Energy Consumption Evaluation of an Adaptive Extensible Processor

Noori, Hamid
Institute of Systems and Information Technologies/KYUSHU

Mehdipour, Farhad
Research Institute for Information Technology, Kyushu University

Goudarzi, Maziar
System LSI Research Center, Kyushu University

Yamaguchi, Seiichiro
Department of Computer Science and Communication Engineering, Kyushu University

他

出版情報：Proceedings of the Second Annual Reconfigurable and Adaptive Architecture Workshop, 2007-12-01
バージョン：
権利関係：

Energy Consumption Evaluation of an Adaptive Extensible Processor

Hamid Noori[±], Farhad Mehdipour[‡], Maziar Goudarzi[§], Seiichiro Yamaguchi[††], Koji Inoue[†], and Kazuaki Murakami[†]

[±]Institute of Systems and Information Technologies/KYUSHU
[†]Department of Informatics, Kyushu University
[‡]Research Institute for Information Technology, Kyushu University
[§]System LSI Research Center, Kyushu University
[††]Department of Computer Science and Communication Engineering, Kyushu University

noori@isit.or.jp
{farhad, seiichiro}@c.csce.kyushu-u.ac.jp, goudarzi@slrc.kyushu-u.ac.jp,
{inoue, murakami@i.kyushu-u.ac.jp}

Abstract

Most embedded systems rely on batteries as their source of energy, and hence, low power consumption is inherently essential for them. In processor-based embedded systems, a large portion of power is consumed for accessing instruction memories (including on-chip caches and off-chip memories), register-file, and also in the clock-distribution tree. Encapsulating critical computation subgraphs as application-specific instruction set extensions is an effective technique to reduce above-mentioned accesses and execution time (clock energy) and consequently, enhance the energy efficiency of these systems. However, the addition of custom functional units to the base processor is required to support the execution of custom instructions, which due to the increase of manufacturing and design costs in new nanometer-scale technologies and shorter time-to-market it is becoming an issue. To address these issues, in our proposed approach, the custom functional units are replaced with a reconfigurable functional unit and instruction customization is done after chip-fabrication. Therefore, while maintaining the flexibility of a conventional microprocessor, the energy reduction feature of customization is utilized for enhancing the energy efficiency. Our reconfigurable functional-unit is capable of executing custom instructions with multiple exits, resulting in 15% more energy saving compared to the single-exit counterpart. Moreover, two custom instruction invocation techniques are compared in terms of energy saving. Experimental results show up to 79% reduction (37% on average) energy saving on MiBench benchmark suit.

1. Introduction

The requirement of portability of embedded systems places severe restrictions on power consumption. Even though battery technology is improving continuously and processors and displays are rapidly improving in terms of power consumption, battery life-time and battery weight are issues that will have a marked influence on how embedded systems can be used. Moreover, more computing power is required by these devices for new generations due to more functionality and complexity of the supported applications [24]. Therefore, power consumption is becoming the limiting factor in the amount of functionality and complexity that can be placed in these devices. These properties raise the need to increase the energy efficiency and to extend the battery lifetime of embedded systems.

Hardware/software partitioning [1] is shown to be effective for minimizing the power consumption of embedded processor-based systems. Other effective techniques to enhance the efficiency of these systems, are using Application Specific Instruction set Processors (ASIPs) or extensible processors [3][4][13][14][15][16]. 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 [2][3][4], due to reducing accesses to different components of the base processors (e.g. memories, decoder, register file, etc) and the execution time (i.e. less clock energy), compared to a conventional embedded microprocessor. The targets of these approaches are custom hardware. Although performance/energy efficiency can be obtained through custom hardwired implementation, however 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 [5][24]. Therefore, reconfigurability is becoming more important in future embedded processors [6].

Although reconfigurable hardware consumes more energy compared to custom hardware, still it shows the potential in energy reduction while maintaining the flexibility [7][8][9][10][11][12].

In this paper, we describe our proposed **AD**aptive **EX**tensible process**OR** (ADEXOR) in which, CIs are generated and added after chip-fabrication automatically. To cover higher percentage of dynamic instructions, unlike other methods for identifying and generating optimal set of CIs such as [13][14][15][16] that focus on CIs with a single entry and a single exit, we propose CIs with single entry but multiple exits. 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. MECIs can cover both directions of branches if both are hot. 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 conditional execution is supported by the CRFU to handle multi-exit feature of the proposed CIs. Experimental results show that using the proposed approach the energy consumption can be reduced up to 79% and 37% in average, while providing flexibility and compatibility. The contributions of this paper are 1) energy evaluation of the proposed architecture, 2) showing its effectiveness on energy reduction, 3) comparing two MECI invocation techniques in terms of energy saving, and 4) showing the effectiveness of proposed MECIs over CIs limited to one basic block on energy efficiency enhancement.

