed often and rely on each other creating opportunities for parallelism. Since software solutions cannot take advantage of parallel situations, hardware implementations are used (such as the RTM). By doing so [2] reveals that processor utilization (amount of processing power an application is able to use) and response times (the time between an external event occurring to the first response by the RTOS) are reduced significantly.

Communication to the RTM is performed from the CPU using a Memory Mapped Interface (MMI) on the address bus. Within the RTM are a static number of records structured into a binary tree, in this case 64 records were implemented that hold 4 fields that include: status, priority, Event ID and delay.

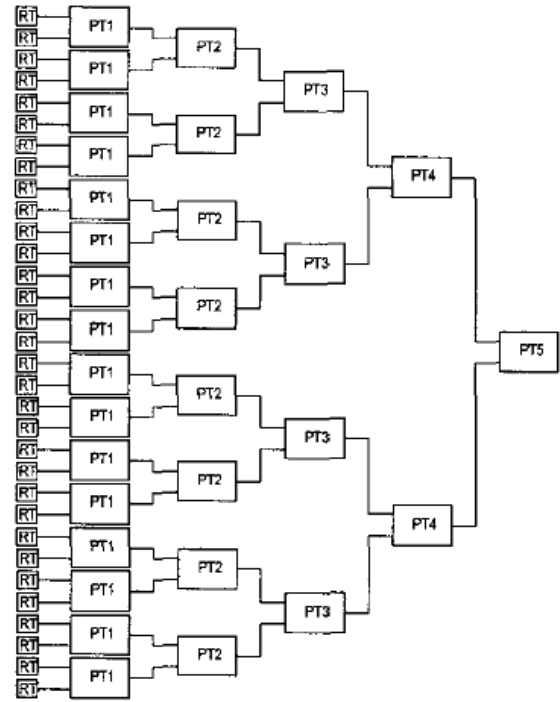


Fig. 3. An example of the RTM tree structure used to determine record priority [2]

Each record is 40 bits long and has been extended to 64-bits for full word MMI purposes. Each records priority and status bits determine whether a task will be scheduled next using static-priority scheduling. Therefore when a task is ready and has the highest priority, it will be scheduled next. For event management, priority is determined differently by the event ID. The event ID has a static priority mapped to it that is used to determine schedule ordering. Time management is the most important factor where the decrementing of each records delay field is equal to the number clock cycle ticks. When a records delay reaches 0, a task is ready to run. Each record countdown delay is implemented using a 16-bit adder with an AND gate comparator, which makes the delay update a fixed constant rather than a linear one.

Fig. 3 shows the high-level diagram of the priority tree. When a record is ready and its priority is the highest, it will be the output of the topmost element of the tree. Because the simplicity of the logic gates within a binary tree, one can see that the RTM can be easily scaled and keep efficiency to hold an increased number of records. Major limitations to this startegy include chip area and higher energy consumption due to increased gate logic.

To test the performance of the RTMs effectiveness at reducing RTOS overhead, two RTOSes were used and measured using MediaBench benchmarks with respect to processing time and worst case response time. Table 4 displays the RTOS system being tested and its feature support. Based on the table, it can be clearly observed that μ C/OS II will perform the best with pre-emption and IPC support. Therefore focus will be on

	μ C/OS II	NOS
Pre-emption	YES	NO
Inter-Process Communication (IPC)	YES	NO
Task Scheduling	O(1) - Bit Vector	O(n) - Sorted Ready Queue
Time Management	O(n) - Unsorted Queue	O(n) - UNIX Callout Queue
Event Management	O(1) - Bit Vector	N/A

Fig. 4. Basic feature comparison of realistic RTOSes [2]

the results of the μ C/OS II RTOS.

Results show that the RTM was able to reduce processing time and worst case response time by 90% and 81% respective for μ C/OS II RTOS. The reduction in processing time is primarily due to time management of task release times becoming constant rather than linear number of cycles by taking advantage of the tree elements within the RTM. The bit vectors used in task scheduling and event management also greatly reduce the processing time to a fixed constant. The worst case response time reduction is primarily due to pre-emption used with the RTM and as workload increases, the response time stays linear.

HW-RTOS using SMP In [3] implement a RTOS in hardware for symmetric multiprocessors (SMP) by using a dedicated units for task scheduling and communication, refer to Fig. 5. The hardware support for an RTOS uses two ARM processors connected to a shared bus and shared RTOS unit. Communication with the hardware RTOS unit, used for task scheduling, is done through dedicated ports that have send and receive buffers. The hardware RTOS unit has access to a lock unit containing memory mapped test-and-set semaphores in hardware for shared variable synchronization between processes.

