Enhancing Energy Efficiency of Processor-Based Embedded Systems through Post-Fabrication ISA Extension

Noori, Hamid
Institute of Systems, Information Technologies and Nanotechnologies

Mehdipour, Farhad
Research Institute for Information Technology, Kyushu University

Inoue, Koji
Department of Informatics, Kyushu University

Murakami, Kazuaki
Department of Informatics, Kyushu University

https://hdl.handle.net/2324/11887

出版情報：International Symposium on Low Power Electronics and Design. 2008, pp.241-246, 2008-08-12
バージョン：
権利関係：

Enhancing Energy Efficiency of Processor-Based Embedded Systems through Post-Fabrication ISA Extension

Hamid Noori
Institute of Systems, Information Technologies and Nanotechnologies
2-1-22 Momochihama- Sawara-ku, Fukuoka 814-0001, Japan
+81-92-852-3450
noori@isit.or.jp

Farhad Mehdipour
Research Institute for Information Technology, Kyushu University
3-8-33, Momochihama, Sawara-ku, Fukuoka, 814-0001, Japan
+81-92-847-5190
farhad@c.csce.kyushu-u.ac.jp

Koji Inoue Kazuaki Murakami
Department of Informatics, Kyushu University
744 Motooka Nishi-ku, Fukuoka 819-0395, Japan
+81-92-802-3794
{inoue, murakami}@i.kyushu-u.ac.jp

ABSTRACT

Application-specific instruction set extension is an effective technique for reducing accesses to components such as on- and off-chip memories, register file and enhancing the energy efficiency. However, the addition of custom functional units to the base processor is required for supporting custom instructions, which due to the increase of manufacturing and design costs in new nanometer-scale technologies and shorter time-to-market, is becoming an issue. To address above issues, in our proposed approach, an optimized reconfigurable functional unit is used instead, and instruction set customization is done after chip-fabrication. Therefore, while maintaining the flexibility of a conventional microprocessor, the low-energy feature of customization is applicable. Experimental results show that the maximum and average energy savings are 67% and 22%, respectively for our proposed architecture framework.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles – *Adaptable architectures.*

General Terms: Performance, Design, Experimentation.

Keywords: Custom Instruction, Reconfigurable Functional Unit, Conditional Execution, Low Energy Embedded Processor.

1. INTRODUCTION

The requirement of portability of embedded systems places severe restrictions on power consumption. Even though battery technology is improving continuously, battery life-time and battery weight are issues. Moreover, more computing power is required by these devices for future generations due to more functionality of the applications [6]. These properties raise the need to increase the energy efficiency of embedded systems.

Hardware/software partitioning [8] is shown to be effective for minimizing the power consumption of processor-based embedded systems. Other effective techniques are using Application Specific Instruction set Processors (ASIPs) and extensible processors [1][3][4][7][9][21]. A custom instruction (CI) encapsulates the computation of a frequently executed subgraph of the program's dataflow graph (DFG). Using CIs results not only in more speedup but also less energy consumption [3][5][17][5] due to reducing accesses to different components of the base processors (e.g. memories, decoder, register file, branch predictor, etc) compared to a conventional embedded processor. The target of these approaches is custom hardware. Although performance and energy efficiency can be obtained through custom hardwired implementation, it impacts flexibility. Moreover, the time and cost of designing and verifying a base processor with augmented custom hardware, causes many issues associated with designing a new processor from scratch, such as longer time-to-market and significant NRE (Non-Recurring Engineering) costs, specially that the NRE and design costs keep on increasing for the new technologies [6][13]. Hence, reconfigurability is becoming more important in future embedded processors [20].

In this paper, we describe our proposed **AD**aptive **EX**tensible process**OR** (ADEXOR) in which, CIs are generated and added after chip-fabrication. To cover higher percentage of dynamic instructions, unlike other methods for identifying and generating optimal set of CIs such as [1][4][7][21] that focus on CIs with single entry and single exit, we propose CIs with single entry but multiple exits (The CI entries and exits that we refer to are control dependencies for one node as opposed to data dependencies). Consequently, the proposed multi-exit CIs (MECIs) can cover hot directions of several branches into the CI without being limited to selecting just one or all of the directions. This brings about larger CIs, more instruction level parallelism (ILP), hiding branch misprediction penalty and reduction in accesses to the branch predictor. Moreover, we use a coarse-grain reconfigurable functional unit (referred in this paper as CRFU) instead of custom functional units, which brings flexibility and enables to support more CIs.