2. Related Work

The conventional embedded microprocessors are the most flexible devices for implementing embedded and mobile systems, however their energy consumption is very high. One solution is moving frequently executed portions of the applications to custom hardware.

Henkel [1] presents an approach that minimizes the power consumption of embedded core-based systems through hardware/software partitioning among a processor and ASICs. A concept of instruction subsetting is introduced in [2] 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 [3] describes an automatic methodology to select CIs to augment an extensible processor, in order to maximize its performance/energy efficiency for a given application program. Biswas et al., present an instruction set extension identification technique in [4] 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. Significant manufacturing and design cost and shrinking time-to-market are becoming issues for these approaches [5][24].

Wan [7] presents 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 [8] 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 accelerators allow for very flexible computations, but they consume more energy, have a longer latency and reconfiguration time compared to coarse grain counterparts. Furthermore, they need a larger amount of memory (more energy consumption) for storing configuration bits.

CRISP [10] is a coarse-grain reconfigurable instruction set 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. In [11] a new low-power Processor-In-Memory-based 32-bit reconfigurable datapath optimized for multimedia applications is presented. They also show the effectiveness of reconfigurable accelerator for reducing energy. Almost all of the proposed reconfigurable processors and systems 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 [12] shows the effectiveness of dynamic hardware/software partitioning on energy reduction, using binary code instead of high level or source code. However, it needs 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 [17]. 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. To obviate extra hardware for online profiler, dynamic optimization and synthesis, two phases are defined.

3. Overview of ADEXOR Architecture

ADEXOR is composed of four main components: *i*) a base processor which is a 4-issue in-order RISC processor, *ii*) a coarse grain reconfigurable functional unit (CRFU) whose functions and connections are 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 ALUs and the CRFU (Fig. 1).

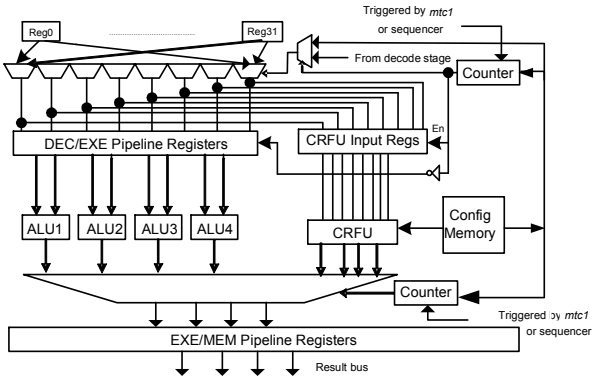

Figure 1. Integrating the base processor with the CRFU

In order not to increase the number of read/write ports of the register file, it is shared between CRFU and ALUs. To make the CRFU inactive while application is running on the base processor and also inactivate ALUs while MECIs are executed on the CRFU, two sets of input registers have been considered for the ALUs and the CRFU, with complement enable signal. The CRFU is in parallel with ALUs. It is fully combinational and is based on matrix of functional units (FUs) with eight inputs, six outputs and 16 FUs (Fig. 2). The CRFU reads (writes) from (to) register file. The CRFU is assumed to be multi-cycle to avoid becoming the critical path of the circuit. The required execution cycles for running MECIs on the CRFU is variable and depends on the depth of the DFG of each MECI and the clock frequency of the base processor which is kept as part of configuration bit-stream.

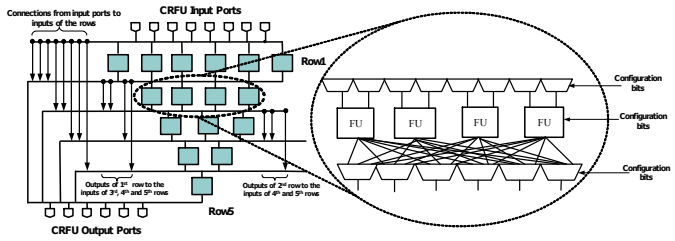

Figure 2. Proposed architecture for the CRFU

When a MECI is detected in the object code, the required clock cycles for executing the corresponding MECI is loaded from configuration memory into the counters. For the specified clock cycles the counters select the configuration bits to enable the required input and output registers of the MECI for the CRFU and its inputs and outputs.