Within the hardware RTOS unit are two separate hardware schedulers, one for each processor, and task communication buffers between them. The communication buffers provide a means of inter-process communication using the previously mentioned semaphores. Scheduling is done using round robin. Tasks can be pre-empted on time slice expiration or become blocked, which causes a context switch in the corresponding ARM processor. When a block occurs, the hardware scheduler within the RTOS unit moves the task to a wait_port_list and the task information is moved to shared memory for later execution. When a pre-emption occurs, a signal to the ARM processor is generated and the next task is chosen by the scheduler. Selection of a task is based on the wait_port_list and a schedulable task list before it is sent out on the next_task_port.

[3] used eCos as the RTOS software in combination with two media related application benchmarks. Results showed that the SMP architecture combined with the hardware RTOS unit provides higher throughput in comparison to a purely software solution. Most of the overhead created by software is removed using the hardware RTOS using during context switching (10,000 versus 947 cycles).

ARPA-MT Embedded SMT Processor RTOS Hardware Accelerator In [4] have developed a similar implementation to that of [3], where the hardware support for real-time

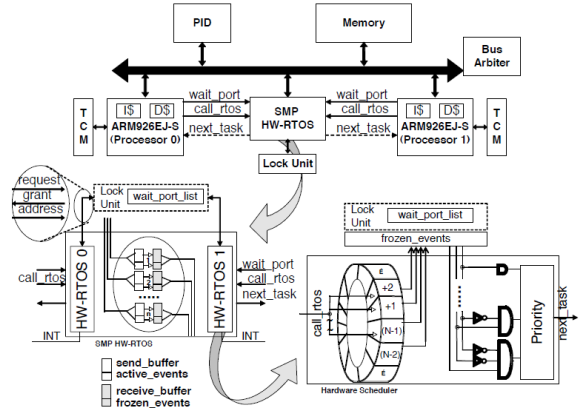


Fig. 5. Detailed implementation of RTOS support hardware using SMP [3]

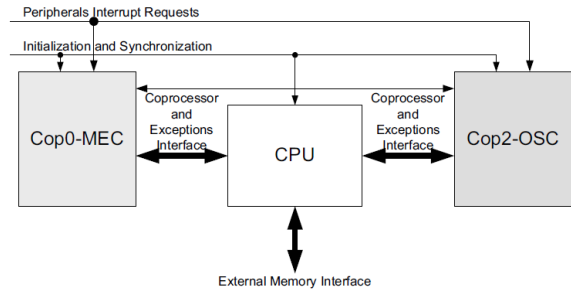


Fig. 6. ARPA-MT tri-processor design [4]

operating systems has been implemented in a separate module. The major difference in the case of the Advanced Real-time Processor Architecture - MultiThreaded (ARPA-MT) is that it is its own separate processor that handles all real-time task management. The ARPA-MT focuses on providing specialized, time predictable, power efficient hard real-time systems using an SMT based implementation. The design of the ARPA-MT is focused towards worst-case execution time (WCET) determinism, task scheduling and resource assignment to tasks.

Fig. 6 displays the arrangement of a three processor SMT implementation based off the MIPS32 architecture. The central CPU uses an integer instruction set with the five traditional stages of pipelining (IF, ID, EX, M, WB). Co-processor-0 handles memory management, exception processing and interrupt handling, while co-processor-2 handles RTOS hardware support with a high resolution real-time clock. Focus will be on the CPU and co-processor-2 since all support related to real-time operation is located here.

Several techniques are used to increase the determinism of the execution time. First, by keeping the pipelining of instructions simplified for each task, the ARPA-MT avoids complex techniques found in superscalar processors to reduce pipeline stalls. Second, by executing multiple simultaneous tasks with interleaving instructions and fine granular time sharing, the ARPA-MT reduces processor stalls due to data or control hazards. Lastly, by using a separate co-processor dedicated to RTOS applications, the ARPA-MT reduces RTOS overhead and increases CPU availability. All of the detailed

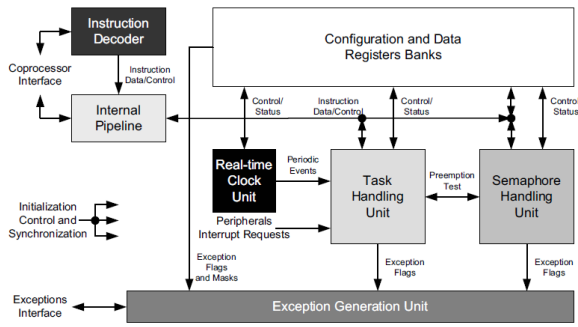


Fig. 7. Detailed implementation diagram of co-processor-2 (or Cop2-OSC seen in Fig 6) [4]

techniques are used in unison to provide better determinism of WCET for task execution within the ARPT-MT.