The main contributions of this paper are *i*) proposing various architectures for integrating CRFU with the base processor, *ii*) proposing two methods for invocating MECIs, and *iii*) energy consumption, area overhead, and performance evaluation of various proposed configurations of ADEXOR. In Section 2, we highlight the related work. The subjects related to MECIs are discussed in Section 3. Then in Section 4, the architecture of the CRFU is given. The experimental results are presented in Section 5 and finally paper is closed by conclusions.

2. RELATED WORK

The conventional embedded processors are the cheapest and the most flexible devices for implementing embedded systems, however their energy consumption is very high.

Henkel [8] presents an approach that minimizes the power consumption of embedded systems through hardware/software partitioning among a processor and ASICs. A concept of instruction subsetting is introduced in [5] to create an ASIP from a more general processor. This work defines the notion of instruction subsetting and explores its use as a means of reducing power consumption from the system level of design. The work in [17] describes an automatic methodology to select CIs for an extensible processor, in order to maximize its performance/energy efficiency for a given application. Biswas et al., present an instruction set extension identification technique in [3] that can automatically identify state-holding custom functional units, thus being able to reduce memory traffic from cache and main memory to improve performance and reduce energy. The target of these approaches is custom hardware. Significant manufacturing and design costs and shrinking time-to-market are issues of these approaches [6][13].

Although reconfigurable hardware consume more energy compared to custom hardware, they've been shown to be effective in energy saving while increasing flexibility. Wan et al. [19] present a fine-grain loosely-coupled reconfigurable architecture template for low-power digital signal processing, and then an energy conscious design methodology to bridge the algorithm to architecture gap. Stitt [16] describes a loop-oriented partitioning for moving critical code from software to a fine-grain loosely-coupled accelerator. They show the effectiveness of their proposed method in energy reduction as well as obtaining higher speedup. XiRisc [9] is a VLIW processor with a tightly coupled fine-grain reconfigurable functional unit. Mapping computation intensive algorithmic portions on the reconfigurable unit allows a more efficient elaboration, thus leading to an improvement in both timing performance and power consumption. Fine-grain reconfigurable accelerators allow for very flexible computations, but they consume more energy, have a longer latency and reconfiguration time compared to coarse grain counterparts. Moreover, they need a larger configuration memory (i.e. more energy consumption and area overhead).

CRISP [2] is a coarse-grain reconfigurable processor designed for multimedia applications. Its reconfigurable functional unit is composed of complex blocks such as ALUs and is tightly coupled with the base processor. Average 2.5 times the performance of a RISC processor is achieved with an average of 18% energy increase. The proposed reconfigurable processors need new programming model, new compiler, source code modifications, or new opcode for CIs which results in binary or object code incompatibility.

The method proposed in [10] shows the effectiveness of dynamic hardware/software partitioning on energy reduction by using binary code instead of source code. However, it needs an online profiler and hardware for dynamic optimization and synthesis. In this method, loops are accelerated on their proposed fine-grain configurable logic.

In our approach a coarse-grain reconfigurable functional unit is used, however to make it more energy efficient, the amount of augmented hardware is evaluated through a quantitative approach [12]. In the proposed architecture there is no need to a new programming model, new compiler, or new opcodes which obviate rewriting or recompiling the source codes. Consequently, our approach maintains binary compatibility and is applicable to cases where the source code is not available.

3. MULTI-EXIT CUSTOM INSTRUCTIONS

3.1 Motivation Example

Fig. 1 shows the control flow graph (CFG) of a part of the main loop in *adpcm*[11]. This part of loop contains six basic blocks (BB1 to BB6). Each node represents an instruction of the base processor. In order to have single-entry, single-exit CIs (atomic CIs), generating CI should be limited to one basic block or both taken and not-taken directions of branches should be included in the CI. If CI is limited to one basic block, due to the small basic blocks, no considerable improvement will be obtained for Fig. 1.

On the other hand, there are two main issues for including both taken and not-taken directions of branches into a CI. First, for some cases the target of taken direction of a branch is very far, or there are some nested branches. In these cases collapsing both taken and not-taken directions of branches results in very large CIs. However, there are always area constraints.

When the target of CIs is a reconfigurable hardware, where the instruction set extension is done after chip-fabrication, there are hardware resource constraints that limit the size of CIs. Suppose that there is a CRFU that can support up to 9 primitive instructions (nodes) and only atomic CIs are allowed. Therefore, the CI for Fig. 1 (starting from node 0) should be limited to BB1 while there is still hardware for more 5 nodes in the CRFU that are not used. However, if the CI were not limited to single exit, then the CI could be extended up to node 8, hence hardware resources in the CRFU could be used more efficiently. For this extension, the CI needs to support more than one exit. In this example, the CI has two exits at nodes 6 (taken direction of branch) and 8. If the taken direction of node 6 should be followed at execution time, although nodes 7 and 8 are included in the CI, they should not be executed or should not affect the final results. Hence, the MECIs are said to be non-atomic.