Each FU of CRFU can support fixed-point instructions of the base processor except *multiply, divide* and *load*. It can support MECIs including at most one *store.* CRFU uses configuration memory to update the program counter (PC) and find the valid exit point, after executing each MECI. The specifications of the CRFU have been determined using a quantitative approach [17]. In our quantitative approach, first the MECIs were generated for 21 applications of Mibench [20] without considering any constraints (i.e. number of inputs, outputs, FUs, connections and etc). Then according to the *mapping rate* (which shows the percentage of the generated MECIs for 21 applications that can be mapped on the CRFU considering their execution frequency and execution time of each application) different constraints such as number of inputs, outputs, FUs and etc are determined for the CRFU. The result is the proposed architecture for the CRFU in Fig. 2, for which 81.34% of generated MECIs for 21 applications can be mapped on it. The remaining 18.66% of generated MECIs are partitioned to smaller and mappable MECIs using our proposed integrated mapping-temporal partitioning framework [19]. The CRFU can support conditional execution via conditional data selection which is required for executing MECI [17].

Two phases have been defined for using ADEXOR: *configuration phase* and *normal phase* (Fig. 3). In the configuration phase, which is done offline, target applications are run on an instruction set simulator (ISS) and profiled at binary level. Then, the start addresses of hot basic blocks (HBBs) are detected. An HBB is a basic block with an execution frequency more than a given threshold. MECIs are generated by linking HBBs. Mapping the MECIs to generate configuration bits for the CRFU is done in this phase as well. In the normal phase, the CRFU, its configuration memory and counters are employed for executing MECIs.

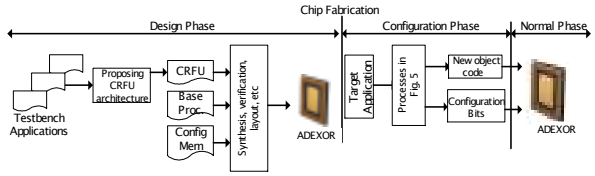

Fig. 3. Different phases for designing and using ADEXOR

4. Generating Multi-Exit Custom Instructions

4.1. Motivation example

Fig. 4 shows the control flow graph (CFG) of the main loop in *adpcm*[20]. The number of instructions in each blocks range from 1 to 6 with an average of 2.24 instructions. Each block is quite small, as shown by these numbers therefore, extending

custom instructions over a basic block helps to cover more instructions. Continuous red arcs are hot directions (with an execution frequency greater than a given threshold) and dotted black ones are not. Both directions are hot for the majority of the branches. In these cases, covering both directions of a branch can aid the creation of larger CIs, hence more parallelism, as well as eliminate branch misprediction penalties, and reducing accesses to the branch predictor. In extensible processors with custom functional units, because the amount of augmented hardware can be determined by the designer before fabrication and since handling CIs with single exit is easier, the designers prefer to select single entry-, single exit subgraphs as a CI. However, in our case because the hardware resources (CRFU) are fixed, in order to be able to extend CIs over basic blocks effectively and use the available hardware (CRFU) efficiently, MECIs are proposed.

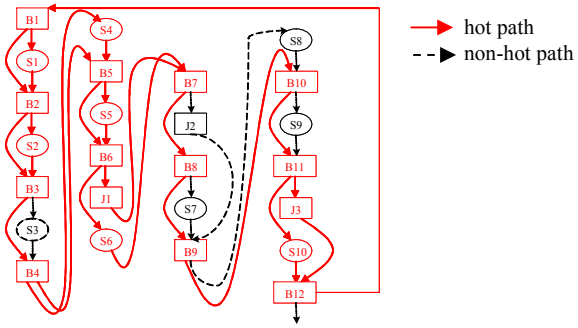

Figure 4. Control flow graph of main loop in *adpcm*

4.2. Generating MECIs

Fig. 5 shows the chain of main functions and tools that are used for generating MECIs. First the applications are run on the ISS (1) and profiled at binary level (2). Using the profiling data, the HBBs are detected (3) and linked to make hot instruction sequence (HIS). MECIs should not cross loop boundaries. Therefore, first, hot loops are detected and sorted from the innermost loop to the outermost in the ascending order of their start addresses. To generate a HIS, the start address of the first HBB of the loop is passed and 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 (4) and added to the current HIS. For each HBB the last instruction is checked. If the not-taken direction is hot the HBB in the not-taken is added to the HIS. If the taken direction is hot, the HBB in taken direction is added and if both taken and not-taken are hot, both are added to the HIS using a recursive function. This process is repeated for each new added HBB until HIS reaches to the end (terminal) points in all directions. When generating HIS is done for the loops, similar process is continued for the remaining HBBs (5).

Then the control dataflow graph (CDFG) is generated (6) and passed to the MECI generator. In current implementation, each MECI can include only fixed-point instructions except *multiply*, *divide* and *load*. It can support at most one *store* instruction, up to five branches and four exit points. The exit points of a MECI are: *i)* branch with only one hot direction, *ii) indirect jump, return* and *call*, *iii)* hot backward branch and *iv)* an instruction where its next instruction is *non-executable*. *Executable instructions* are those instructions that can be executed by the CRFU and *non-executable* (i.e. *floating point, load, divide, multiply,* second *store*) are those that are not supported by the CRFU.