Fig. 7 displays the architectural design of co-processor-2. Task management is performed by the task handling unit, which includes: specialized instruction implementation, scheduling, dispatching, interrupt management, time constraint management and table storage for task control blocks (TCB). An applications timing constraints are transparent to co-processor-2, which allows it to schedule tasks in real-time. Task synchronization of shared resources is accomplished through the use of the semaphore handling unit that contains binary semaphores.

An important aspect of the ARPA-MT design is the removed block and sleeping states. These are removed because the stack resource policy (SRP) has been implemented here. The SRP will only allow a process to pre-empt another if it can obtain all resources required before execution rather than block or sleep during execution. The SRP technique thus leads to a reduction in context switching within co-processor-2.

There are three classifications of tasks within the ARPA-MT, each with their own scheduling technique being implemented. Below is the list of their classification type and scheduling scheme in descending order of criticality based priority:

- Non real-time: First Come First Serve (FCFS)
- Soft real-time: Rate Monotonic policy (RM)
- Hard real-time: Earliest Deadline First (EDF)

In the case of EDF, as a task comes closer to its deadline, its priority is raised to ensure the deadline is not missed.

Tasks in the ARPA-MT can be event or time triggered. Focusing on the real-time clock unit within Fig. 7, it can be observed that the periodic triggering of timing events are controlled here using a high resolution clock. All external events that are aperiodic are generated in the form of interrupt requests to the task handling unit.

Using a custom API called OReK, the ARPA-MT was tested using a purely software solution and one that uses the co-processor-2. Both implementations were measured on the basis of WCET and it was found that the ARPA-MT performs better overall than a purely software solution. The overall gain was achieved by the removal of the software overhead.

The MERASA Project The Multi-Core Execution

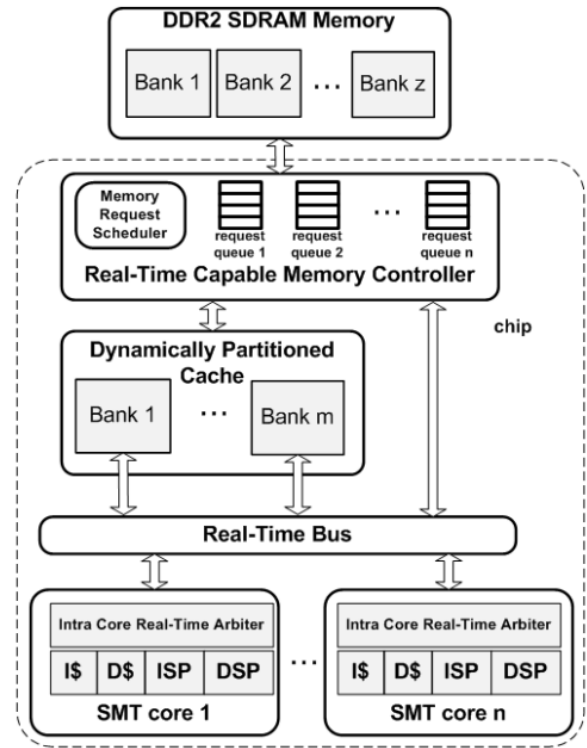


Fig. 8. High level design of the MERASA chip [6]

of Hard Real-Time Applications Supporting Analysability (MERASA) processor was part of a recently completed research project financed and used by partners such as Honeywell International [5], [6]. The focus of the MERASA project was to minimize the gap between the real and computed value for WCET, as well as provide developers with tools for WCET analysis. Using SMT, the MERASA processor provides tight WCET analysis.

The MERASA chip has several different implementations, but for the purpose of this report, only the completed FPGA based implementation is considered. The MERASA chip implemented on an FPGA features a four SMT cores with four threads per core (16 threads in total). Fig. 8 displays the multi-core architecture of the MERASA processor. Communication with each core is done using a real-time bus with an arbiter for each SMT core. The threads are split up within each core to execute two classifications of tasks, which are listed below in order of descending criticality:

- 3 non-hard real-time task threads (NHRTT)
- 1 hard real-time task thread (HRTT)

The MERASA processor uses a two-level scheduler that manages threads over the multiple SMT cores. Scheduling is performed by using the information stored in 256 byte thread control blocks. Each control block contains a pointer to the pervious and next thread control block to create a double linked list structure. There are two doubly linked lists, one for NHRTTs and another for HRTT, each is referred to by the scheduler to determine execution order.

The first level scheduler seen in Fig. 9 uses fixed priority to

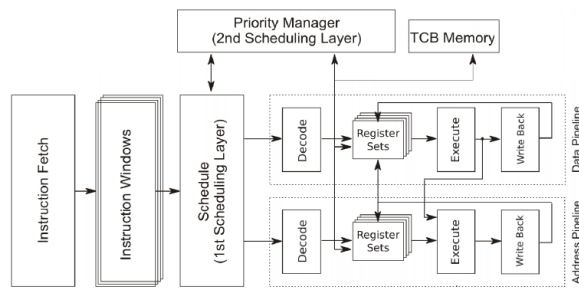


Fig. 9. MERASA core design diagram [7]

determine pipelined execution order of HRTTs and NHRTTs. The second level scheduler provides register and program counter initialization logic for the thread control blocks to begin execution on a core. Note that both scheduler levels are implemented in hardware.