The second issue is that the execution frequencies of taken and not-taken directions widely vary for different branches.

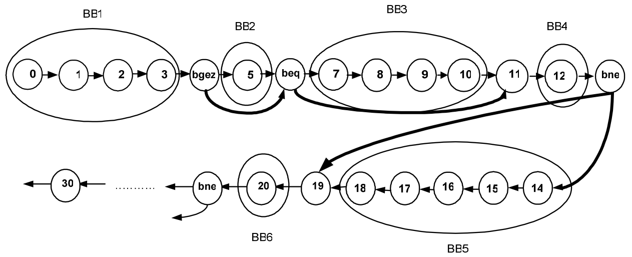


Fig. 1. Control flow graph of a part of adpcm loop

For example, in Fig.1, for the branches of nodes 4 and 6 the execution frequency of taken and not-taken is almost 50%-50% and 40%-60%, respectively. However for the branch of node 13, it is 95%-5%. Hence, for nodes 4 and 6, it is worth to collapse both directions in the CI but for node 13, excluding not-taken direction of branch from CI and adding more nodes from taken direction (starting from node 19) instead, results in more efficient use of the available hardware resources of the CRFU.

3.2 Tool Chain for Generating MECIs

Fig. 2 shows the tool chain that is used for generating MECIs. First, the applications are run on an instruction set simulator and profiled (our simulation environment is based on Simplescalar-PISA configuration [15]). Using the profiling data, the hot basic blocks (HBBs are basic blocks with an execution frequency more than a given threshold) are detected, read from object code, and linked to make a hot instruction sequence (HIS). The HIS is a link list of HBBs which is used as an input for generating MECIs. Indirect jump, call, return, and hot backward branches determine the end points (terminal) of a HIS.

MECIs should not cross loop boundaries, since they result in multi-entry CIs which we do not support. 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 checked whether it has been covered by previous MECIs or not. 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 a recursive function is applied to the last instruction of the HBB (which is a control instruction) for linking HBBs of hot directions to the HIS. This function examines the last instruction of the HBB. If it is *indirect jump*, *return*, *call*, or *hot backward branch* it returns from the function. If it is a branch and not-taken direction is hot, the function recalls itself with the target address of not-taken direction and if taken direction is hot, it is recalled with the target address of taken direction. In this way the HBBs are linked to each other and a HIS is generated[12].

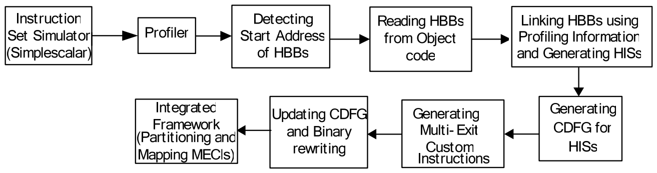

Instruction Set Simulator (Simplescalar)
Profiler
Detecting Start Address of HBBs
Reading HBBs from Object code
Linking HBBs using Profiling Information and Generating HISs
Generating CDFG for HISs
Generating Multi-Exit Custom Instructions
Updating CDFG and Binary rewriting
Integrated Framework (Partitioning and Mapping MECIs)

Fig. 2. Tool chain for generating MECIs

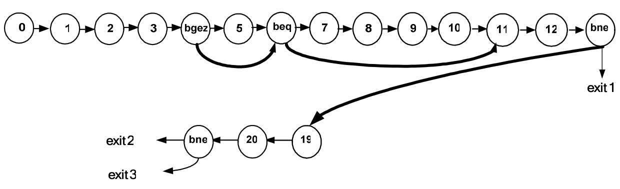


Fig. 3. Generated HIS for Fig. 1

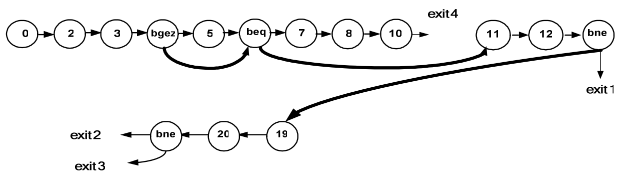
This process is repeated for each new added HBB until HIS reaches to the end (terminal) points in all directions. After processing the loops, the process is continued for the remaining HBBs. The remaining HBBs are sorted in ascending order according to the start address and then HIS generation starts from the smallest address to the largest using a similar algorithm. The control dataflow graph (CDFG) is then generated for each HIS and passed to the MECI generator. Fig. 3 shows the HIS for Fig. 1, assuming that the node 21 is a hot backward branch to node 0 and not-taken direction of node 13 is not hot.