MECI generator looks for the largest subgraph that can be executed on the CRFU, in the CDFG. Then, after checking the flow-, anti-, and output-dependence, *executable instructions* in each HBB are moved and added to the entry point (head) and exit point(s) (tails) of the detected subgraph (7). For those parts that instructions are moved, the object code is rewritten. Then it is passed to the integrated framework [19] to partition large MECIs to mappable MECIs on the CRFU (9). Next, generated MECIs are mapped on the CRFU.

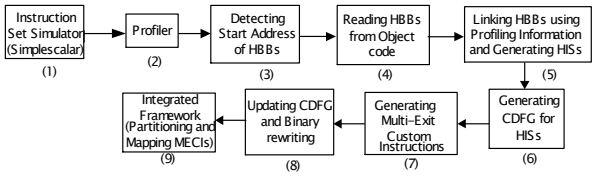

Fig. 5. Tool chain for generating MECIs

To insert MECIs into the instruction stream and remove the individual instructions from the stream at the *normal phase*, two MECI invocation techniques are used. In the first method the entry point instruction of the subgraph of each MECIs is overwritten by *mtc1* (move to coprocessor) instruction in the object code to flag the start of a MECI. In the *normal phase* when 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 a hardware called *sequencer* is utilized. The *sequencer* is a table (a context address memory or CAM) that keeps the address of logically previous instructions of MECIs. The *sequencer* is initialized at the *configuration phase*. In the *normal phase* 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.

5. Energy Consumption Evaluation Model

In 180nm (our target technology) the leakage power is negligible compared to the active power [23], therefore, it has been neglected in our model. The energy consumption of the base processor comprises from three main parts: *i)* energy of the clock-tree, *ii)* energy of different components of the base

processor, and *iii)* energy for accessing off-chip memory. The energy of clock-tree is the product of number of execution clock cycles and the energy for each clock pulse. The energy of each component is measured by number of accesses to each component times the energy of each access and finally, the energy of off-chip access is the product of number of misses of instruction and data caches and energy for each off-chip access (Eq. (1)).

$$Base\ Processor\ Energy = (num\_exec\_cycle * energy(clock)) + \sum_{comp} num\_access(comp) * energy(comp) + (icache\_miss + dcache\_miss) * energy\_off\_chip\_access \quad \text{Eq.(1)}$$

where $energy(X) = power(X) * clock\_period$ Eq. (2)

The *num_access(comp)* and *power(comp)*, respectively are number of accesses and power consumption of different components of the base processors (e.g. instruction and data caches, integer and floating point ALUs, result bus, register file, and etc) which are obtained through running each application on the Wattch [21]. The *num_exec_cycle, icache_miss,* and *dcache_miss,* which show the number of execution clock cycles for an application, number of instruction and data cache misses, are obtained using Wattch as well. The penalty for each miss is assumed to be 20 cycles and according to [22] the energy for each off-chip access is considered to be 40nJ. The target of Wattch has been set for 180nm technology. The *clock_period* is clock period of the base processor. Five different *clock_period* fvalues are examined in Section 5.

The energy of ADEXOR is calculated similar to the base processor, which is added to energy overhead (due to the CRFU, and configuration memory). The numbers of accesses to different components (*Anum_access*), instruction cache miss (*Aicache_miss*), and execution clock cycles (*Anum_exec_cycle*) changes due to the execution of MECIs on the CRFU (Eq. 3). The MECIs do not affect the number misses of data cache. The *Energy_Overhead* is the energy overhead of ADEXOR due to the extra hardware.

$$ADEXOR\ Energy = Anum\_exec\_cycle * enrgy(clock) + \sum_{comp} Anum\_access(comp) * energy(comp) + (Aicache\_miss + dcache\_miss) * energy\_off\_chip\_access + Energy\_Overhead \quad \text{Eq. (3)}$$

It was mentioned that the CRFU has six outputs but Fig. 1 depicts that it has four outputs. To support CRFU with six outputs without increasing the number of write ports of the register file, two registers have been added to the CRFU. When a MECI has more than four outputs, four of them are written in one cycle and the remaining ones in the next cycle. Therefore, for executing MECIs with more than four outputs (comprising around 13.2% of MECIs generated for 21 applications), one more clock cycle is needed.