The MERASA core uses an atomic read-write instruction in hardware for synchronization of shared variables between threads. The synchronization primitives are stored within the real-time capable memory controller in Fig. 8. Locking access to the bus is reduced by using an atomic read-write rather than using a separate read, then write to the bus. In this way, WCET is reduced and becomes more predictable. Other synchronization primitives are available in software. The MERASA chip was trialed to determine if it could determine the WCET of scheduled tasks and schedule them in an effective manner to meet deadlines. It was found with a WCET analysis tool that the MERASA chip ensures that HRTTs meet their deadlines while also provided service to NHRTTs.

III. CHARACTERISTICS OF RTOS SUPPORT HARDWARE

There have been several characteristics seen throughout the review of different RTOS support implementations in hardware. The main characteristics are highlighted in this section.

Hardware Overhead Hardware overhead will be defined as the amount of logic required to implement a particular RTOS in hardware. Throughout this paper several implementations have been observed with a wide variety of hardware usage for RTOS support. From the simplistic implementation of logic gates [2], to dedicated real-time management units [1], [3], and finally with the most complex being processor based implementations [4], [5].

When designing hardware support for RTOSes numerous factors need to be taken into consideration such as chip space, adaptability, energy consumption, logic delays and implementation costs. From what was reviewed, the most successful implementations seem to use dedicated real-time management units and processors. The successful implementations require more chip space, energy and logic to be implemented. In the case of the processor based implementations, these are less adaptable to different architectures. Therefore, chip designers need to explore the trade-offs in the hardware design space when implementing hardware support for RTOS.

Determinism and WCET A common theme among the papers was the concepts of determinism and WCET [2], [5], [6]. Chip designers constantly work towards increasing the determinism of their architectural designs in order to better predict WCET. The more accurately WCET can be predicted for a task, application, thread etc, the better the hardware scheduler can determine its scheduling priority. In RTOS, the scheduling of a task to meet its deadline is paramount. Therefore, increasing determinism in design should be an architects primary goal in hardware based RTOS support.

Embedded vs. COTS Solutions All of the papers gathered here have used embedded solutions in order to support RTOS execution. For the purpose of this report, a COTS solution is one that is made by chipset manufacturers such as AMD or Intel. The use of COTS is actually explicitly avoided for two reasons [6]. First, power consumption is much higher and cannot be optimized for the situation it is being implemented in. Second, is that COTS hardware does not have deterministic behaviour. Therefore, without guaranteed deterministic behaviour, the use of COTS processors should be avoided in implementing hardware support for RTOS.

IV. CONCLUSION

Throughout this paper many RTOS implemented with different levels of hardware support have been reviewed, with used within modern multiprocessing architectures. It was found that architects developing RTOS supporting hardware should consider the overhead costs associated to the amount of hardware being used and explore the trade-offs. Architects should also be using designs focused around determinism and avoid COTS chips in building their RTOS supporting hardware. No matter the level of hardware support implemented for RTOS execution, designers will see improved results over purely software based solutions.

REFERENCES

- [1] J. Lee, I. Mooney, V.J. Daleby, K. Ingstrom, T. Klevin, and L. Lindh, "A comparison of the hardware RTOS with a hardware/software RTOS," in *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*, 2003, pp. 683 – 688.
- [2] P. Kohout, B. Ganesh, and B. Jacob, "Hardware support for real-time operating systems," in *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, 2003, pp. 45 – 51.
- [3] A. Nacul, F. Regazzoni, and M. Laiolo, "Hardware scheduling support in smp architectures," in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, 2007, pp. 1 –6.
- [4] A. S. R. Oliveira, L. Almeida, and A. B. Ferrari, "The ARPA-MT embedded smt processor and its RTOS hardware accelerator," *Industrial Electronics, IEEE Transactions on*, 2009.
- [5] J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaiff, C. Rochange, H. Casse and, P. Sainrat, and T. Ungerer, "RTOS support for parallel execution of hard real-time applications on the merasa multi-core processor," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, May 2010, pp. 193 –201.
- [6] F. J. Cazorla. (2010) Fpga prototype of the basic single-core merasa processor. [Online]. Available: http://www.merasa.org/dissemination/02_architecture.pdf
- [7] P. D. T. Ungerer. (2009) Merasa multicore architecture. [Online]. Available: http://www.merasa.org/downloads/Deliverable_5_1.pdf