3.3 Generating MECIs

In the current implementation, each MECI includes only fixed-point instructions except *multiply*, *divide*, and *load*. It can support at most one *store* instruction.

MECI generator looks for the largest sequence of instruction (subgraph) that can be executed on the CRFU, in the CDFG. Then, after checking the flow-, anti-, and output-dependence between instructions, *valid instructions* in each HBB are moved and added to the entry point (head) and exit point(s) (tails) of the detected largest instruction sequence (subgraph). *Valid instructions* are those instructions that can be executed by the CRFU and *invalid* (i.e. *floating point, load, divide, multiply,* second *store*) are those that are not supported by the CRFU.

For those parts where instructions are moved, the object code is rewritten. Moving instructions should be limited inside a basic block. 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 invalid. Exit point addresses of a MECI are detected and saved as part of its configuration data. These are used to select a valid exit point when the MECI is executed on the CRFU [12]. Fig. 4 shows the CFG of generated MECI (with four exits) for the HIS of Fig. 3 as well as the object code before and after generating MECI. The bold instructions are those instructions that have been collapsed into the MECI. Note that, *1)* nodes 1 and 9 are *load* (invalid) instructions, *2)* nodes 0, 10 and 11 are valid instructions *3)* there is no data-dependences between nodes 0 and 1 and also nodes 9 and 10, *4)* there are data-dependences between nodes 9 and 11, and *5)* nodes 0 and 1 and also nodes 9 and 10 have been swapped in the object code.



inst. #	*address*	*inst.*	*operands (dest, src1, src2)*
0	400410	addu	R13 R0 R0
1	400418	lw	R23 100 R2
2	400420	addiu	R4 R4 2
3	400428	subu	R3 R2 R11
4	400430	bgez	400440 R3
5	400438	addiu	R13 R0 8
6	400440	beq	400468 R13
7	400448	subu	R3 R0 R3
8	400450	addu	R10 R0 R0
9	400458	lw	R8 R9 0x3
10	400460	slt	R2 R3 R9
11	400468	addu	R8 R8 R9
12	400470	ori	R10 R10 1
13	400478	bne	4004a8 R2
14	400480	addiu	R10 R0 4
15	400488	subu	R3 R3 R9
16	400490	addu	R8 R8 R9
17	400498	sra	R9 R9 0x1
18	4004a0	slt	R2 R3 R9
19	4004a8	ori	R10 R10 2
20	4004b0	subu	R3 R3 R9
21	4004b8	bne	400410 R2
22	4004c0	slt	R2 R3 R9

Code before generating MECI

inst. #	*address*	*inst.*	*operands (dest, src1, src2)*
1	400410	lw	R23 100 R2
0	**400418**	**addu**	**R13 R0 R0**
2	**400420**	**addiu**	**R4 R4 2**
3	**400428**	**subu**	**R3 R2 R11**
4	**400430**	**bgez**	**400440 R3**
5	**400438**	**addiu**	**R13 R0 8**
6	**400440**	**beq**	**400468 R13**
7	**400448**	**subu**	**R3 R0 R3**
8	**400450**	**addu**	**R10 R0 R0**
10	**400458**	**slt**	**R2 R3 R9**
9	400460	lw	R8 R9 0x3
11	**400468**	**addu**	**R8 R8 R9**
12	**400470**	**ori**	**R10 R10 1**
13	**400478**	**bne**	**4004a8 R2**
14	400480	addiu	R10 R0 4
15	400488	subu	R3 R3 R9
16	400490	addu	R8 R8 R9
17	400498	sra	R9 R9 0x1
18	4004a0	slt	R2 R3 R9
19	**4004a8**	**ori**	**R10 R10 2**
20	**4004b0**	**subu**	**R3 R3 R9**
21	**4004b8**	**bne**	**400410 R2**
22	4004c0	slt	R2 R3 R9

