九州大学学術情報リポジトリ
Kyushu University Institutional Repository

# Custom Instructions with Multiple Exits: Generation and Execution

Noori, Hamid
Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University

Mehdipour, Farhad
Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University

Inoue, Koji
Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University

Murakami, Kazuaki
Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University

他

https://hdl.handle.net/2324/6362

出版情報：情報処理学会研究報告. 2007 (4), pp.109-114, 2007-01. 情報処理学会ARC研究会
バージョン：
権利関係：ここに掲載した著作物の利用に関する注意 本著作物の著作権は（社）情報処理学会に帰属します。本著作物は著作権者である情報処理学会の許可のもとに掲載するものです。ご利用に当たっては「著作権法」ならびに「情報処理学会倫理綱領」に従うことをお願いいたします。

# Custom Instructions with Multiple Exits: Generation and Execution

Hamid Noori[†] Farhad Mehdipour[†] Koji Inoue[†] Kazuaki Murakami[†] and Maziar Goudarzi[††]

[†]Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University,
[††] System LSI Research Center, Kyushu University

E-mail: {noori, farhad}@c.csce.kyushu-u.ac.jp, {inoue, murakami}@i.kyushu-u.ac.jp, goudarzi@slrc.kyushu-u.ac.jp

**Abstract** In this paper, we propose an adaptive extensible processor in which custom instructions are generated and added after chip-fabrication. A reconfigurable functional unit is utilized to support this feature. The proposed reconfigurable functional unit is based on a matrix of functional units which is multi-cycle with the capability of conditional execution. A quantitative approach is utilized to fix the constraints of the architecture. Unlike previously proposed custom instructions, ours include multiple exits. Conditional execution has been added to support the multi-exit feature of custom instructions. Experimental results show that multi-exit custom instructions enhance the performance by an average of 46% compared to custom instructions limited to one basic block. A maximum speedup of 2.89, compared to a 4-issue in-order RISC processor, and an average speedup of 1.66 was achieved on MiBench benchmark suite.

**Keyword** Extensible Processor, Conditional Execution, Custom Instruction, Reconfigurable Functional Unit

## 1. Introduction

General Purpose Processors (GPPs), Application Specific Integrated Circuits (ASICs), Application Specific Instruction-set Processors (ASIPs), and extensible processors are well-known approaches for designing an embedded System-on-Chip (SoC). Although availability of tools, programmability, and rapid deployability in embedded systems are good reasons for common use, GPPs usually do not offer the necessary performance required. ASICs have much higher performance and lower power consumption though they are inflexible and have an expensive and time consuming design process. ASIPs have greater flexibility than ASICs and more potential to meet the challenging high-performance demands of embedded applications compared to GPPs. However, the synthesis of ASIPs traditionally involves the generation of a complete instruction set architecture (ISA) for the targeted application. This full-custom solution is too expensive and has a long design turnaround time.

Another method for providing enhanced performance is application-specific instruction set extension. By creating application-specific extensions to an instruction set, the critical portions of an application's dataflow graph (DFG) can be accelerated by mapping them to custom functional units. Custom instructions (CIs) reduce the latency of critical paths and number of intermediate results written to the register file. Though not as effective as ASICs, instruction set extension improves performance and decreases energy consumption of processors by reducing access to the instruction cache (dynamic energy) and execution time (static energy). Instruction set extension also maintains a degree of system programmability, which enables them to be utilized with more flexibility. The main problem with this method is the significant non-recurring engineering costs associated with implementation. The addition of instruction set extensions to a baseline processor for each application brings many of the issues associated with initially designing a new processor.

To reduce time-to-market and non-recurring engineering cost, an **AD**aptive **EX**tensible process**OR** (ADEXOR) is proposed in which CIs are generated and added after chip-fabrication. Generating and adding CIs are done fully automatically and transparently according to the behavior of target applications. Custom functional units are replaced with a *reconfigurable functional unit* (referred in this paper as CRFU) to support this capability. Using a reconfigurable functional unit also aids in supporting additional CIs. In proposing a proper architecture for the CRFU, a systematic quantitative approach was followed. Our CRFU is a coarse grain accelerator based on a matrix of functional units (FUs). It is tightly coupled with the base processor.

Both [4][4] and [5][5] show that a higher speedup can be obtained by extending CIs over basic blocks. In addition, a new method is used to relax CIs over basic blocks. Unlike other proposed CIs which are single entry and single exit, ours are single entry but multiple exits.

