A Framework for Automatic Generation of Application-Specific FPGA-based SoC

Tetsuo Miyauchi
School of Information Science
Japan Advanced Institute of Science and Technology
Nomi, Ishikawa 923-1292 Japan
t-miyauc@jaist.ac.jp

Kiyofumi Tanaka
School of Advanced Science and Technology
Japan Advanced Instituted of Science and Technology
Nomi, Ishikawa 923-1292 Japan
kiyofumi@jaist.ac.jp

Abstract— As IoT or CPS devices/systems increase, efficient, cost effective real-time embedded systems are getting important. For providing various highly application-specific systems, we are developing a design environment based on a framework for automatic generation of application-specific FPGA-based SoC. Use of the environment makes it possible for designers to automatically generate a target system design which is highly adapted to the application. Our targets for optimization/customization are multicore processors, real-time operating systems, and acceleration hardware in FPGA. This paper shows the outline of the framework.

I. Introduction

Along with popularization of IoT (Internet of Things) and CPS (Cyber Physical Systems), importance of technologies for real-time embedded systems is increasing. In addition, use of programmable devices such as FPGA (Field Programmable Gate Array) is spreading to construct application-specific systems. In order to achieve high efficiency in system running and cost-performance, we aim at building a development environment which produces hardware/software cooperative SoC (System-on-Chips) systems dedicated to specific real-time applications. The environment consists of adaptation of multicore processors to applications, optimization of real-time operating systems (RTOS), hardware acceleration of system calls, and generation of hardware accelerators for computation kernels in applications with high-level synthesis. For designing highly efficient SoC systems, developers have only to describe application software codes. Then the development environment inputs the software descriptions and automatically outputs the RTL descriptions and RTOS which are adapted to the application.

In this paper, we show the outline of the design environment which we have been developing. In particular, we focus on how to generate application-specific multicore processors and RTOS kernel with dedicated system calls.

II. Outline of the Environment

We aim at providing a design environment which automatically generates SoC designs which are highly adapted to the target applications. Figure 1 depicts our development environment for FPGA-based SoC systems. The inputs are application source codes written in C language, the full set of RTOS codes, and system parameters such as the type of the target FPGA device, the number of cores to be implemented, structure of cache memory, real-time scheduling policy used, and performance requirements. All the inputs are given to each sub module of the environment: multicore RTL generator, RTOS generator, and accelerator generator.

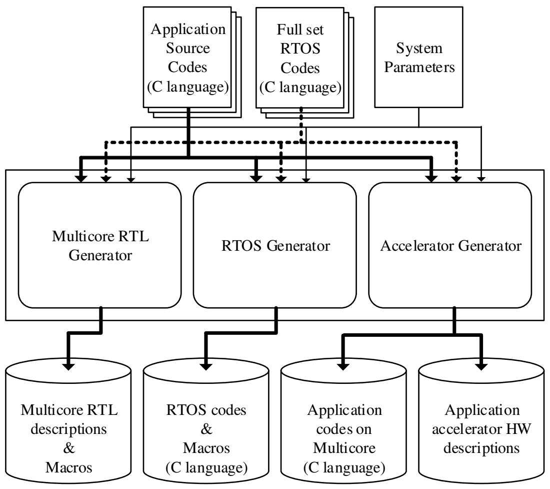

Application Source Codes (C language)
Full set RTOS Codes (C language)
System Parameters
Multicore RTL Generator
RTOS Generator
Accelerator Generator
Multicore RTL descriptions & Macros
RTOS codes & Macros (C language)
Application codes on Multicore (C language)
Application accelerator HW descriptions

Fig. 1. Organization of the development environment.

The multicore RTL generator outputs descriptions of multicore RTL adapted to the application, where the minimum amount of hardware (FPGA) resources necessary to run the application is implemented in each core in order to improve the running speed and reduce the energy dissipation. The details are described in Section III.

The RTOS generator, which is described in Section IV, optimizes the RTOS codes so that only necessary system calls which consist of only in-erasable code fragments are included in the target system and that scheduling algorithm in the scheduler is customized to the application. The purpose is to improve performance in real-time processing.