The number of execution clock cycles for ADEXOR (*Anum_exec_cycle*) comprises of two parts.

$$Anum\_exec\_cycle = num\_exec\_cycle\_W/O\_MECI + Clk\_MECI\_CRFU(clock\_period) \quad \text{Eq. (4)}$$

num_exec_cycle_W/O_MECI is the number of execution clock cycles for running those portions of the application that are executed on the base processor and *Clk_MECI_CRFU* is number of clock cycles required for executing MECIs on the CRFU. *Clk_MECI_CRFU* depends on the clock frequency of the base processor because, for faster clock frequencies more clock cycles are required for executing MECIs on the CRFU. Therefore, the total energy of clock for ADEXOR is calculated utilizing the following equation:

$$Total\ energy\ clock = (num\_exec\_cycle\_W/O\_MECI + (NCLK\_GATING * Clk\_MECI\_CRFU)) * energy(clock) \quad \text{Eq. (5)}$$

The CRFU is fully combinational therefore there is no need to the clock for executing MECIs on the CRFU. If the clock can be gated while executing MECIs on the CRFU, *NCLK_GATING* will be zero, otherwise it will be one.

The energy overhead of *mtc1* MECI invocation technique has three main terms. The first term relates to the register file. Because the register file has been shared between ALUs and the CRFU, its fan-out has been doubled therefore, we assume that each access to the register file consumes two times more energy compared to the base processor without the CRFU. The second term corresponds to the CRFU which is the summation of the product of its power and its delay for each execution of a MECI. The final term is the energy of the configuration memory which is the energy for each access times the total execution of MECIs for the target application (Eq. (6)).

$$mtc1\_energy\_overhead = ((Anum\_access(regfile) * energy(regfile) + \left(\sum_{i=1}^{\#MECIs} exec\_freq(i) * ((CRFU\_power * CRFU\_delay) + Config\_Mem\_Energy) * OVH\_FACT \right) \quad \text{Eq. (6)}$$

#MECIs is the number of MECIs generated for each application and *exec_freq* is the execution frequency of each MECI. The VHDL code of the CRFU was developed and

synthesized using Synopsys tools and Hitachi 180nm library. 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 hence, we can have a CRFU with variable latency in which the latency depends on the depth of each MECI. The delays 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 the aforementioned numbers, depth of its DFG and base processor clock frequency. However we assume the worst case for *CRFU_delay* in Eq. (6) which is 9.8ns. Verilog-XL from Cadence, Power Compiler from Synopsys and 180nm technology cell library from Hitachi were exploited to measure the power of CRFU. The power consumption of the CRFU is 246.335 mW which is used as *CRFU_power* in Eq. (6). The CRFU needs 375 bits for control signals and 240 bits for immediate values and exit points. Therefore, each MECI needs 615 bits (~ 80 bytes) in total for its configuration. The configuration memory is assumed to keep up to 100 MECIs. Therefore, the size of the configuration memory is 80x100 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 [18] in 0.18µm. The area is 0.77mm^2 and the energy for each access is 0.198 nJ (*ConfigMem_Energy* at Eq. (6)).

The energy overhead of *sequencer* approach equals the energy overhead of *mtc1* plus the energy of the *sequencer*. The energy of sequencer is the energy for each access to sequencer times the number of access to the instruction cache.

$$sequencer\_energy\_overhead = mtc1\_energy\_overhead + (Anum\_access(instr\_cache) * sequencer\_Energy * OVH\_FACT) \quad \text{Eq. (7)}$$

The *sequencer* table was modeled using CACTI. The area of the *sequencer* is 0.61 mm^2 and the energy for each access is 0.29 nJ (*Sequencer_Energy*).

In *mtc1* the energy overhead is less compared to the *sequencer,* due to less required hardware, however the disadvantage is that the *mtc1* instruction itself should be fetched and decoded. The other advantage of *sequencer* is that the object code does not need to be modified.

Since the energy of the base processor, the CRFU, configuration memory and the *sequencer* have been measured using different tools, a factor has been considered for the energy overhead (*OVH_FACT*). The effect of this coefficient on the total energy reduction is studied in the next section. The *access reduction* and *energy reduction* used in the next section are measured using the following equations.

$$access\ reduction(X) = \frac{num\_access(X) - Anum\_access(X)}{num\_access(X)} * 100$$

$$energy\ reduction = \frac{base\_processor\_energy - ADEXOR\_energy}{base\_processor\_energy} * 100$$

6. Evaluation Results

The configuration of the base processor is in Table 1. We tried 21 applications of Mibench [20].