Code after generating MECI

Fig. 4. Generated MECI for HIS of Fig. 3

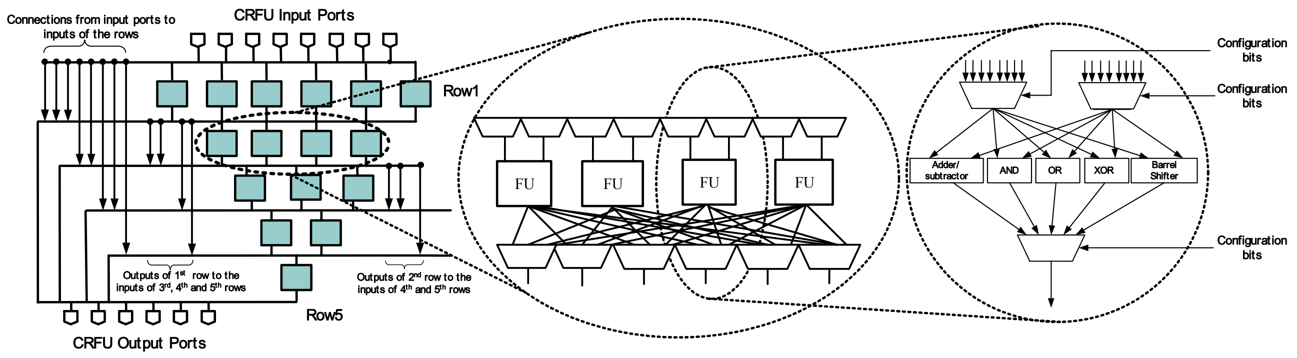

Connections from input ports to inputs of the rows
CRFU Input Ports
Row1
Outputs of 1st row to the inputs of 3rd, 4th and 5th rows
Outputs of 2nd row to the inputs of 4th and 5th rows
Row5
CRFU Output Ports
FU
Configuration bits
Adder/ subtractor
AND
OR
XOR
Barrel Shifter

Fig. 5. Proposed architecture for the CRFU

3.4 Invocating MECIs

Two MECI invocation techniques are proposed to handle MECIs detection and execution during run-time. In the first method (*invoke-mtc1*) the entry point instruction of the subgraph of each MECI is overwritten by *mtc1* (move to coprocessor) instruction in the object code to flag the start of a MECI. When the application is executed on the base processor and the *mtc1* is decoded, its operand is used for indexing and loading configuration bits from configuration memory of the CRFU for the corresponding MECI.

In the second approach (*invoke-seq*), a hardware called *sequencer* is utilized. The *sequencer* is a CAM (Context Address Memory) that keeps the addresses of entry nodes of MECIs (e.g. in Fig. 4 0x400418 is saved in the *sequencer*). Then, for each access to the instruction cache, the program counter is applied to the *sequencer*. For a hit the corresponding data is used for indexing the configuration memory to load the configuration bits of the MECI on the CRFU. Obviously *invoke-seq* imposes more area and energy overhead compared to *invoke-mtc1*. However, in *invoke-mtc1* we have the overhead for fetching and executing *mtc1* instructions and hence, less dynamic instructions coverage.

4. THE ARCHITECTURE OF THE CRFU

Our CRFU is a coarse-grain accelerator based on matrix of functional units (FUs) that support fixed point operations (excluding multiply and divide). It has eight inputs and six outputs. The width, depth and number of FUs of the CRFU are 6, 5, and 16 respectively. The final architecture of the CRFU is shown in Figure 5. The 8 input ports have been replicated and distributed among different rows to facilitate data access (7, 3, 2, 2, and 1 inputs for Row1 to Row5, respectively). The output of each FU in a row can be used by all FUs in the subsequent row (connections with length one). Besides to these connections, there are four connections with length two (Row1 $\rightarrow$ Row3 × 2, Row2 $\rightarrow$ Row4 × 2) and two connections with length three (Row1 $\rightarrow$ Row4, Row2 $\rightarrow$ Row5) and one connection with length four (Row1 $\rightarrow$ Row5).

The HDL code of the CRFU was developed and synthesized using Design Compiler (from Synopsys) and Hitachi 0.18µm library. The area of the CRFU is 1.7 mm^2. The CRFU needs 375 bits for control signals and 240 bits for immediate values and exit points. Therefore, each MECI requires a total of 615 bits for configuration. CRFU is a multi-cycle functional unit, to avoid being the critical path of the circuit. Each FU output can be accessed directly via the output ports of the CRFU and the depth of each MECI (length of critical path in the DFG) is known after mapping. Due to these facts, the CRFU can have a variable execution time in terms of the number of clock cycles, in which the required execution clock cycles are determined according to the depth of each MECI, the clock frequency of the base processor, and the delays of the CRFU for MECIs with various depths from 1 to 5 (which are 2.3ns, 4.2ns, 6.1ns, 8.0ns, and 9.8ns, respectively). According to the synthesis result, the clock frequency of ADEXOR is 130MHz, therefore, MECIs with depths 1, 2, and 3 need one clock cycle while MECIs with depth 4 and 5 require two clock cycles to be executed on the CRFU.

The CRFU is tightly coupled with the base processor. It is in parallel with the ALU. It reads/writes to/from the register file. The ADEXOR has two phases: *configuration phase* and *normal phase*. The *configuration phase* is done offline. In *configuration phase*, the tool chain in Fig. 2 is used for generating MECIs and their corresponding configuration bit-stream which are stored in the configuration memory. Inserting *mtc1* instructions or initializing *sequencer* are done in this phase as well. In the *normal phase* the bit-streams from the configuration memory are used and loaded on the CRFU for executing MECIs.

