IMPROVING THE PERFORMANCE OF RTOS USING MULTIPLE REGISTER FILES ARCHITECTURE

Jong-Woong Kim, Soo-Hyun Kwon, Kab-Su Han and Jeong-Hoon Cho
EEC, Kyung-pook National University, Daegu, Republic of Korea

Keywords: Task Context Switch, Real-Time OS, Embedded System, Register file (RF).

Abstract: In recent years, real-time operating systems (RTOS) have been becoming more and more important in embedded systems because of increasing the number of task with complex functions and the need of faster response time. Faster response time is strongly related to the task context switch time and especially task context switch time is the most important factor to determine the performance of RTOS. Most embedded systems are suffering from processing it. In this paper, we present the technique to improve the performance of RTOS by reducing the task context switch overhead. To achieve this goal, we suggest multiple register files architecture and a task to register file mapping algorithm based on rate monotonic (RM) scheduling algorithm for efficiently using our new architecture. Also we show the experimental results to improving our technique using ATmega103 implementation in FPGA. As a result we can decrease the task context switch overhead up to 23% depend on the number of register files even though there are some area overheads by increasing the number of register files.

1 INTRODUCTION

Recently, as embedded systems have extended their application from traditional manufactures, logistics, service business to high-tech aviation, space, military, multimedia communication, and next generation energy business, they have to support multiple, real-time and multimedia processing capabilities and the wire/wireless network service. These trends result in increasing the complexity of embedded system and embedded applications as well and also the usage of real-time operating system (RTOS). RTOS is the an operating system intended for real-time applications and serves application requests nearly real-time based on the prioiry. It is desighed to minimize critical sections of system code so that it can process application requests within a certain amount of time. Thesedays RTOS is widely used in the cellular phone communication equipment, industry facilities and vehicle and so on, and there are some popular RTOS used in each relevant field repectively like IOS, REX, L4 and OSEK. RTOS provides task scheduling, resource management, synchronization, communication and precise timing so it can cover various applications (John and Rajkumar, 2004). However there exists some overheads such as task scheduling, time management, and event management which cause the performance degradation of RTOS like difficult prediction or late response time. Therefore it is important to reduce these overheads to guarantee the performance of RTOS.

Task context switching refers to the computing process of storing the present state and restoring the previous state of CPU so that CPU can resume from the same point at a later time. It can make multiple processes possible to share a single CPU. Though task context switching is an essential feature of a multitasking embedded system, it causes unavoidable system overhead at the same time (Philip, 1996) (Hassan, 2000). In general, the overhead of context switching is about 10~20% in total running time (Dan, 2007).

In this paper we suggest the technique for reducing the overhead of task context switching that is the most critical part of RTOS. Our technique organizes the multiple register files (MRF) architecture and a mapping algorithm between tasks and register files (RF) based on the rate monotonic (RM) scheduling algorithm(Stewart and Barr, 2002). Our technique makes it possible to reduce the task context switching overhead so that it can gurantee faster reponser time.

This paper is organized as follows. In section 2, we present the previous and related works and describe our MRF architecture and scheduling for fast context switching in section 3. In section 4 we show the experimental results about the context switching improvement using our approach. Finally we make a summary of our work and conclusions and talk about our future work in section 5.

2 RELATE WORKS

In the past many researches on RTOS have focused on the scheduling mechanisms but most of them did not pay attention to the task context switching. But as the importance of task context switching is getting bigger, there are more researches aiming at reducing task context switching overhead. We can divide them into two parts; one is for minimizing the register movements in each context switching and the other is minimizing the frequency of context switching. First of all, (Snyder, Whalley and Baker, 1995) and (Zhou and Petrov, 2006) suggested an architecture and compiler techniques to reduce the register movements in each context switching. Their ideas were based on the difference of the number of used registers of each task during the execution time. So they prepared the table containing the context switching points influences the table size, performance and response time, which needs a precise task scheduling in real-time system. On the other hand for reducing the number of context switching more registers are usually required. (Alverson, Callahan, Cummings, koblenz, Porterfield and Smith, 1990), (Adiletta, Rosenbluth, Bernstein, Wolrich and Wilkinson, 2002) and (Kongetira, Aingaran and Olukotun, 2005) suggested an architecture where the hardware threads have their own RFs and (Nuth and Dally, 1995) proposed a hardware which has a large fully-associative shared RF set. Although both strategies are suitable for high-end multithreaded processors, a large number of register for each thread causes considerable drawbacks of embedded processors such as the higher manufacturing cost and more power consumption.

In this paper, our approach reduces register movements when the task context switching occurs so we can achieve faster response time.

