Vol.2019-EMB-51 No.7
2019/6/21

A Study of Real-Time Extension for RISC-V Processors

Aye Myat Mon[1,a)] Thiem Van Chu[1,b)] Kiyofumi Tanaka[1,c)]

Abstract: In development of applications for real-time embedded systems, RTOS functions (system calls) are often used for easily designing the target system. However, behavior of RTOS is one of factors of unpredicted execution times, which affects the real-time processing. In this research, we investigate a possibility of making the execution times of RTOS constant or at least limiting them to relatively low upper bounds by introducing hardware functions. The proposed hardware is implemented as RTOS Management Unit (RMU) with a RISC-V processor in an FPGA .

1. Introduction

In real-time systems, task execution times should be fixed for predictability and schedulability. However, actual execution times tend to vary since they are influenced by various factors such as unpredicted cache behavior and other disturbance. This makes it difficult to estimate tasks' execution times exactly.

Use of RTOS (Real-Time Operating System) functions makes development of real-time applications easier. However, behavior of RTOS is one of factors of unpredicted execution times. While tasks' functions differ from application to application, functions of RTOS are fixed, which means there is a possibility of making the execution times of RTOS constant or limiting them to some upper bound.

This research aims to provide RTOS which exhibits constant execution times by introducing special hardware mechanisms, RMU (RTOS Management Unit), and custom instructions for manipulating them. This makes it easy to estimate execution times of tasks that use RTOS services. As a base processor architecture, we adopt RISC-V [1] instruction set which is a relatively new ISA.

Although various RTOSs have been produced so far, almost all of them are supposed to run on conventional processors/controllers. This is the case for those on RISC-V processors, for example, embOS [2]. On the other hand, we focus on hardware mechanisms for RTOS so that RTOS codes spend fixed and constant execution times. There are several hardware solutions for RTOS such as HartOS [3] and Sierra [4] which aim to accelerate RTOS processing. By contrast, we attempt to keep the RTOS's execution times constant with the minimum hardware resources.

Generally, task codes and RTOS codes are executed on the same processors. This means execution of RTOS is affected by behavior of tasks' execution and vice versa. For example, RTOS data may be evicted by tasks' data in cache memory. Therefore, to keep the execution times of RTOS constant, isolation of data storage areas is effective. In our approach, along with the special storage area for RTOS, new instructions are defined for managing the RTOS data efficiently.

2. Implementation of RISC-V Processor

We are implementing an embedded processor based on RISC-V instruction set architecture. 49 instructions in RV64I (base integer instructions), 13 in RV64M (multiply extension), and 6 privileged instructions for exception and interrupts can be executed by the processor.

Figure 1 shows the processor organization which basically follows 5-stage pipeline structure in [5]. Compared to [5], forwarding units are enhanced to solve more data de-

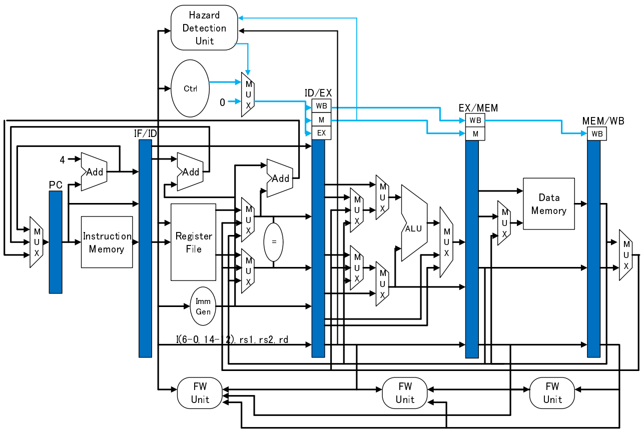

Fig. 1 Pipeline structure.

1 Japan Advanced Institute of Science and Technology
JAIST, Asahidai 1-1, Nomi, Ishikawa 923–1292, Japan
a) ayemyatmon@jaist.ac.jp
b) thiem@jaist.ac.jp
c) kiyofumi@jaist.ac.jp

pendencies.

3. RTOS Management Unit (RMU)

Our RTOS Management Unit (RMU) consists of task control blocks (TCBs), queue data structures. and their management logic. Literature [6] proposed efficient queue management in hardware with shift registers. Instead of using shift registers, our hardware builds various queues (such as ready queue) by manipulating pointer values in TCBs for simplicity.

For constructing a queue structure, each TCB includes next pointer (NEXT), previous pointer (PREV), deadline (DEADLINE). In addition, it takes as input a command (task insertion/deletion), task id, and deadline values of other tasks, and outputs its deadline value.