Multi-exit custom instructions (MECIs) are generated by linking hot basic blocks (HBBs). An HBB is a basic block with a greater execution frequency than a given threshold.

This paper has the following organization: A general overview of ADEXOR architecture is presented in Section 2. Section 3 discusses the algorithms for generating MECIs. The design methodology and quantitative approach for the proposed CRFU architecture are explained in Section 4. The experimental results are given in Section 5. The paper closes with conclusions and future work.

## 2. General Overview of Processor Architecture

ADEXOR, targeted for embedded systems, is composed of four main components: *i*) a base processor, *ii*) a coarse grain reconfigurable functional unit (CRFU) with functions and connections controlled by configuration bits, *iii*) a configuration memory for keeping the configuration bits of the CRFU for each MECI, and *iv*) counters for controlling the read/write signals of the register file and selecting between processor functional units and the CRFU (Fig. 1).

The base processor is a 4-issue in-order RISC processor that supports MIPS instruction set. The CRFU is parallel with other functional units of the processor (PFU). It is based on a matrix of functional units (FUs) with multiple inputs and outputs. The CRFU reads (writes) from (to) the register file. Each FU of the CRFU can support all fixed-point instructions of the base processor except *multiply, divide,* and *load*. *Multiply* and *divide* were excluded due to their low execution frequency and the large area that is needed for hardware implementation, and *loads* were ignored because of the cache misses and long memory access time which makes the execution latency unpredictable. Current implementation can support MECIs including one *store*. The CRFU is multi-cycle which requires execution cycles to be determined according to the depth of the DFG of each MECI and the clock frequency of the base processor. The number of execution cycles has to be recorded as part of the configuration data. Configuration memory also is used to update the program counter (PC) and find the valid exit point after executing each MECI. The architecture (numbers of inputs, outputs, FUs, etc.) is determined using a quantitative approach at *design phase* (Fig. 2).

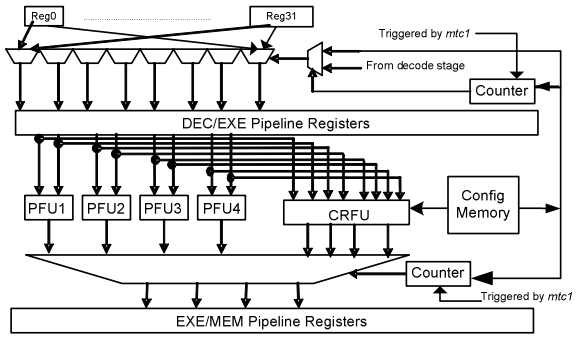


Fig. 1. Integrating the base processor with the augmented hardware.

The counters are used for controlling the register read/write signals of the register file and switching between PFUs and the CRFU. They are activated as soon as a MECI is detected. At this time, the required number of clock cycles for executing the corresponding MECI are loaded from the configuration memory into the counters. During the specified clock cycles, the counters select the configuration bits for choosing the input and output registers of the MECI for the CRFU while simultaneously selecting the CRFU outputs.

There are two phases when using ADEXOR: the *configuration phase* and the *normal phase* (Fig. 2). The configuration phase is done offline. In this phase, target applications are run on an instruction set simulator (ISS) and profiled. Then, the start addresses of hot basic blocks (HBBs) are detected [6][6]. MECIs are generated by linking the HBBs. Mapping the MECIs to generate configuration bits for the CRFU is done in this phase as well. To support the execution of MECIs, the object code is modified in the configuration phase. In the normal phase, the CRFU, configuration memory, counters, and new object code are employed for executing MECIs.

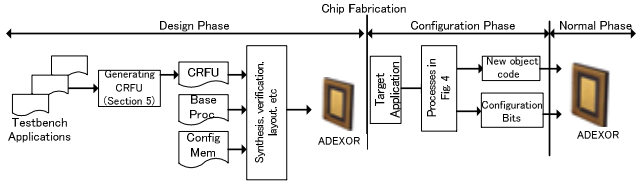


Fig. 2. Different phases for designing and using ADEXOR

The ISS is modified to profile two points of the processor: *i)* program counter (PC), and *ii)* committed instructions. There is a table for each profiler. The PC

profiler uses the same profiler as the one proposed in [6] and is used to detect the start address of the HBBs. The Instructions Profiler monitors the committed instructions and looks for branches to find the execution frequency and the taken counts. The instruction profiler table has three fields for each entry: *i)* address of branch instruction, *ii)* a counter for branch frequency, and *iii)* a counter to record the taken counts of branches.