Table 1. Base processor configuration

Issue	4-way
Level-one Instruction Cache (2-way) and Data Cache (4-way)	16KB, 1 cycle for hit, 20 cycles for miss
Execution units	4 integer, 4 floating point
Multiplier (int, floating), Divider (int, floating)	(1, 1) (1, 1)
Branch predictor	bimodal
Branch prediction table size	2048
Extra branch misprediction latency	3 cycles

● *Component accesses reduction*

Fig. 6 shows the percentage of access reduction for different components of the base processor when *sequencer* is utilized. Due to collapsing instructions as custom instructions and according to the types and instructions that can be supported by MECIs, accesses to *decoder*, *branch predictor*, *register file*, *instruction cache*, *integer ALUs* and *result bus* are reduced. Moreover, due to access reduction to instruction cache, number of instruction cache misses and hence off-chip memory access is reduced as well. Reduction of instruction cache misses is up to 86% for *rijndael*. As expected, because the *register file* and the *result bus* have been shared between the CRFU and the ALUs, the percentage of their access reduction is less compared to the other components.

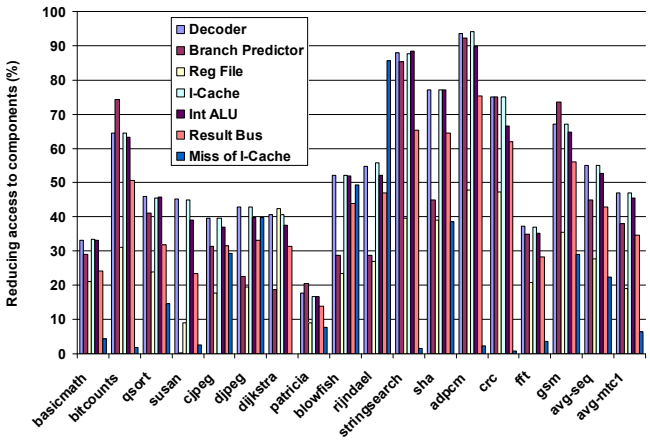

Figure 6. Access reduction to different parts of the base processor using the CRFU & *sequencer*

Specially, for the register file, since 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 (for 21 applications) regarding to the two proposed MECI invocation approaches: *sequencer* and *mtc1*. The *avg-mtc1* is almost 8% less compared to *avg-seq*, due to the *mtc1* instructions execution overhead. The average instruction cache miss reduction for *sequencer* is 22.5% while for *mtc1* is 6.5%. The average access reduction for register file, results bus, branch predictor, and other components are 27%, 43%, 45% and around 53% using *sequencer*, respectively.

●*Clock energy reduction*

Using MECIs and CRFU enhances the performance of the base processor, hence less clock cycles (consequently, less energy) is needed for executing an application. Fig. 7 shows the percentage of clock energy reduction using CRFU compared to the base processor without CRFU.

For higher clock frequencies, the base processor has to wait more clock cycles until the execution of a MECI finishes on the CRFU. The first bar for each application shows the energy saving percentage for clock when clock is gated while executing MECIs on the CRFU. Other bars show the energy saving for clock considering different frequencies in a case that clock gating is not applied but the base processor is stalled while the MECIs are executing on the CRFU (for *sequencer* invocation technique). By clock gating 10% more energy can be saved compared to a non-clock gating 300MHz frequency processor, in average (which is 42.6%).

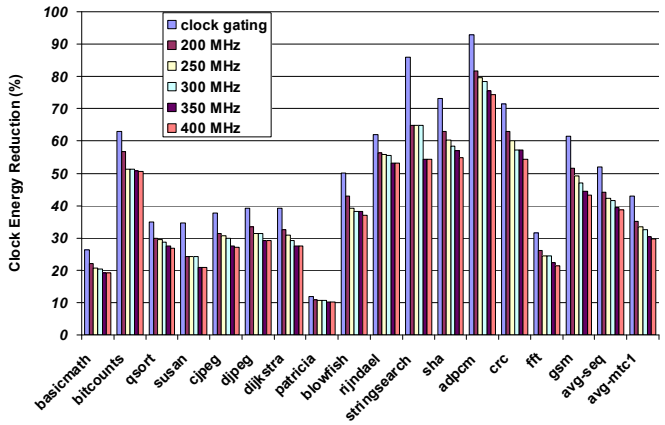

Figure 7. Clock energy reduction

●*Total energy reduction*

Fig. 8 shows the total energy reduction for the ADEXOR compared to the base processor for different clock frequencies and clock gating (for *sequencer*). The reason for the highest energy reduction by *adpcm* (79% for clock gating) is that it has a main loop with 56 instructions, including 12 branches. For 7 of these branches, both taken and not-taken are hot, so that 27% of branches are mispredicted. Therefore, a considerable percentage of executed clock cycles belong to the penalty of the mispredicted branches (18%). For those branches with both directions being hot, the MECIs include both directions, and hence, the CRFU architecture eliminates cycles related to mispredicted branches. Also, since HBBs are linked and longer MECIs are generated, more ILP can be extracted. For applications like *basicmath, susan, patricia,* and *fft* that most of the dynamic instructions are floating point, multiply, divide and load (69%, 45%, 44% and 57%, respectively), the energy reduction is less than average which is already expected since those instructions are not covered in MECIs. The average total energy reduction for clock gating is 42% and for 300MHz clock frequency is 37%. The average of total energy reduction for *mtc1* is almost 8% less than *sequencer*.