5. EXPERIMENT RESULTS

It is assumed that the base processor is a single issue in-order RISC processor (MIPS instruction set) with one ALU, one multiplier, one divider and floating point unit. Multiply and divide are run in parallel with ALU operations. The register file is exploited with four read ports and two write ports containing 32×32-bit registers. According to the synthesis results, multiplication and division take 5 and 8 cycles, respectively. Table 1 includes further details of the base processor.

The CRFU has 8 inputs and 6 outputs (Fig. 5) but the register file includes 4-read/2-write ports. We examine two architectures for integrating CRFU with the base processor. For the first architecture (referred as *arch1*), the available register file is used and the numbers of read/write ports of the register file are not modified. In this case for MECIs with more than four inputs, one more clock cycle is needed for reading other extra inputs. Also for MECIs with more than two outputs and less than five one extra clock cycle and for MECIs with more than four outputs two extra clock cycles are required for writing the results to the register file. For the second architecture (referred as *arch2*) the 4-read/2-write ports register file is replaced with an 8-read/ 4-write ports register file. In this case only one extra clock cycle is needed for MECIs with more than four outputs, however it affects the area overhead, clock frequency and energy consumption. We use different applications from Mibecnh [11] to perform the experiments.

Table 1. Base Processor Configuration

Issue	1-way
L1-I Cache, L1-D Cache	32K, 4 way, 1 cycle latency for hit, 20 cycles for miss
Execution units	1 Int unit, 1 FP unit , 1 div (8 cycles), 1 mult (5 cycles)
Branch predictor	bimodal
Branch prediction table size	256
Extra branch misprediction	3
Clock frequency	135 MHz

●*Area overhead*

The base processor (Table 1) was modeled using VHDL and synthesized using Hitachi 0.18µm library. The area of the base processor (without considering instruction and data caches) is 4.5mm^2. We modeled the instruction and data caches using CACTI [18] for 0.18µm. The area of a 32KB 4-way cache is 2.25mm^2. Considering the area of caches, the total area of the base processor is 9.0 mm^2.

The area of CRFU is 1.7 mm^2 (Section 4). Each MECI needs totally 615 bits (~ 80 bytes) for its configuration bit-stream. The configuration memory is assumed to keep up to 32 MECIs. Therefore, the size of the configuration memory is 80×32 bytes SRAM with a 640-bit width data bus, so that in one clock cycle the configuration can be loaded to the CRFU. The configuration memory was modeled using CACTI in 0.18µm. The area of configuration memory is 0.56mm^2. By adding the CRFU and configuration memory to the base processor (*arch1*) the area increases by 25.1%. In the case of using *invoke-seq* for invocating MECIs, the area of the *sequencer* (0.092mm^2) should also be considered, which results in 26.1% area overhead. By replacing the original register file with the one including 8-read/4-write ports, the area of ADEXOR (*arch2*) compared to the base processor increases by 30% for *invoke-mtc1* and 31% for *invoke-seq*. The clock frequency decreases by 3.7% (to 130MHz).

●*Energy Consumption Evaluation*

The power consumption of the base processor and the CRFU (Hitachi 0.18µm) are 71.5mW and 229.7mW, respectively. The power consumption of CRFU is larger than the base processor, however it is used only for executing MECIs, while the base processor is used in each clock cycle. We used CACTI 4.2 [18] to determine the energy for accessing a 32KB 4-way caches (instruction and data) and configuration memory (in 0.18µm), which are 0.294 nJ and 0.146 nJ, respectively. According to [22] it is assumed that for each cache miss and access to off-chip memory 25.0 nJ energy is consumed.

Using *invoke-seq*, there is another energy overhead that relates to the *sequencer* (which is a CAM). According to CACTI 4.2 the energy for each access to a full-set-associative memory with 32 entries is 0.184 nJ. For *arch2* compared to *arch1*, there is another energy overhead using a register file with more read/write ports. In 180nm the leakage power is negligible compared to the active power [14], therefore, it has been neglected in our evaluation.

Using MECIs and the CRFU result in less energy consumption because of shorter execution time and fewer accesses to different components of the base processor such as decoder, branch predictor, register file, ALU, and instruction cache. Reduction in access to instruction cache results in fewer instruction cache misses, hence fewer off-chip memory accesses which are too energy consuming. Fig. 6 shows the access reduction percentage of different components of the base processor for *invoke-seq* approach applied to different applications of Mibench.