The accelerator generator extracts the main parts in the application and translates them into RTL descriptions by invoking high-level synthesis tools, as well as outputting the remaining parts as software codes for the multicore processors. It is possible to obtain high responsiveness and mitigate jitters of application tasks with long execution times.

Finally, all the generated descriptions/codes are processed by appropriate compilers or logic synthesis tools to construct the SoC system.

III. Multicore RTL Generator for FPGA

Capacity of FPGA devices has been becoming relatively large so that soft multicore processors can be accommodated in a single FPGA chip using programmable resources. Along with it, use of ASIP (Application-domain Specific Instruction-set Processors) [1] is a promising option for building efficient application-specific embedded systems. As an example, an ASIP development environment, ASIP Meister [2], generates an application specific processor core for given parameters such as register length, a set of instructions, pipeline depth, and so on.

To create an application specific processor, one approach is to build a processor with a new instruction set. Another is to make use of a subset of instructions from an existing instruction set [3]. The latter makes it possible to reuse existing compilers while the former requires a corresponding new compiler. Techniques for selecting instructions in [4] show that the selection according to application programs simplifies the processor's microarchitecture and reduces the amount of hardware resources. However, the target is only a single core processor and does not have cache memory. The analysis of program codes does not take into account dependency between instructions which determines necessity of forwarding paths/dependency detection units. On the other hand, our techniques described later deal with multiple cores and cache memory and try to eliminate forwarding paths if they are unnecessary for the application.

The literature [5] shows a customizable multithreaded architecture for FPGA and the development tools. Forwarding paths are eliminated when the interleaved multithreaded configuration is selected and the number of threads is enough large compared to the pipeline depth, since instructions in different stages are from different threads and therefore they do not suffer from dependency inside a thread. On the other hand, our techniques do not require interleaved multithreaded microarchitecture to remove the forwarding paths. Therefore, our techniques can be applied to a wide range of microarchitecture.

A. *Flow of Multicore Configuration*

The configurator generates multicore RTL descriptions in Verilog HDL. The flow of generating an application-specific multicore design is shown in Figure 2. Each process is described in the following subsections.

A.1. Architecture of the processor

We have developed a multicore processor, of which the core is MIPS architecture [6], written in Verilog HDL. The features of the processor we designed are as follows.

1. *5-stage pipeline*: This processor core has a 5-stage pipeline, which consists of IF (Instruction Fetch), ID (Instruction Decode), EX (Execution), MEM (Memory access) and WB (Write Back to register) stages.

2. *Branch decision*: When a conditional branch instruction is executed, the decision of whether the control proceeds to the following instruction (not-taken) or branches (taken) is made in ID stage.

3. *Delay slot*: A conditional/unconditional branch instruction has one delay slot, so that the following instruction of a branch instruction is always executed regardless of the branch decision.

4. *Forwarding unit*: When an instruction uses the results of the former instructions, the pipeline processing does not have to be stalled as long as the results can be obtained via the pipeline registers even if they have not been written back to general-purpose registers. The forwarding unit detects these cases.

5. *Detection of pipeline stall*: When an instruction uses the results of the former instructions but the forwarding unit cannot supply the results immediately, the pipeline has to be stalled. The pipeline-stall-detection unit decides and controls the pipeline stall.

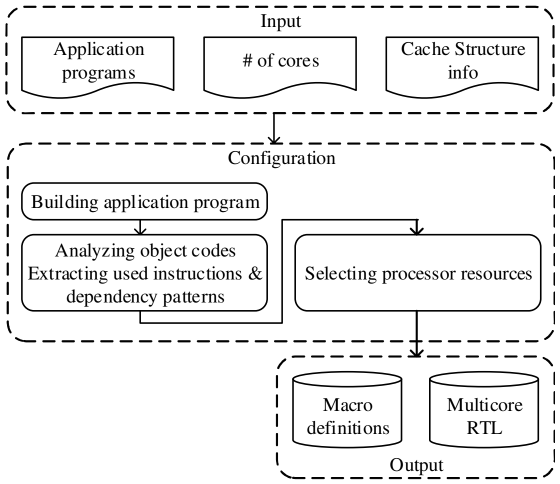

Input
Application programs
# of cores
Cache Structure info
Configuration
Building application program
Analyzing object codes Extracting used instructions & dependency patterns
Selecting processor resources
Macro definitions
Multicore RTL
Output

Fig. 2. Multicore configuration flow.