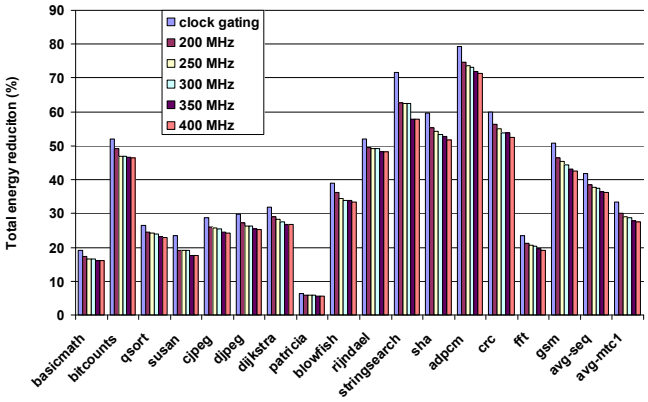

Figure 8. Total energy reduction

●*The effect of energy overhead on total energy reduction*

To study the effect of energy overhead on the total energy reduction, various values were assigned to the *OVH_FACT* in Eq. 6 and Eq. 7 as shown in Fig. 9. The energy consumption of the CRFU is 12.4 times of the configuration memory and the energy overhead of the *sequencer* is 1.7 times of *mtc1*, in average. The results show that for each 10x factor of *OVH_COEF* the average total energy reduction degrades almost by 1.3%, 0.8%, 2% and 1.2% considering the *sequencer* with clock gating, *mtc1* with clock gating, *sequencer* at 300MHz and *mtc1* at 300MHz, respectively. Although, the energy overhead of *sequencer* is more than *mtc1*, still it can save more energy, due to higher coverage of dynamic instructions. Even if the energy consumption of the base processor becomes 50 times smaller or the overhead energy becomes 50 times bigger, for 300MHz clock frequency, *sequencer* approach still can save almost 5% more energy. For the 100x factor this number decreases to 1%.

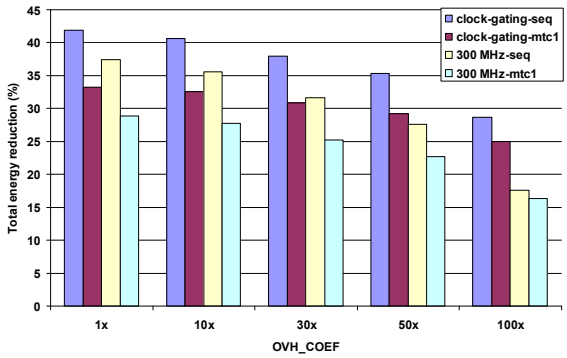

Figure 9. The effect of energy overhead on the total energy reduction

●*MECIs vs. CIs*

To see the effectiveness of MECIs compared to custom instructions limited to one HBB (named as CIs), we limited the MECI generator to one HBB and regenerated the CIs. Then, we redesigned the CRFU using the same quantitative approach. The number of inputs, outputs and FUs are the same as before, but it has simpler connections, less functions and FUs and does not support conditional execution. The area of CRFU reduces to 1.15 mm^2 and its delay for a CI with a critical length of five is 7.66 ns. The power also decreases to 206.85 mW. Each CI configuration needs 512 bits. Therefore, the energy for each access to the configuration memory reduces to 0.16 nJ. Fig. 10 shows the energy reduction obtained by MECIs and CIs compared to the base processor for some applications when the *mtc1* is used. The energy overhead for MECIs is 2.6 times of CIs in average. The results show that MECIs can save 15% and 12% more energy compared to CIs, correspondigly for clock gating and 300MHz clock. Even for an *OVH_FACT* equal to 100x the MECIs save 8% and 3.5% more energy compared to CIs for the cases of clock gating and 300MHz clock frequency, respectively.