## 3. Generating Multi-Exit Custom Instructions

### 3.1. Tool Chain Utilized for Generating MECIs

Fig. 3 shows the chain of main functions and tools that are used for generating MECIs. First, the applications are run on an instruction set simulator and profiled. Using the profiling data, the HBBs are detected, read from object code, and linked to make a hot instruction sequence (HIS). MECIs should not cross loop boundaries. Therefore, hot loops are detected and sorted from the innermost loop to the outermost in the ascending order considering the start addresses. To generate a HIS, the start address of the first HBB of the loop is passed and checked if it has been covered by previous MECIs. If it has not been covered, the HBB is read from the object code and added to the current HIS. An HBB reading terminates when a control instruction is encountered. Then the function in Fig. 4 is applied to the last instruction of the HBB.

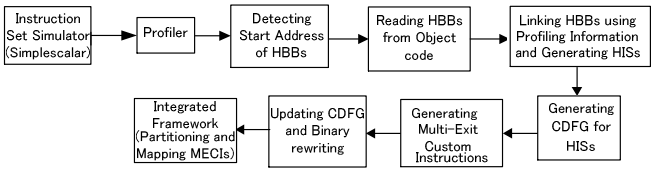


Fig. 3. Tool chain for generating MECIs

This process is repeated for each new added HBB until HIS reaches to the end (terminal) points in all directions. When HIS generation is completed for the loops, the process is continued for the remaining HBBs. First, the remaining HBBs are sorted in ascending order according to the start address and then HIS generation starts from the smallest address to the largest using a similar algorithm. The usual length of a HIS is around 15 to 60 instructions.

```
Function MAKE_HIS (objfile, HIS, start_addr)
1 if (HBB with start_addr is not included in previous MECIs) then read_add_HBB2HIS (objfile, HBB(start_addr), HIS) else return;
2 switch last_instruction(HBB)
3 case (indirect jump, return or call): return;
4 case (direct jump): MAKE_HIS(objfile, HIS, target address of jump);
5 case (branch):
5-1 if (it is hot backward) then return;
5-2 elsif (not-taken direction is hot) then MAKE_HIS(objfile, HIS, target address of not-taken direction) else return;
5-3 if (taken direction is hot) then MAKE_HIS(objfile, HIS, target address of taken direction) else return;
6 default: return;
```

Fig. 4. Function for checking the control (last) instruction of an HBB

The control dataflow graph (CDFG) is then generated for each HIS. In the CDFG, each input of nodes can have more than one source due to the different paths generated by the branches. Therefore, in the CDFG, all possible sources are generated for each input in addition to the effective branches used to select the sources. Then the CDFG is passed to the MECI generator.

### 3.2. Generating MECIs

In the current implementation, each MECI includes only fixed-point instructions except *multiply*, *divide*, and *load*. It can support only a single *store* instruction and up to five branches.

Our MECI generator gets the control flow graph (CFG) of each HIS called *Gc(V, Ec)(*where *V* is the set of nodes which denote primitive operations or instructions of the base processor, and edges in *Ec* represent the control dependencies*)* as input and looks for the largest valid convex subgraph *Gc'(V', Ec')* in the *Gc*. Next, the same set of *V'* in the corresponding DFG of the HIS called *Gd(V, Ed)* (where *V* is the set of nodes denoting primitive operations or instructions of the base processor, and edges in *Ed* represent the data dependencies) is specified and named as *Gd'(V', Ed')*. Then, by moving instructions in the object code, we try to add valid nodes to the entry and exit nodes of *V'* in *Gc'* using the following algorithm. As previously mentioned, HIS has a single entry node but can have multiple exit nodes.