3 OUR APPROACH

3.1 Architecture

During task context switching the task context of general-purpose registers, program counter (PC) and stack pointer of the present task are stored and the context of the next running task are restored. As we analyze the task context switch process of general RTOS, it spend much time storing data of general purpose register and restoring. Its overhead time is given by:

$$(T_{str} \times N_{str}) \times (N_{Greg} + N_{Sreg}) \qquad (1)$$

Where N_{Greg} refers to the number of general-purpose registers, N_{Sreg} means the number of status registers, N_{str} is the number of store operations. We assume that there is only one instruction for store operation, T_{str} means required time to perform single store operation (Hassan, 2000), (Dan, 2007). We can calculate store time of context switch using equation 1 and also can calculate restore time

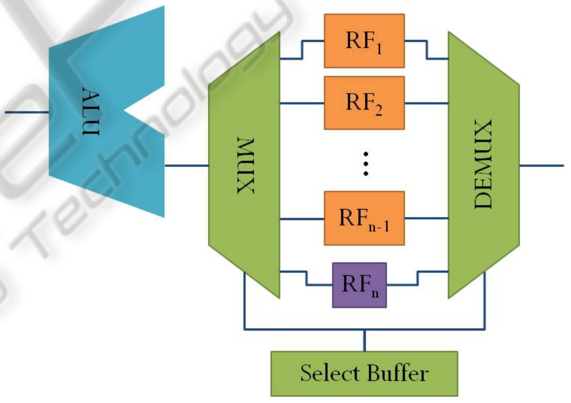

Figure 1: Multiple register files architecture

We propose the MRF architecture by adding more RFs to minimize the context switch overhead, especially focus on reducing time of store and restore. Figure 1 shows our MRF architecture. The number of RFs is flexible so that we can change it based on the number of tasks in embedded applications. But there are constraints of area we will explain about this in section 4.4. To select a RF we add MUX, DEMUX and the select buffer which can make selecting a RF easy by just writing the identifier of RFs in the buffer. And we add a special RF for interrupt service routines (ISR). For example when ISR is occurred regularly by timer in uC/OS-II(Jean, 1998), it calls various functions like OSTimeTick(), OSIntExit(), etc. Because some data can be lost during ISR, we need this special RF to store and restore current state. The result of some experiments we found that the size of this special RF is much smaller than other RFs and you can see it in figure 1. We reserve nth RF, RF_n for ISR.

3.2 Task to RF Mapping Algorithm

There are two kinds of context switching in our MRF architecture; one is in which the number of tasks is less than or equal to the number of RFs and the other is in which the number of tasks is more than the number of register files.

If the number of tasks of applications is less than or equal to the number of register files, each task can have its own RF. It means whenever task context switch is occurred there is no store and restore process of register. Therefore we can change equation 1 for calculating new task context switch overhead time of this ideal case:

$$(T_{str} \times N_{str}) \times N_{Sreg} + N_{ch} \tag{2}$$

Where N_{ch} is the number of the select instruction of select buffer. Instead of store and restore of the whole registers except status registers, only writing the identifier of RFs to the select buffer, the task context switch is done. For example think about a processor with 32 general-purpose registers. When task context switch occurred, there are 32 PUSH and 32 POP instructions to store and restore context of registers. But our MRF architecture just needs 4 instructions for task context switch. Whenever the task context switch is occurred, we can reduce 60 instructions.

And the number of tasks in applications is more than the number of RFs, it's not possible to one-to-one matching between tasks and RFs anymore. Some tasks can have their own RFs but some have to share RFs. To minimize the number of task context switch we use priorities of the RM scheduling algorithm to select 1:1 mapping tasks. RM scheduling algorithm is one of the static-priority scheduling algorithm, the static priorities are assigned on the basis of the period of the job. A task with shorter period will have a higher job priority. Based on this feature of RM scheduling algorithm we assign the priority of each task depending on their period, and then we assign a RF to a task with high priority. It can guarantee less number of task context switching.

Task to RF mapping algorithm

```
input: task set with m tasks, RFset with n-1 RFs
output : mapping result between tasks and RFs
1: assign priorities to all tasks using RM scheduling
2: sort task set by their priorities in increasing order
3: while ( |RF| > 1 ) do
4:     remove task from the first of task set
5:     remove RF from RF set
6:     assign task to RF
7: end while
8: if ( task set ≠ {∅})
9:     assign all left tasks in task set to RF_{n-1}
10: end if
```

Task to RF mapping algorithm shows the mapping sequence. The inputs to this algorithm are a task set containing m tasks and a RF set with n-1 RFs except the special RF(RF_n). Based on RM scheduling algorithm we assign a priority to each task in line 1. Here we assume the smaller number means the higher priority. And then we order m tasks in *task set* by their priorities in increasing order in line 2. Until the number of RF in *RF set* is equal to or less than *1*, we assign a RF to a task one on one (line 3-7). After that if there are tasks which do not have their own RF., they are mapped to the last RF_{n-1} and they share it.