When a new task is inserted to the queue, a TCB which satisfies the following conditions updates its pointer values.

```
/* Task insertion to a queue */
Input:  TSKID_IN;    /* task id of a new task */
        DEADLINE[N]; /* Deadlines of all tasks */

/* Updating NEXT */
if ( (DEADLINE[TSKID] <= DEADLINE[TSKID_IN]) &&
     (NEXT == -1 or
      DEADLINE[NEXT] > DEADLINE[TSKID_IN]) )
   Write TSKID_IN in NEXT;

/* Updating PREV */
if ( (DEADLINE > DEADLINE of new task) &&
     (PREV == -1 or
      PREV.DEADLINE <= DEADLINE of new task) )
   Write TSKID_IN in PREV;
```

Conditions for task deletion is given as follows.

```
/* Task deletion from a queue */
Input:  TSKID_IN;    /* task id of a deleted task */
        NEXT_IN[N];  /* next pointers of all tasks */
        PREV_IN[N];  /* prev pointers of all tasks */

if (NEXT == TSKID_IN )
   Write NEXT_IN[TSKID_IN] in NEXT;

if (PREV == TSKID_IN )
   Write PREV_IN[TSKID_IN] in PREV;
```

An example of manipulating an EDF-based ready queue is shown in **Fig. 2**. The top figure depicts three tasks (task 5, task 2, and task 3) connected in ascending order of deadline values. "Queue top" data structure consists of two pointers: task id of the head task (NEXT) and that of the tail task (PREV). This data structure is regarded as a TCB with TSKID = -1 and a target of the above conditions' evaluation. Insertion of a new task (task 7) is shown in the bottom figure. Relevant pointer values are updated according to the deadline values.

All information in TCB is stored in dedicated registers. They are memory-mapped, that is, each register is given its own address. TCB information can be referenced by issuing conventional load and store instructions with the addresses. The reason why we select memory mapping techniques rather than implementing special instructions by the RISC-V implementation-dependent extension is that memory mapping makes it possible to easily use the existing compiler tool chain.

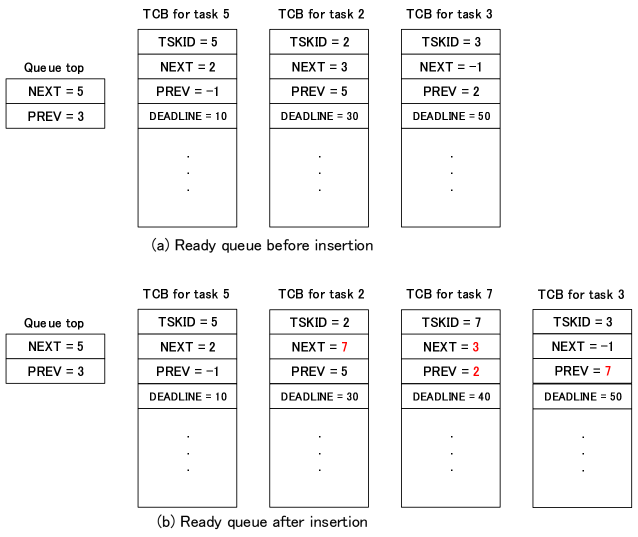

Fig. 2 Structure of ready queue.

In addition to referencing the registers, RMU operations (queue insertion and deletion) are invoked by issuing memory reference instructions with special addresses.

4. Conclusion

In this research, we propose techniques which make the execution times of RTOS constant by introducing hardware functions as RTOS Management Unit (RMU). The proposed hardware is planned to be implemented on an FPGA along with a RISC-V pipelined processor. All the logic circuits are described in VHDL. It is planned to evaluate execution times of RTOS as well as improved efficiency in the use of cache memories from reducing conflicts between RTOS and application data.

Acknowledgments This work was supported partly by JSPS KAKENHI Grant Number JP 19K11873 for designing RMU, and partly by SHIBUYA Science Culture and Sports Foundation for designing a RISC-V processor.

References

[1] `https://riscv.org`
[2] `https://www.segger.com/risc-v/`
[3] A.B.Lange, et al., "HartOS – a Hardware Implemented RTOS for Hard Real-time Applications," Proc. of IFAC/IEEE Intl. Conf. on Programmable Devices and Embedded Systems, Vol 45, No. 7, pp.207–213, 2012.
[4] N. Forsberg, "Analysis of a Hardware-Based Real-Time Kernel," Thesis in MALARDALEN Univ. 2014. `https://www.diva-portal.org/smash/get/diva2:730890/FULLTEXT01.pdf`
[5] D. A. Patterson, J. L. Hennessy, "Computer Organization and Design RISC-V Edition," Morgan Kaufmann, 2017.
[6] Y. Tang, N .W. Bergmann, "A Hardware Scheduler Based on Task Queues for FPGA-Based Embedded Real-Time Systems," IEEE Trans. on Computers, Vol. 64, No. 5, pp.1254–1267, 2015.