A multicore processor is constructed with a combination of the cores adapted to the application programs.

The processor can be configured with from one to eight cores. Figure 3 shows the organization of an eight-core processor. The number of cores is given to the development environment (configurator) in advance. Each core has on-chip instruction memory (IMEM) which stores instructions (program codes) and data cache memory (Cache) which stores data, which is multi-instruction, multi-data structure.

When a multicore processor is configured, the configurator outputs Verilog-HDL descriptions for the indicated number of cores and the coretop module (Figure 3) which connects the cores, the instruction memories and the data caches. The configurator builds the connection of each module properly, so the multicore circuit is generated.

A.2. Configurator GUI for input

Figure 4 shows the graphical user interface of the multicore configurator, which accepts configuration settings and application program files, and invokes the automatic generation of the application-specific multicore descriptions. The configurator inputs the number of cores to be implemented, cache structure information (size and associativity), source files' path and library files' path. It has other interfaces for object codes' path, starting

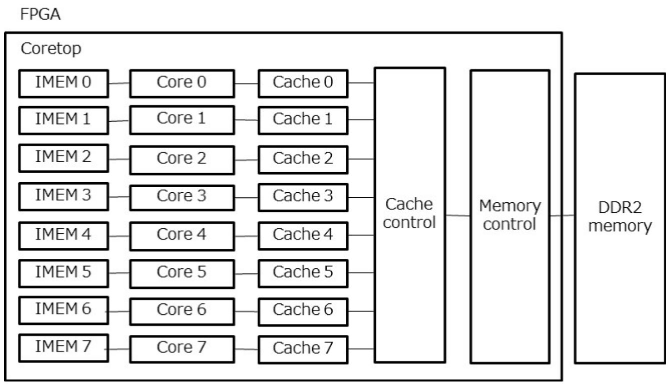
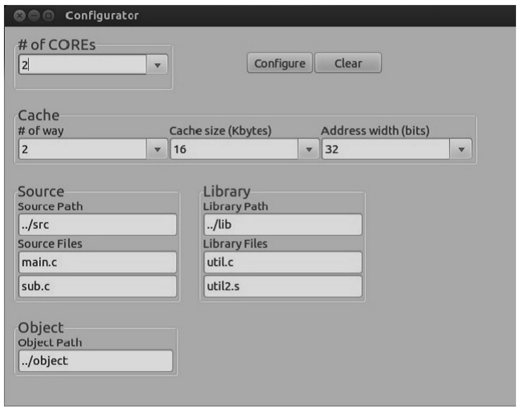


Fig. 3. Multicore processor structure.



# of COREs: 2

Cache
# of way: 2
Cache size (Kbytes): 16
Address width (bits): 32

Source
Source Path: ../src
Source Files: main.c, sub.c

Library
Library Path: ../lib
Library Files: util.c, util2.s

Object
Object Path: ../object

Fig. 4. Configurator GUI.

configuration, and clearing generated objects. (This system runs on Linux.)

A.3. Building application program

The first step of the configuration is to invoke the compiler (`gcc`) with the application source codes in C language or MIPS assembly language, and generate the object codes of the application.

A.4. Analyzing object codes

The object codes of the application are analyzed. First, the object codes are dumped to text information by the disassembler (`objdump` in GNU package). The text file is analyzed and machine instructions which are actually used in the application are extracted.

Then, possibility of forwarding and pipeline stall is checked. The configurator searches the instruction sequences for dependency between instructions, then decides the necessity of each forwarding or pipeline stall unit. If it is found that some forwarding or stall patterns do not occur, the corresponding units are removed from the final implementation. The outline about the correspondence between dependency patterns and forwarding/stall detection units is as follows. The details are found in [7].

Forwarding patterns are classified into five cases in our processor core.

1. *Forwarding ALU result to the next instruction*:
   When the result of ALU calculation is used in the next instruction, the result is forwarded to the next instruction via a pipeline register. An example is as follows.