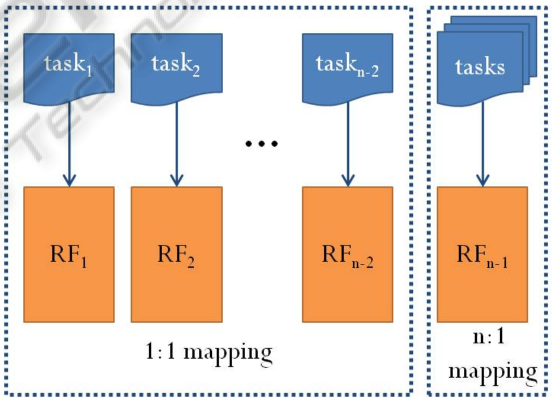

Figure 2: the example of mapping.

Figure 2 shows the result of the m tasks to n RFs mapping when m is bigger than n. The last RF_n is the special RF for ISR so we don't use it for tasks. After mapping is done n-2 tasks use their RFs and the m-(n-2) tasks share one RF for task context switch. In other words, n-2 RFs are assigned to n-2 tasks with high priorities and the remainder use the single RF.

$$(T_{str} \times N_{str}) \times N_{Sreg} + N_{ch} + T_{chk} \tag{3}$$

$$(T_{str} \times N_{str}) \times (N_{Sreg} + N_{Greg}) + N_{ch} + T_{chk} \tag{4}$$

where T_{chk} is the time to check whether the next processing task has its own RF or shares a RF. Equation 3 calculates task context switch time for the 1:1 mapping task and equation 4 is for tasks sharing one RF. Equation 3 has a checking time, T_{chk} by comparing with equation 2 but it is small enough to ignore because it can be executed by simple conditional instruction. To maximize the effect of equation 3 we use RM scheduling in task to RF mapping and can reduce the amount of time to store and restore general registers data than previous architecture.

4 RESULTS

4.1 Experimental Setup

We implemented ATmega103 architecture in Virtex-4 FPGA board using VHDL. We added RFs in our implementation of the ATmega103 - we change the number of RFs from 2 to 6. For convenience we call ATmega103 in FPGA with single RF "Original", our multiple register files architecture "MRF". We run uC/OS-II with 5 tasks and 1 idle task on both architectures. uC/OS-II executes each task periodically with its unique period. When there are no tasks to execute, uC/OS-II executes an idle task automatically which counts the number of the execution of this task. We set one time tick as 10ms and evaluate performance in each case by changing the number of tasks and RFs.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\sqrt[n]{2} - 1 \qquad (5)$$

Where U is the usage factor of CPU, C is the execution time of each task, T is the period of task and n is the total number of task. We assigned execution times and periods to them by equation 5(Stewart and Barr, 2002). In RM scheduling, to guarantee no starvation we have to satisfy the equation 5. According to period of a task, each task has its own priority. So T_1 with the shortest period has the highest priority and it activates every 1 tick(10ms) and executes a loop 7000 times. Table shows the features of tasks we use. We run this application for 4000 tick.

Table 1: Task scheduling.

task	Period (tick)	Execution time (# of loop)	Priority
T1	1	7000	0
T2	3	7800	1
T3	5	15000	2
T4	14	18900	3
T5	57	21650	4

4.2 Without Starvation

uC/OS-II executes 6 tasks including one idle task by changing the number of RFs from 1 to 6. To show how fast our approach is we measure the total execution time of idle task and the longer execution count of idle task means reduction of execution time of tasks. Figure 3 shows that idle task has been running about 23% more than original method and it means that an overall execution time of tasks is reduced about 23%. But there is no change except MRF on uC/OS-II so it relate to reduction of context switch.

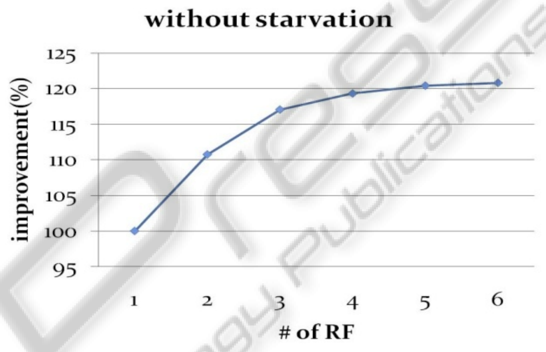

Figure 3: The execution time of idle task without starvation.

4.3 With Starvation

We change the execution time of T4 from 18900 to 19100 so that the application can't satisfy the equation (5) anymore and it results in the starvation during program execution.

