A Restricted Dynamically Reconfigurable Architecture for Low Power Processors

Takeshi Hirao Dahoo Kim Itaru Hida Tetsuya Asai Masato Motomura

Hokkaido University, Graduate School of Information Science and Technology,
Sapporo, Hokkaido, 060-0814, Japan

***Abstract*—Reconfigurable processors have widely attracted attention as an approach to realize high-performance and highly energy-efficient processors that map a target program's hot path to a reconfigurable datapath. In this paper, we propose a Control-Flow Driven Data-Flow Switching (CDDS) variable datapath architecture for embedded applications that demand extremely low power consumption in a wide range of uses. This architecture is characterized by following two features: (1) achieving both flexibility and low energy consumption by limiting the scope of the dynamic reconfiguration, (2) realizing smooth migration from the existing architecture by mapping the existing instruction sequence to the datapath. Preliminary evaluation on small programs have revealed that the CDDS accelerator achieves approximately 3 to 6 times the performance/power improvements, compared to a base processor.**

I. Introduction

In recent years, low power consumption has been one of the main criteria governing the design of various embedded processors, such as those for mobile devices, robotics, sensor networks, etc. At the same time, high-end processors have been struggling to improve their performance under severe total power budget constraints (*i.e.*, "power wall" problem). In essence, improving performance/power is a central theme in general-purpose processor architecture design today.

Multicore processors try to improve performance through parallel processing, *i.e.*, by increasing the number of processor cores. Since these multicore processors still keeps an underlying single processor architecture essentially unchanged, however, the performance/power is not expected to improve significantly. It is fairly well-known, in general purpose processor architecture, that ALU consumes around 10% of total power consumption, rest being consumed in fetching and decoding instructions, accessing register files, pipelining, etc. (see [1] for example.) That is, if non-ALU power consumption can be reduced with some intelligent architectural innovations, processors can drastically reduce their power consumption. As such, there have been various studies trying to improve performance/power of such general-purpose processors by integrating a customized accelerator which process hot path portions of applications.

Configurable processors use a fully-customized hardware accelerator, which is implemented (configured) prior to the fabrication of processor chips. Though a hardware accelerator can significantly improve the performance/power over that of a conventional processor, it is no more than a limited solution specific to target applications.

Reconfigurable processors address this concern by introducing reconfigurable accelerator which can be reprogrammed (reconfigured) before execution of different applications. Here, reconfigurable accelerator is generally composed of several memory elements, programmable interconnects and an array of processing elements (PEs). A key issue in such a reconfigurable accelerator design is to find a reasonable balance, prior to the fabrication time, between general versatility and performance/power improvement.

Intrinsic limitation of such reconfigurable processors is that they cannot accommodate a task that exceeds the hardware capacity of the reconfigurable datapath. Dynamically reconfigurable processors have been proposed to overcome this limitation. In this architecture, a task that exceeds hardware capacity of the reconfigurable datapath is first (at compile time) divided into multiple hardware contexts. These hardware contexts are then executed in time-division manner at execution time (*i.e.*, dynamic reconfiguration). Though dynamically reconfigurable processors can enhance general versatility in comparison to above "static" reconfigurable processors, potential drawbacks would be performance/power degradation. This is because frequent dynamic reconfiguration should cause additional power consumption which is not directly associated with the execution of a given task. A key issue here is, again, trade-off between general versatility and performance/power improvements.

This paper presents a new reconfigurable accelerator architecture targeting embedded processors. Since embedded processors are used in both control-intensive and data-intensive application domains, we need to pay careful attention to keep general versatility of the architecture. Needless to say, extremely low power and relatively high performance, *i.e.*, very good performance/power, is also a mandatory requirement for embedded processors. We named our architecture a Control-flow Driven Data-flow Switching (CDDS) variable datapath architecture: by limiting the dynamic reconfiguration for accommodating control-intensive tasks as minimum as possible, the CDDS architecture tries to achieve both versatility and performance/power improvement required for embedded processors.

Rest of the paper is organized as follows. Section 2 reviews recent related works in embedded processor domain. Section 3 introduces the CDDS architecture concept. Section 4 then describes the details of the CDDS processor, which integrates CDDS accelerator into a conventional processor. Section 5 discusses preliminary evaluation results on some small applications. Section 6 summarizes the paper.

II. Related work

Green Droid [2] is a configurable processor for mobile devices. It includes hardware accelerators for a several hot paths existent in the Android OS. It was reported that it reduces

978-1-4799-2079-2/13/$31.00 ©2013 IEEE

the energy consumption by 91% compared to a baseline CPU. Though the Android OS is one of major platforms for mobile devices, this solution suffers from the limitation mentioned in the previous section, namely, being difficult to deploy same approach toward variety of embedded applications.

ADRES [3] is a dynamically reconfigurable processor consisting of a VLIW processor and a reconfigurable matrix composed of function units. The performance could be improved by simultaneously using the reconfigurable hardware accelerator and the VLIW processor to execute the loop portion of the hot path. Since it relies on VLIW processor for control-flow handling, a program with multiple branches can not be processed in the reconfigurable matrix.