For the entry node $v_i \in V'$ in *Gc'(V', Ec),* we look for a $v_j$ in *Gc(V, Ec)* so that $level(v_j) = level(v_i)-1$. If $v_j$ exists and is not valid, it is added to the *moving_node_list* set. The *moving_node_list* is the list of nodes that should be

moved in the object code. If $v_j$ is valid then the flow- and anti-dependence between $v_j$ and $\forall$ $v_k \in$ *moving_node_list* are checked. If a dependency does not exist then $v_j$ can be moved and added to the *V'* as a new entry node. The corresponding part of the object code, where the moving instructions will occur, should be checked whether it is a target of branch instructions before adding $v_j$ to *V'*. If that section of code is a target of branch instructions then the moving instructions should be canceled. If $v_j$ can not be added to *V'* due to the existence of dependencies then $\forall$ $v_m \in V'$ and $\forall$ $v_k \in$ *moving_node_list* flow- and anti-dependence are checked. If there is no dependency, then *V'* is moved and added to $v_j$ if the corresponding part of object code is not a target of branches. In both cases, moving should not cause instructions to cross over basic block boundaries. This process is continued for the new entry node $v_i$ of the updated list *V'* until $level(v_j) = -1$, for $v_j$ in $Gc(V, Ec)$. A similar idea is applied for all existing exit nodes.

In the current version, a MECI can have up to four exit points. The types of exit points are: *i) branch* with only one hot direction, *ii) indirect jump* and *return*, *iii) call*, *iv)* hot backward *branch* and *v)* an instruction whose descendant instruction is not valid (i.e. floating point, divide, multiply and load). Exit point addresses of a MECI are detected and saved as part of its configuration data. They are used to select a valid exit point when the MECI is executed on the CRFU. The instructions that have been moved should be rewritten in the object code and the CFG and the DFG of the HIS need to be updated as well. MECIs with $|V'| \leq 5$ are ignored.

After collapsing a subgraph as a MECI and overwriting the object code for moved instructions, the entry node of the subgraph of each MECI is then rewritten by a *mtc1* (move to coprocessor) instruction in the object code. *mtc1* is used to flag the subgraph as a MECI. The operand of the *mtc1* specifies the index to the configuration memory of the CRFU. The CRFU has a variable delay for each MECI which is dependent on the DFG depth of the MECI and the base processor clock frequency. To support this feature, in the normal mode, when a *mtc1* is using its operand as an index for configuration memory, the counters controlling the select signals of multiplexers (muxes) are loaded from the configuration memory for the corresponding MECI with the required execution clock cycles. In addition, the configuration is simultaneously loaded to the CRFU. The counters select the configuration bits to enable the required input/output registers from the register file for the MECI. They also select the CRFU output instead of PFU outputs. This causes the execution currently being carried out on the PFUs to switch to the CRFU for the specified clock cycles. When the execution of a MECI on the CRFU finishes, the counters force the muxes to switch to the decoder signals (to control the enable signals of registers) and PFUs (to execute the remaining of the application) (Fig. 1). Meanwhile, according to the results of the branches, the next PC is loaded from the configuration memory by a valid exit address and execution continues after the exit node.

## 4. Proposing an Architecture for the CRFU

### 4.1. Supporting Conditional Execution

In the DFG of CIs, the nodes (instructions) receive their input from a single source whereas, in the CDFG of a MECI, nodes can have multiple sources corresponding to the different paths generated by branches. The correct source is selected at run time according to the results of branches. Therefore, the CRFU should have some facilities to support conditional execution and generate valid output data and exit point.

We propose conditional data selection muxes for controlling selectors of muxes used for FU inputs, outputs of the CRFU, and exit point switches. Figure 5 (a) shows an example of a CRFU (with 5 FUs) without supporting conditional execution. In this architecture, the selection bits for input muxes of FU4 and FU5 are controlled by configuration bits.

To support conditional data selection, we have modified the hardware as shown in Figure 10 (b). In the proposed architecture, the selector signals of muxes used for choosing data for FU inputs (the *Data-Selection-Mux*), along with the CRFU output and exit point (not shown in the figure) are each controlled by another mux (the *Selector-Mux*). The inputs of Selector-Mux (one-bit width) originate from the FUs (which execute branches) of the upper rows and the configuration memory in order to control the selector signals conditionally, as well as unconditionally. The selectors of *Selector-Mux* are controlled by configuration bits. It should be noted the outputs of FUs are only applied to the *Selector-Muxes* in the lower-level rows, not in the same or upper rows. A similar structure is used for selecting the valid output data of the CRFU. To select the final exit point, a mux with four exit addresses as inputs is used. The selectors of this mux are controlled identically by the *Selector-Mux*.