   ```
   add  $10, $8, $9
   sub  $12, $10, $11
   ```

2. *Forwarding ALU result to the instruction immediately after the next instruction*: When the result of ALU calculation is used in the instruction immediately after the next instruction, the result is forwarded to the instruction. An example is as follows.

   ```
   add  $10, $8, $9
   sub  $14, $13, $12
   lw   $11, 0($10)
   ```

3. *Forwarding ALU result to conditional branch instruction*: When the result of ALU calculation is used in the conditional branch instruction just after the next instruction, forwarding from MEM stage to ID stage is performed. An example is as follows.

   ```
   add $10, $8, $9
   sub $14, $13, $12
   beq $10, $11, label
   ```

4. *Forwarding ALU result to the following store instruction*: When the result of ALU calculation is used in the next store instruction as the stored value, the value in WB stage is forwarded to MEM stage. An example is as follows.

   ```
   add $10, $8, $9
   sw $10, 0($11)
   ```

5. *Forwarding from jal,jalr*: Jump and link instructions (`jal`, `jalr`) write the return address in `$31` register. When the jump target instruction is a jump register instruction (`jr`) with `$31`, the return address being written in `$31` is forwarded to `jr`. An example is as follows.

   ```
       jal label
       nop
   label:
       jr $31
   ```

There are five cases where the pipeline stalls.

1. *Result of load instruction used by the next instruction*: When the result of a load instruction is used by the next instruction, the pipeline has to be stalled. An example is as follows.

   ```
   lw $8 0($9)
   add $9, $8, $10
   ```

2. *Result of ALU/load instruction used by the next conditional branch*: When the result of ALU calculation or the load instruction is used in the next conditional branch instruction, pipeline has to be stalled. An example is as follows.

```
add $10, $8, $9
beq $10, $11, label
```

3. *Result of load instruction used by conditional branch after the next instruction*: When the result of a load instruction is used in a conditional branch instruction after the next instruction, the pipeline has to be stalled for one cycle. An example is as follows.

```
lw $10, 0($9)
nop
beq $10, $11, label
```

4. *Result of ALU used by jr or jalr*: When the result of ALU calculation is used in the next jump register or jump and link register instruction, the pipeline has to be stalled. An example is as follows.

```
add $10, $8, $9
jr  $10
```

5. *Result of load instruction used by jr or jalr after the next instruction*: When the result of a load instruction is used in jr or jalr after the next instruction, the pipeline has to be stalled. An example is as follows.

```
lw $10, 0($9)
nop
jr $10
```

A.5. Selecting processor resources

After the analysis of the application programs is done, hardware resources/components which are necessary for the instructions in the application to be executed can be fixed. In other words, unnecessary resources for the application (for example, multiplexors, adders, divider, etc.) can be excluded from the final implementation.

Figure 5 depicts an example of selecting necessary resources. `M_MUX2` is a multiplexor for extracting a target byte (8 bits) from a 32-bit word read from the memory. This multiplexor is used by load-byte instructions (`lb` and `lbu`). It is not used if any load-byte instructions are not required. If unnecessary, `M_MUX4` and the connections from `M_MUX2` to `M_MUX4` including sign/zero-extension components are not needed either, which leads to removing `M_MUX2`, `M_MUX4`, and the two extension components. Similarly, if the application does not use load-halfword instructions (`lh` and `lhu`), `M_MUX3`, `M_MUX5`, and the corresponding two extension components can be eliminated.

A.6. Output

RTL descriptions of the processor and a macro definition file are the output of the configurator. Selected resources for the application are indicated by the macro definition file. In the macro definition file, each resource is assigned with its symbolic constant as follows.

```
‘define  memmux_2_nouse
‘define  memmux_3_nouse
           ...
           ...
```

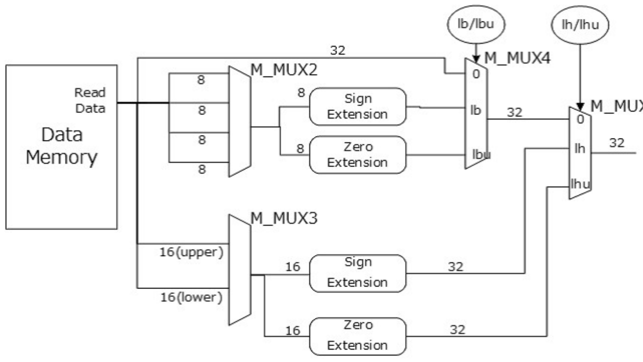


Fig. 5. Selection of multiplexors.