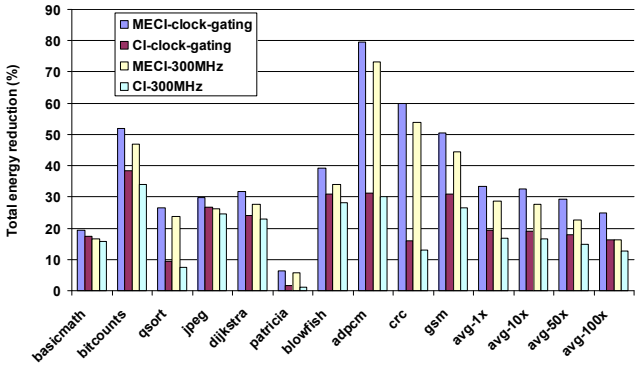

Fig. 10. Total energy reduction (MECIs vs. CIs)

6. Conclusions

In order to improve the energy efficiency of processor-based mobile computing systems, we proposed an architecture framework for an adaptive extensible processor to have the flexibility of general purpose processors while utilizing the energy reduction feature of custom instructions and maintaining the compatibility of the binary code. In this architecture, CIs are generated and added after chip fabrication. To support post-fabrication instruction set customization, the custom functional units are replaced by a coarse-grain reconfigurable functional unit. To cover higher percentage of dynamic instructions, custom instructions with multiple exits are proposed, instead of single exit. Using the proposed custom instructions, number of accesses (i.e. energy) to different components of the base processor (e.g. instruction cache, ALUs, branch predictor, result bus, decoder and register file) is reduced as well as the number of misses of instruction cache and clock energy. The total energy reduction is 42% when clock can be gated while executing MECIs on the CRFU in average and 37% for 300MHz clock frequency. Our experimental results show that by extending custom instructions over multiple HBBs (or using MECIs) the average energy saving increases by 15% compared to the custom instructions limited to only one HBB.

Acknowledgment

We would like to thank all members of the System LSI Laboratory of Kyushu University for their valuable comments during our technical meetings. This research was supported in part by Grant-in-Aid for Encouragement of Young Scientists (A) 17680005.

References

[1] J. Henkel, "A Low-Power Hardware/Software Partitioning Approach for Core-based Embedded Systems", *DAC* 1999.

[2] W. E. Dougherty, et al., "Subsetting behavioral intellectual property for low power ASIP design", *Journal of VLSI Signal Process.*, 1999.

[3] F. Sun et al, "Custom Instruction Synthesis for Extensible-Processor Platforms", *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 2004.

[4] P. Biswas et al. "Automatic Identification of Application-Specific Functional Units with Architecturally Visible Storage", *DATE 2006*.

[5] T. Sakurai, "Meeting with the forthcoming IC Design", Keynote Address, *ASP-DAC* 2007

[6] S. Wong et al., "Future Directions of Programmable and Reconfigurable Embedded Processors", *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, January 2004.

[7] M. Wan et al., "Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System", *Journal of VLSI Signal Processing*, 2001.

[8] G. Stitt et al., "Energy Savings and Speedups from Partitioning Critical Software Loops to Hardware in Embedded Systems", *ACM Transactions on Embedded Computing Systems*, February 2004.

[9] A. Lodi et al., “A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications”, *IEEE Journal of Solid-State Circuits*, vol. 38, no. 11, pp. 1876–1886, 2003.

[10] F. Barat et al., “Low-Power Coarse-Grained Reconfigurable Instruction Set Processor”, *FPL 2003*.

[11] M. Lanuzza et al., “Cost-Effective Low-Power Processor-In-Memory-based Reconfigurable Datapath for Multimedia Applications”, *ISLPED*, 2005.

[12] R. Lysecky et al., “A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning”, *DATE* 2005.

[13] K. Atasu et. al, “Automatic application-specific instruction-set extension under microarchitectural constraints”, *DAC* 2003.

[14] D. Goodwin et al., “Automatic generation of application specific processors”, *CASES 2003*.

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

[16] N. Clark et. al., “An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors”, *ISCA 2005*.

[17] H. Noori, et al., “Generating and Executing Multi-Exit Custom Instructions for an Adaptive Extensible Processor”, Design Automation and Test in Europe (DATE), 2007.

[18] D. Tarjan, et al., Cacti 4.0, HP Laboratories, Technical Report, 2006.

[19] H. Noori, et al., “Handling Control Data Flow Graphs for a Tightly Coupled Reconfigurable Accelerator”, International Conference on Embedded Software and Systems (ICESS-07), 2007.

[20] Mibench, www.eecs.umich.edu/mibench

[21] D. Brooks et al., “Wattch: a framework for architectural-level power analysis and optimizations”, *ISCA* 2000.

[22] C. Zhang, F. Vahid, and W. Najjar, “A Highly Configurable Cache Architecture for Embedded Systems,” ACM Transactions on Embedded Computing Systems, Vol. 4, No. 2, May 2005.

[23] O. Semenov, et al., “Burn-in Temperature Projections for Deep Sub-micro Technologies”, International Test Conference, 2003.

[24] Tohru Furuyama, **“**Challenges of Digital Consumer and Mobile SoC’s: More Moore Possible?”, Keynote Address, Design Automation and Test in Europe (DATE), 2007.