Reduction of instruction cache misses is up to 48% for *fft*. As expected, because the *register file* has been shared between the CRFU and the ALU, the percentage of its access reduction is less compared to the other components. Besides, for the register file, before executing each MECI all the required input registers for different paths in a MECI should be read. *avg-seq* and *avg-mtc1* show the average access reduction regarding to the two proposed MECI invocation approaches: *invoke-seq* and *invoke-mtc1*. The *avg-mtc1* is almost 7% less compared to *avg-seq*, due to the *mtc1* instructions execution overhead. The average instruction cache miss reduction for *invoke-seq* is 8.2% while for *invoke-mtc1* is 7.5%. The average access reduction for instruction cache, register file, branch predictor, and other components are 55%, 23%, 42% and around 55% using *invoke-seq*, respectively.

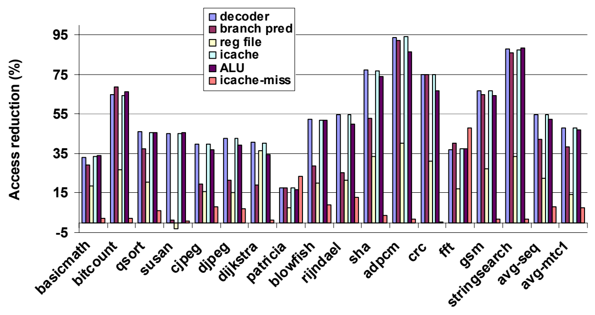


Fig. 6. Access reduction to different components of the base processor for *invoke-seq*

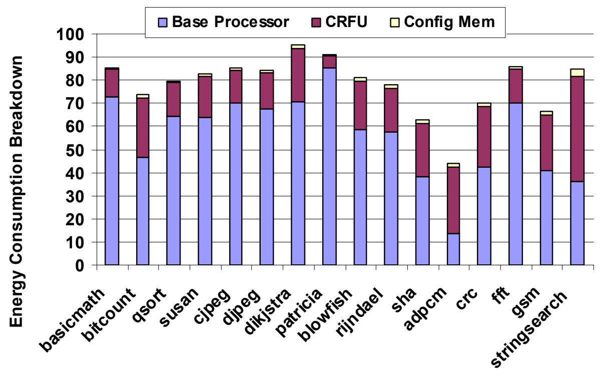


Fig. 7. Breakdown of energy consumption for *arch1/mtc1*

Fig. 7 and 8 show the normalized breakdown of energy consumption for *arch1/mtc1* and *arch2/sequencer*. The energy consumption of the base processor is assumed to be 100%. For *arch1/mtc1* energy consumption is reduced for all applications, however, for *arch2/sequencer,* some applications (e.g. *basicmath*, *dijkstra*, and *patricia*) consume more energy compared to the base processor, due to the *sequencer* and larger register file. For applications like *adpcm*, *crc*, *gsm*, and *sha* that MECIs cover a high percentage of dynamic instructions, more energy saving is obtained compared to other applications. Using MECIs and the CRFU improves the performance as well. Table 2 shows the minimum, maximum and average energy saving and speedup for different configurations of ADEXOR.

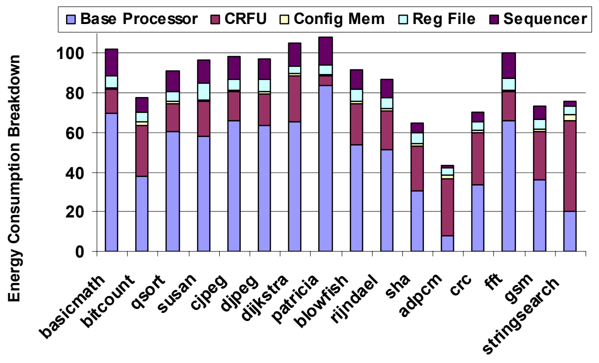


Fig. 8. Breakdown of energy consumption for *arch2/sequencer*

Table 2. Min, Max and Average energy saving and speedup for different configurations of ADEXOR

	Energy Saving Min,Max,Avg(%)	**Speedup Min,Max,Avg**	**Area Overhead**
arch1/mtc1	4.8, 56, 21.9	1.0, 3.6, 1.47	25.1%
arch1/sequencer	-3.1, 48.3, 18.9	1.13, 4.4, 1.67	26.1%
arch2/mtc1	1.9, 66.7, 16.0	0.92, 3.9, 1.58	30%
arch2/sequencer	-8.1, 56.7, 13.6	1.1, 4.9, 1.87	31%

6. CONCLUSIONS

To shorten time-to-market and reduce high design and NRE costs of extensible processors, an adaptive extensible processor was proposed in which CIs are generated and added after fabrication. An approach was presented for generating and executing CIs including multiple basic blocks. These CIs can include branch instructions and have single-entry but multiple exits. The coarse-grain reconfigurable functions unit used for executing MECIs is based on functional units with 8 inputs, 6 outputs and 16 FUs.