The corresponding processor RTL source files include the macro descriptions as follows.

```
/* MUX2 MEM */
‘ifdef  memmux_2_nouse
    ;    /* M_MUX2 is removed */
‘elsif  memmux_2_mux
    MUX8_4to1  CPU_MUX8 (
        .A (DATA_IN[31:24],   .B (DATA_IN[23:16],
        .C (DATA_IN[15:8],    .D (DATA_IN[7:0],
        .SEL (DATA_ADDR[1:0],  .Z (DATA_IN2_8 ) );
‘else    /* ERROR: Never Reached */
    assign DATA_IN2_8 = 32’hf0200bad;
‘endif
```

Synthesizing the source RTL codes with the macro definition file, proper resources are chosen to build the application-specific processor. For more detailed information, [7] can be referred to.

IV. Optimization of RTOS

Use of RTOS is effective in designing efficient real-time systems since RTOS provides various synchronization/communication capabilities as well as real-time scheduling. However, RTOS is multifunctional to cope with various real-time embedded applications, and therefore it is desired to remove unused functions for execution/space efficiency and cost-performance. Examples of optimization of RTOSs are found in [8, 9], where only functions/components used by the application are included in the implemented RTOS. While these techniques try to prepare fine-grained functions and select appropriate functions according to the application, some of execution/control paths in the functions are still unused. On the other hand, we try to eliminate as many unused code fragments as possible in system calls provided by RTOS.

We have developed an RTOS kernel conforming to μITRON4.0 specification[10]. μITRON4.0 is a specification for embedded operating systems [11]. μITRON4.0 can provide adapted systems where only system calls which are called in the application are linked to the executable binaries. Basically, each system call is supposed to provide a single (primitive) function. However, looking into the specification, two or more options are included in several system calls. For example, in communication mechanisms

using mail-boxes, messages are managed in FIFO order or they can be prioritized according to their fixed priorities. A system call for mail-boxes copes with both management policies which are selected in runtime according to the given attributes. Analyzing the source codes in terms of the attributes can eliminate some paths which are never traced in runtime.

In addition, each system call checks several errors defined in the specification. The checking is done during the first phase of its execution. When an application is fixed in advance, several errors can be regarded as never happening ones, which leads to removing the corresponding checking codes.

While various useful real-time scheduling algorithms have been proposed in theory of real-time processing, μITRON4.0 specification defines only a fixed-priority (static) scheduling. Our environment enables to select various scheduling algorithms according the applications, while keeping application programming interfaces of μITRON4.0[1].

With fine-grained elimination of code fragments and use of flexible scheduling mentioned above, the purposes of our environment are downsizing of codes and improvement of real-time processing. The following subsections describe techniques of code elimination and selection of scheduling algorithms.

A. Adaptation of Functions Based on Attributes

Although each system call in μITRON4.0 is basically unifunctional in a large sense, it includes multiple paths (code fragments) to support several options based on attributes. Here, we show an example of a system call, snd_mbx**, which sends a message to a mail box.**

In μITRON4.0, messages can be exchanged between tasks via mail-box services. Each mail-box holds messages in FIFO order or in prioritized order where priorities are attached to messages. Which order is used depends on the attribute with which the mail-box is associated when it is created. In most applications, the attributes are given statically. Therefore, a part of codes corresponding to unused management order can be removed by analyzing the source codes.