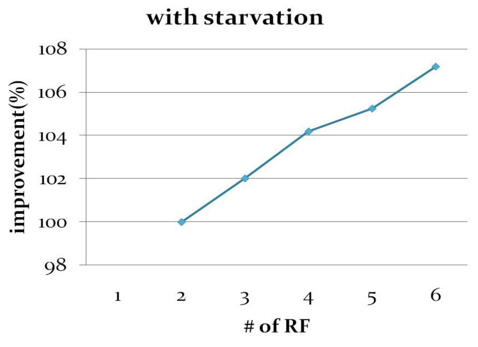

Figure 4: The execution time of idle task with starvation.

As shown Figure 5, when there is one RF, uC/OS-II can't execute the tasks properly because of the starvation. But we change the number of RFs, uC/OS-II can execute them well. Because reduction of execute time make them runnable. More we add RFs, we get more count of idle task. It means that the context switch overhead is reduced when the number of RFs is approaching the number of task. This experiment shows that our MRF makes possible to properly execute6 applications with starvations.

4.4 Area Overhead

We compare the additional area of RFs. The proportion of single RF in VHDL-original excluding the program and data memory is about 21%. We compared the area proportions which come from HDL synthesis tools, while changed the number of RFs from 2 to 6 in Virtex-4 FPGA. Figure 5 shows the result.

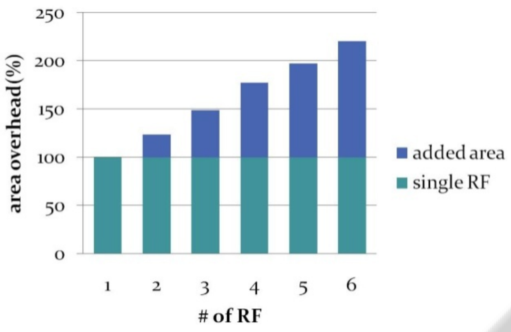

Figure 5: Comparison about area.

Figure 5 shows that the total area including MUX, DEMUX and a select buffer, a special RF increases proportionally to the number of RF in FPGA. When the number of RF is 5 the total area is almost double than original architecture in FPGA. But in this experiment we didn't implement the memories in ATmega103 and the CPU area including RF occupies very small amount in the whole chip set so that the increasing area by adding RFs is to be negligible.

5 CONCLUSIONS

In this paper, we presented MRF architecture and a task to RF mapping algorithm based on RM scheduling for fast task context switch. Our MRF architecture makes faster response time possible by reducing store and restore task context. We also propose the mapping algorithm between tasks and RFs which guarantees that the most frequently executing tasks can have their own RFs. We implemented ATmega103 by using VHDL and FPGA for experiment and modified uC/OS-II to run tasks on MRF architecture. As we expected the total area increased linearly by adding more RFs but we can reduce task context switching overheads by 23%, shown as figure 3. And we can run the tasks properly when starvation is occurred in our MRF architecture.

We just assume static-priority scheduling algorithm on MRF architecture, in the future, we have to research other scheduling algorithms on the MRF like EDF. Also, we need to research another evaluation method, e.g. count instructions, count time ticks. Finally we have to compare the result among other solutions to ensure.

REFERENCES

John A. Stankovic. R. Rajkumar, 2004. The book, Kluwer Academic Publishers. "Real-Time Operating System," Real-Time Systems.

Philip A. Laplante, 1996. The book, Real-Time systems Design and Analysis: *An Engineer's Handbook, Second Edition*.

Hassan Gomaa, 2000. Designing Concurrent, Distributed, and Real-Time Applications with UML.

Dan Tsafrir, 2007. "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)," *Experimental computer science on Experimental computer science*.

David B. Stewart and Michael Barr, 2002. Paper. *Introduction to Rate Monotonic Scheduling*, Embedded Systems Programming.

J. S. Snyder, D. B. Whalley, and T. P. Baker, 1995. Paper. *Fast context swtiches: Compiler and architectural support for preemptive scheduling*. Microprocessors and Microsystems.

X. Zhou and P. Petrov, 2006. Paper. *Rapid and low-cost context-switch through embedded processor customization for real-time and control applications*. Proceedings of the 43rd annual Conference on Design Automation.

R. Alverson, D. Callahan, D. Cummings, B. koblenz, A. Porterfield, and B. Smith, 1990. Paper. *The Tera computer system*. Proceedings of the 1990 International Conference on Supercomputing,

M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. 2002. Paper. *The next generation of intel ixp network processors*. Intel Technology Journal.

P. Kongetira, K. Aingaran, and K. Olukotun, 2005. Paper. *Niagara: A 32-way multithreaded sparc processor*. Micro, IEEE.

P. R. Nuth and W. J. Dally, 1995. Paper. *The named-state register file: Implementation and performance*. IN Proc. 1st Intl Symp. on High-Performance Computer Architecture HPCA.

Jean J Labrosse, 1998. *The book*., R&D Books [M]. uC/OS-II: The Real-Time Kernel.