Two techniques were used for invocating MECIs including: *invoke-seq* and *invoke-mtc1*. In *invoke-seq* approach, more hardware (more area and energy overhead) is needed, however it can cover more dynamic instructions compared to *invoke-mtc1* approach. Using *invoke-seq* approach results in more average speedup (30% more compared to *mtc1*) while by using *invoke-mtc1* approach, more average energy saving (3% more compared to *invoke-seq*) can be reached. We also tried two architectures for integrating the CRFU with the base processor. In one case the register file has 4-read/2-write ports (*arch1*) and in the other case it has 8-read/4-write ports (*arch2*). Larger register file in *arch2* results in 5% more area overhead and 20% more speedup while 5% less energy saving can be obtained compared to *arch1*, in average. Experimental results show that the energy consumption is reduced up to 67% and 22% in average.

7. ACKNOWLEDGMENTS

We would like to thank Institute of Systems, Information Technologies and Nanotechnologies and all members of the System LSI Lab. of Kyushu University. This research was supported in part by Core Research for Evolutional Science and Technology (CREST) of Japan Science and Technology Corporation (JST).

8. REFERENCES

[1] Atasu, K., Pozzi, L. and Ienne, P., 2003. Automatic application-specific instruction-set extension under microarchitectural constraints. Proc. of DAC.
[2] Barat, F., Lauwereins, R. and Deconinck, G., 2003. Low-Power Coarse-Grained Reconfigurable Instruction Set Processor, Proceeding of FPL.
[3] Biswas P., Dutt, N.D., Ienne, P. and Pozzi, L., 2006. Automatic Identification of Application-Specific Functional Units with Architecturally Visible Storage, Proceeding of DATE.
[4] Blome, J., Chu, M., Mahlke, S., Biles, S. and Flautner, K., 2005. An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors. ISCA.
[5] Dougherty, W., Pursley, D.J., Thomas, D.E., 1999. Subsetting behavioral intellectual property for low power ASIP design. J. of VLSI Signal Process.
[6] Furuyama, T., 2007. Challenges of Digital Consumer and Mobile SoC's: More Moore Possible? Keynote Address, DATE.
[7] Goodwin, D. and Petkov, D., 2003. Automatic generation of application specific processors. CASES.
[8] Henkel, J. 1999. A Low-Power Hardware/Software Partitioning Approach for Core-based Embedded Systems, Proc. of DAC.
[9] Lodi, A. et al.2003. A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications. IEEE Journal of Solid-State Circuits, vol. 38, no. 11.
[10] Lysecky, R. and Vahid, F., 2005, A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning, Proc. of DATE.
[11] Mibench, www.eecs.umich.edu/mibench
[12] Noori, H., Mehdipour, H., Inoue, K. and Murakami, K., 2008. A Reconfigurable Functional Unit with Conditional Execution for Multi-Exit Custom Instructions. IEICE Trans. Electron., Vol. E91-C, No. 4, pp. 497-508.
[13] Sakurai, T., 2007. Meeting with the forthcoming IC Design, Keynote Address, Proceeding of ASP-DAC.
[14] Semenov, O. et al. 2003, Burn-in Temperature Projections for Deep Sub-micro Technologies, International Test Conference.
[15] Simplescalar, www.simplescalar.com
[16] Stitt, G., Lysecky, R. and Vahid, F., 2004, Energy Savings and Speedups from Partitioning Critical Software Loops to Hardware in Embedded Systems, ACM Transactions on Embedded Computing Systems.
[17] Sun, F., Ravi, S., Raghunathan, A. and Jha, N.K., 2004. Custom Instruction Synthesis for Extensible-Processor Platforms. IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems.
[18] Tarjan, D. et al. 2006. Cacti 4.0 HP Laboratories, Technical Report.
[19] Wan, M. et al. 2001, Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System. Journal of VLSI Signal Processing.
[20] Wong, S., Vassiliadis, S., Cotofana, S., 2004, Future Directions of Programmable and Reconfigurable Embedded Processors, Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation.
[21] Yu, P. and Mitra, T. 2004, Characterizing Embedded Applications for Instruction-Set Extensible Processors. Proc of DAC.
[22] Zhang, C, Vahid, F. and Najjar, W. 2005. A Highly Configurable Cache Architecture for Embedded Systems. ACM Transactions on Embedded Computing Systems, Vol. 4, No. 2.