As a result of the analysis, a macro definition file is generated as follows.

```
#define  CHK_MBX_MFIFO
           ...
```

A part of the source file of snd_mbx **for sending a message via a mail-box is as follows.**

```
...
#if defined(CHK_MBX_MPRI) && defined(CHK_MBX_MFIFO)
/* Both FIFO and PRI (Based on  runtime decision) */
if ((mbxcb -> mbxatr & TA_MPRI) != FALSE) /* PRI */
    _kernel_queue_insert_msgpri ( ... );
else {  /* FIFO */
    ....
    ....
}
#else
#ifdef CHK_MBX_MPRI  /* PRI */
_kernel_queue_insert_msgpri ( ... );
#endif /* CHK_MBX_MPRI */

#ifdef CHK_MBX_MFIFO   /* FIFO */
    ....
    ....
#endif /* CHK_MBX_MFIFO */
#endif
...
```

Compiling this source code chooses only necessary code fragments according to the definitions of CHK_MBX_MPRI **and** CHK_MBX_MFIFO**. When only FIFO order is used in the application, the codes corresponding to the management with prioritized order are removed.**

B. Adaptation of Error Check Codes

Our kernel provides seventy system calls in the standard profile defined by μITRON4.0. Each system call includes several error checking codes. For example, a system call, act_tsk**, which activates a task, checks four types of errors,** E_CTX**,** E_ID**,** E_NOEXS**, and** E_QOVER**.** E_CTX **means that the system call is called in a CPU-locking state or in non-task context (e.g., in interrupt handlers).** E_ID **means that the task identifier given through an argument is not within the predefined range.** E_NOEXS **means that a task corresponding to the task identifier has not been registered. The last error,** E_QOVR**, means the count of activation requests for the task exceeds the predefined value. Depending on how, where, and when to call** act_tsk **in the source codes, it is possible to statically decide whether each of these four types of errors can happen or not. If the errors are found not to occur, the corresponding code fragments can be eliminated.**

According to the analysis results, a macro definition file is generated where a symbolic constant is defined if the corresponding error has to be checked in runtime. An example is below.

```
#define  CHK_ACT_TSK_E_CTX
#define  CHK_ACT_TSK_E_ID
          ...
```

A part of the source file of act_tsk **includes macro descriptions as follows.**

```
             ...
#ifdef CHK_ACT_TSK_E_CTX
  if ( _KERNEL_SNS_LOC () || _KERNEL_SNS_CTX () )
      /* cpu lock state or non-task context */
      return E_CTX;
#endif /* CHK_ACT_TSK_E_CTX */

#ifdef CHK_ACT_TSK_E_ID
  if ( tskid < 1 || tskid >= TMAX_TSKID )
    /* not within the range */
    return E_ID;
#endif /* CHK_ACT_TSK_E_ID */
             ...
             ...
#ifdef CHK_ACT_TSK_E_NOEXS
  if ( _kernel_tskid_tab[tskid] != TRUE ) {
      /* not registered yet */
      _kernel_unl_cpu ();
```

[1]Strictly speaking, systems generated by our environment cannot be regarded as ones conforming to μITRON4.0, since the scheduling rules are different from the specification.

```
        return E_NOEXS;
    }
#endif /* CHK_ACT_TSK_E_NOEXS */
            ...
            ...
    if ( tcb -> actcnt < TMAX_ACTCNT ) {
        /* less than the max count */
            ...
            ...
    }
#ifdef CHK_ACT_TSK_E_QOVR
    else { /* more than the max count */
            ...
            ...
        _kernel_unl_cpu ();
        return E_QOVR;
    }
#endif /* CHK_ACT_TSK_E_QOVR */
            ...
```

Compiling this source code excludes unnecessary code fragments and generates an appropriate binary executable file which reflects the analysis results.

C. Adaptation of Scheduler

μITRON4.0 specification defines a scheduling rule which is based on fixed (static) priorities of tasks. On the other hand, in order to cope with various real-time embedded applications, our RTOS implementation provides several options for scheduling: fixed priority-based, EDF-based, and adaptive TBS(Total Bandwidth Server)-based scheduling [12, 13]. According to designers' choice, the scheduler is replaced with appropriate one. The purpose of this replacement is to improve real-time processing in terms of response times and jitters.

The EDF-based and adaptive TBS-based techniques require additional data structure for time (deadline) management. Therefore, as well as the scheduler itself, other kernel data structure such as task control blocks (TCB) is extensible in our kernel implementation. Designers have only to select options when building their applications to make use of the sophisticated scheduling algorithms.