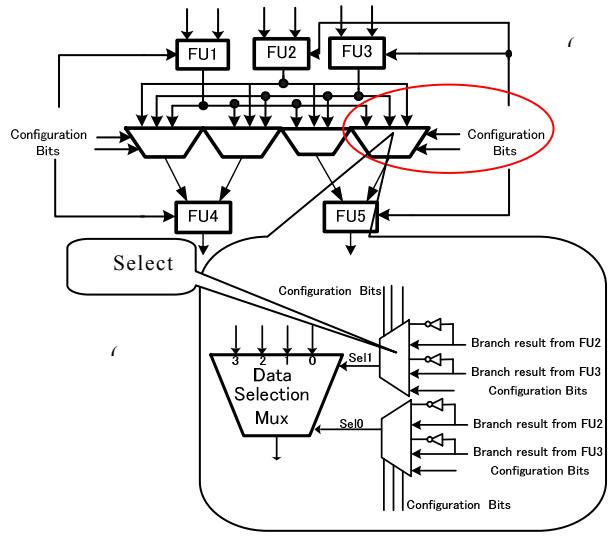


Fig. 5. Adding more hardware to the CRFU (a) to support conditional execution (b)

## 4.2. Proposed Architecture for the CRFU

We use the tool chain in Fig. 3 for the proposed quantitative approach to determine architecture parameters of the CRFU such as number of inputs, outputs, FUs, width, depth and etc.

Our simulation environment is based on Simplescalar (PISA configuration) [2]. 22 applications of Mibench [1] were selected as inputs for our quantitative approach. In this paper, the term *mapping rate* refers to the percentage of generated MECIs for 22 applications that could be mapped on the CRFU to the total number of generated MECIs for 22 applications. The execution frequency of MECIs was considered while calculating the mapping rate. All 22 applications of Mibench executed to completion. Since execution time varies for each application, a weight was assumed for each to equalize the execution time for comparison.

To determine the proper numbers for the CRFU inputs and outputs, all generated MECIs for 22 applications were mapped on the CRFU without considering any constraints (infinite number of inputs, outputs, FUs, depth, width, etc). By examining the mapping rate for different numbers of inputs, outputs and FUs, we achieved the curves shown in Fig. 6 and Fig. 7. These diagrams show that by choosing 8, 6 and 16, respectively for the number of inputs, outputs, and FUs, a high percentage (88.21%) of generated MECIs (for 22 applications) can be mapped on the CRFU.

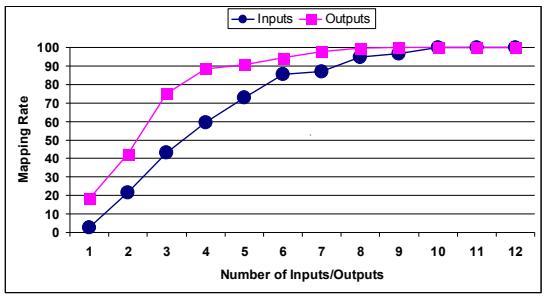


Fig. 6. The effect of different number of inputs, outputs on the mapping rate for 22 applications

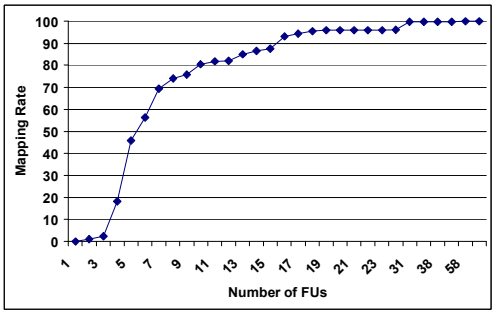


Fig. 7. The effect of different number of FUs on the mapping rate for 22 applications

We performed the same procedure to fix the architecture of the CRFU. The final architecture of the CRFU is shown in Figure 8.

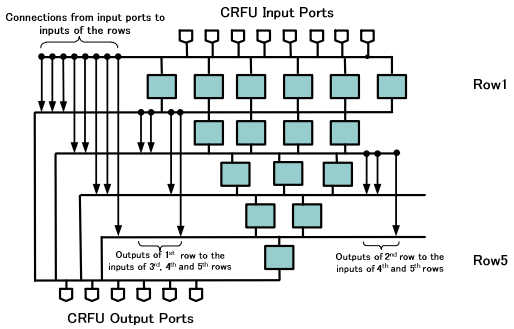


Fig. 8. Proposed architecture for the CRFU

The VHDL code of the proposed architecture was developed and synthesized using Synopsys tools and Hitachi 0.18 µm. The area of the CRFU is 2.1 mm$^2$. Since each FU output can be accessed directly via the output ports of the CRFU and also the *depth* (length of critical path in the DFG) of each MECI is known after mapping, we can have a CRFU with variable latency which depends on the depth of each MECI. The delay of the CRFU for