CMA [4] is a recent example of reconfigurable processors for low power applications. This architecture was proposed based on the observation regarding MuCCRA [5] and DRP [6], which are dynamically reconfigurable architectures, that they require additional power for dynamic reconfigurations: CMA PE array is composed of combinatorial circuits without registers, and is not dynamically reconfigured during execution, for saving power. In addition, PEs are combined using switches without passing through registers for saving additional power. Though this architecture provides an interesting design point for utilizing a reconfigurable accelerator for low-power processors, it looses general versatility quite a deal. The PE array can execute only a task that maps into a combinational circuit. If it needs to keep data in registers, such as in a loop execution, it has to use an external controller. In addition, since it cannot handle branch operations, it can process only a straight-line and short task. In essence, this architecture tries to achieve as much as high performance/power at the price of severely limited versatility.

In comparison to the studies described above, the CDDS architecture tries to find another and better balancing point between versatility and performance/power in embedded processing environment. It features dynamic reconfiguration at runtime for providing the flexibility required to handle control branches, which is inevitable for coping with reasonable range of embedded applications. On the other hand, it restricts the scope of the dynamic reconfiguration as much as possible for improving performance/power. When compared with conventional dynamically reconfigurable processors, since only a limited portion is actually reconfigured at runtime, additional power consumption due to reconfiguration can become dramatically small. When compared to "static" reconfigurable architectures such as CMA, on the other hand, it can execute even tasks consisting of a complex control flow.

III. Architecture Concept

Difficulties in mapping a control-intensive program to a reconfigurable accelerator lies in mapping complicated control/data flow graph (CDFG) to a datapath (an array of PEs and memories). Thus, first of all, it is worthwhile looking into the nature of CDFG. When a program is represented in a CDFG, a set of data flows to be actually executed is selected at a branching point in the control flow. Here, one can notice that the data flow actually affected at this point is the connection between the executed data flow and the selected data flow.

Often cases, dynamically reconfigurable architectures use their runtime reconfiguration as means to change hardware configurations at such a branching point [6]. This way, datapath structure (especially inter PE connections) can be kept

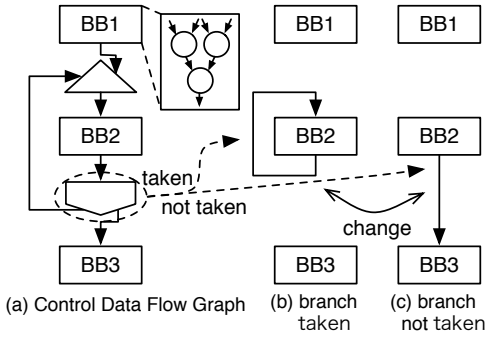

Fig. 1. Key observation: only inter-BB data flow need to be changed according to a branch in control flow.

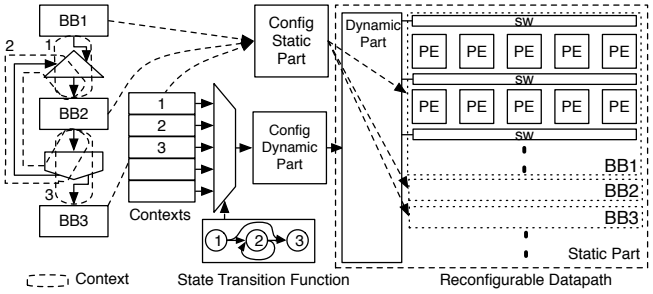

Fig. 2. The CDDS architecture concept: dividing a hot path into static part and dynamic part.

simple and shaped since complex control flow is handled by context switching mechanism built external to the datapath. Observation given above, however, seems to tell us it may be sufficient to change only the hardware configuration related to the data flow that is affected at the branching point, not the whole hardware configuration.

A. Control-Flow Driven Data-Flow Switching (CDDS)

This insight leads us to derive (design) a new architecture, where a reconfigurable datapath is divided into a static part and a dynamic part. The static part runs only combinatorially, depending on the pre-determined data dependency between instructions. Only the dynamic part is reconfigured at run time when a control flow branches are encountered. The name CDDS came from its intrinsic characteristic, where only the minimum portion of data flow switches under the supervision of a control flow. Since the scope of dynamic reconfiguration is quite limited, there is a good chance for this architecture to achieve both versatility and performance/power improvement fairly well.

Fig. 1 shows an example CDFG for illustration purpose. This program first runs from Basic Block 1 (BB1) to BB2. Since no branch exists in this period, data flows in BBs are not switched at runtime. Then depending on the result of the branch, data flows between BBs are switched. For example, when a branch is taken, the data flow from BB2 to BB2 is needed as shown in Fig. 1(b). On the other hand, if the branch is not taken, the data flow from BB2 to BB3 is required as shown in Fig. 1(c).

According to the concept explained above, this program is then divided and mapped into two parts in the CDDS

architecture: a static part and a dynamic part as shown in Fig. 2. All the intra-BB data flows are extracted and configured into the static part. Other inter-BB data flows are also extracted and configured into multiple contexts for the dynamic part. The dynamic part is reconfigured according to a context during execution of the task, and provides data to the BBs mapped on the static part according to the selected control flow. For example, context 2 and context 3 provides data connection from BB2 to BB2 and BB2 to BB3, respectively (see Figs. 1 and 2). Context switch is controlled by a state transition function generated from the CDFG.