High complexity of the sophisticated scheduling algorithms leads to large overhead of scheduling operation. We plan to make it possible to select hardware implementation of the scheduler to alleviate the overhead. Along with it, other system calls, after adaptation to the application, can be candidates of hardware implementation according to designers' demands.

V. Conclusions

In this paper, we introduced a framework for designing application-specific systems where multicore processors and real-time operating systems are adapted to the application. In the generated systems, only hardware and software resources which the applications actually require are implemented, which leads to highly efficient systems in terms of runtime, energy dissipation, and cost-performance. At present, generation of homogeneous, adapted multicore processors and generation of adapted RTOS software implementation are done.

In the near future, we undertake heterogeneous multicore processors where different tasks are allocated to different cores which are adapted to the corresponding tasks. In addition, hardware implementation of RTOS primitives including the scheduler is a plan to be followed and we compare our environment with some existing techniques [14, 15].

Acknowledgements

A part of this work is being supported by Japan Society for the Promotion of Science (JSPS) KAKENHI Grant Number JP 15K00073.

References

[1] M. K. Jain, M. Balakrishnan, and A. Kumar, "ASIP Design Methodologies: Survey and Issues," Proc of 14th Intl. Conf. on VLSI Design, pp.76–81, 2001.

[2] M. Imai, Y. Takeuchi, K. Sakanushi, and N. Ishiura, "Advantage and Possibility of Application-domain Specific Instruction-set Processor (ASIP)," IPSJ Transactions on System LSI Design Methodology, Vol.3, pp.161–178, 2010.

[3] P. Yiannacouras, J. Rose, and J. G. Steffan, "The Microarchitecture of FPGA-Based Soft Processors," Proc. of Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES), pp.202–212, 2005.

[4] P. Yiannacouras, J. Gregory, and J. Rose, "Application-Specific Customization of Soft Processor Microarchitecture," Proc. of Intl. Conf. on Field Programmable Gate Arrays (FPGA), pp.201–210, 2006.

[5] R. Dimond, O. Mencer, and W. Luk, "CUSTARD - A Customisable Threaded FPGA Soft Processor and Tools," Proc. of Intl. Conf. on Field Programmable Logic and Applications, pp.1–6, 2005.

[6] D.A. Patterson and J.L. Hennessy, Computer Organization and Design, 4th ed. *Morgan Kaufmann*, 2011.

[7] T. Miyauchi and K. Tanaka, "Building Automatic Optimizing Environment for Multicore Processors," Proc of Embedded Systems Symposium (ESS), pp.99–104, 2015. (In Japanese)

[8] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk, "Design Rationale of the PURE Object-Oriented Embedded Operating System," Proc. of Intl. Workshop on Distributed and Parallel Embedded Systems (DIPES), pp.231–240, 1998.

[9] C. Böke, M. Götz, T. Heimfarth, D. E. Kebbe, F. J. Rammig, and S. Rips, "(Re-)Configurable Real-Time Operating Systems and Their Applications," Proc. of Intl. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), pp.148–155, 2003.

[10] K. Tanaka, "Real-Time Operating System Kernel for Multithreaded Processor," Proc. of Intl. Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA), pp.91–99, 2006.

[11] μITRON4.0 Specification Ver.4.00.00, ITRON Committee, TRON ASSOCIATION.

[12] K. Tanaka, "Adaptive Total Bandwidth Server: Using Predictive Execution Time," Proc. of Intl. Embedded Systems Symposium (IESS), pp.250–261, 2013.

[13] K. Tanaka, "Virtual Release Advancing for Earlier Deadlines," ACM SIGBED Review, Vol.12, No.3, pp.28–31, 2015.

[14] M. Sindhwani and T. Srikanthan, "Framework for Automated Application-Specific Optimization of Embedded Real-Time Operating Systems," Proc. of Intl. Conf. on Information Communications & Signal Processing, pp.1416–1420, 2005.

[15] F. J. Rammig, M. Götz, T. Heimfarth, P. Janacik, and S. Oberthür, "Real-Time Operating Systems for Self-coordinating Embedded Systems," Proc. of Intl. Symp. on Object and Component-Oriented Real-Time Distributed Computing (ISORC), 2006.