MECIs with various depths from 1 to 5 are **2.2 ns, 4.2 ns, 6.1 ns, 7.9 ns** and **9.8 ns**, respectively. The required clock cycles for executing each MECI is determined according to these numbers, depth of DFG and base processor clock frequency. The CRFU needs 375 bits for control signals and 240 bits for immediate values and exit points. So each MECI needs 615 bits in total for its configuration.

## 5. Experimental Results

The configuration of the base processor is in Table 2.

Table. 2. Base processor configuration

| | |
|---|---|
| Issue | 4-way |
| L1- I cache | 32K, 2 way, 1 cycle lat. |
| L1- D cache | 32K, 4 way, 1 cycle lat. |
| Unified L2 | 1M, 6 cycle latency |
| Execution units | 4 Integer, 4 FP |
| RUU & Fetch queue size | 64 |
| Branch predictor | bimodal |
| Branch prediction table size | 2048 |
| Extra branch misprediction latency | 3 |

To see the effect of base processor clock frequency on the speedup obtained using MECIs, five different frequencies were tried (Fig. 9). The speedup diminishes in higher frequencies since clock period becomes smaller but the CRFU delay remains unchanged. The high speedup obtained for *adpcm* is due to the main loop with 56 instructions, including 12 branches. Both taken and not-taken are hot for 7 of these branches which result in misprediction for 27% of the branches. Therefore, a large number of the executed clock cycles are due to the mispredicted branches (18%). The MECIs include both hot directions on a branch if present and hence, the CRFU architecture eliminates cycles of mispredicted branches.

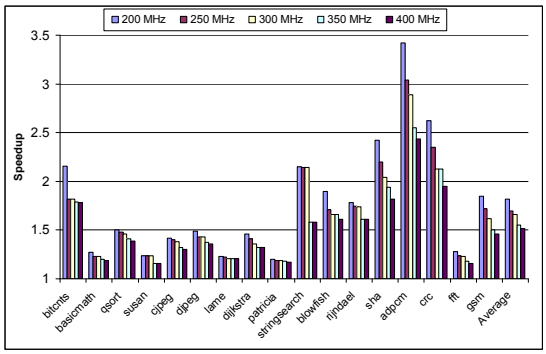


Fig. 9. Effect of the clock frequency on speedup

## 6. Experimental Results

To address the time-to-market and non-recurring engineering costs of extensible processors, an adaptive extensible processor was proposed in which custom instructions are generated and added after fabrication. An approach was presented for generating and executing custom instructions including multiple basic blocks. These custom instructions can include branch instructions and have multiple exits. Using a quantitative approach, a coarse grain reconfigurable functional unit was designed with conditional execution capability to support the multi-exit custom instructions. The experimental results show that by extending custom instructions over multiple hot basic blocks (HBBs), a maximum speedup of 2.89, compared to a 4-issue in-order RISC processor, and an average speedup of 1.66 was achieved on MiBench benchmark suite.

Our future works focus on energy evaluation of the proposed architecture and also exploring the design space for the CRFU considering area, performance and energy consumption.

## Acknowledgement

This research was supported in part by the Grant-in-Aid for Creative Basic Research, 14GS0218, Encouragement of Young Scientists (A), 17680005, and the 21st Century COE Program. We are grateful for their support. We also wish to express our gratitude to all members of System LSI laboratory at Kyushu University for their helpful advice and material and spiritual support.

## Reference

[1] http://www.eecs.umich.edu/mibench/.

[2] http://www.simplescalar.com/.

[3] P. Yu and T. Mitra, Characterizing Embedded Applications for Instruction-Set Extensible Processors, DAC 2004.

[4] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, “An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors”, ISCA 2005.

[5] P. Yu and T. Mitra, “Characterizing Embedded Applications for Instruction-Set Extensible Processors”, DAC 2004.

[6] H. Noori, K. Murakami, and K. Inoue, “A General Overview of an Adaptive Dynamic Extensible Processor”, Proc. of the Workshop on Introspective Architectures, 2006.