Here, single branch generates two contexts (one for taken and the other for not taken). Since there is always an initial context, total number of contexts for a given task is 2n+1, where n is the number of branches in the task. This architecture can handle complex control sequence with branches as long as the maximum number of contexts is not exceeded. It is also possible to translate forward branches to a conditional execution sequence to reduce the number of required contexts (see next section).

Since there is no arithmetic unit in the dynamic part, the context does not include operational information. Therefore, the configuration bits of a context can be drastically reduced compared to those in conventional dynamically reconfigurable processors.

B. Configuration information generation

Since the CDDS architecture is geared toward integration within a conventional embedded processors, we need to consider portability of existing codes seriously. Unlike those previous works, which require writing a new program or modifying an existing program to generate the configuration information, the CDDS architecture can take over the program assets by generating configuration information based on the existing binary file. Additional advantage of using the binary file is that it requires only a light-weight design tool in generating configuration information.

For this purpose, PEs in Fig. 2 will be designed to have basically an identical ALU to the one in a main processor to be accelerated. Fig. 3 explains configuration information generation flow. First, a binary code generated by a compiler is disassembled, and then hot paths are extracted from the assembly code and the configuration information for both the static and dynamic part is generated out of it. The reconfigurable accelerator configuration (RAC) instruction is placed at the starting segment of the program, and the reconfigurable accelerator run (RAR) instruction is placed at the position of the hot path. When the main processor decodes RAC, the CDDS accelerator reads the configuration information, and maps it to the reconfigurable accelerator. Next, when the main processor decode RAR, the CDDS accelerator starts to run. Because the reconfigurable accelerator is configured when the processor starts up, no overhead is required for the configuration at execution time.

IV. The CDDS processor

Fig. 4 shows a proposed CDDS processor block diagram, which is composed of a main processor and the CDDS accelerator. The main processor is a slightly extended standard embedded processor. The CDDS accelerator is composed of a reconfigurable datapath with main and sub switch array, a context controller that changes contexts, and a configuration loader that reads the configuration information from instruction

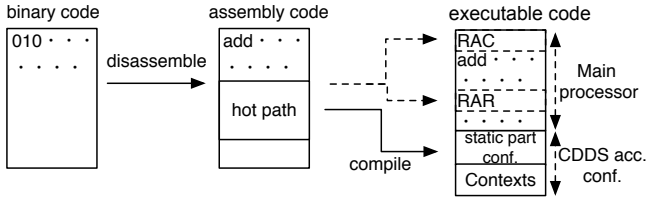

Fig. 3. Configuration information generation flow.

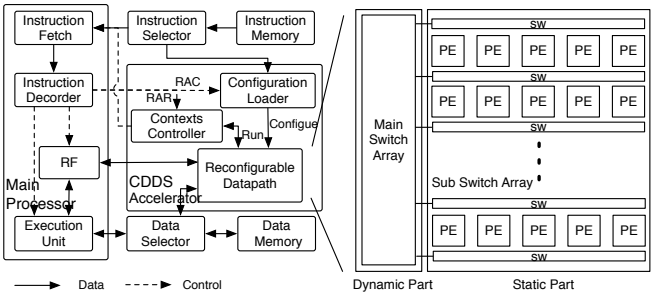

Fig. 4. The CDDS processor block diagram.

memory common to the main processor. The main switch array is the dynamic part mentioned in Section 3.1, and the sub switch array is the static part, as illustrated in Fig. 4.

When the main processor decodes RAC instruction, the configuration loader reads the configuration information, and configures the reconfigurable datapath. When the main processor decodes RAR instruction, the context controller set an initial context, and the CDDS accelerator starts to run. The CDDS accelerator processes hot paths in a program, and the main processor runs the rest of the program (the main processor is halted while the CDDS accelerator is running). Inputs and outputs of a task on the CDDS accelerator is read from (write into) the main processor via the register file (RF). It also read/write data from/to data memory through data selector.

A. Main switch array

Fig. 5 shows the main switch array, which provides reconfigurable connections between BBs configured in the sub switch array, the RF, and the temporary registers. It transfers data from the RF to the BBs configured in the sub switch array, and it also forwards the calculated results from a BB to other BBs. When the execution of a mapped task is completed, the CDDS accelerator returns the control to the main processor, and the calculated results are stored in the RF through the main switch array. When succeeding contexts need results calculated in previous contexts, those results should be temporarily stored somewhere upon a context switch. Thus, the main switch array is equipped with distributed temporary registers for this purpose (this is akin to register renaming in conventional processor architectures).

B. Sub switch array

As shown in Fig. 6, instruction sequence in each BB is mapped onto the sub switch array, where an instruction is mapped on a PE and data dependencies among instructions are translated into connections between corresponding PEs.

Fig. 7 shows the structure of sub switch array. The number of PEs in a stage defines the maximum number of instructions

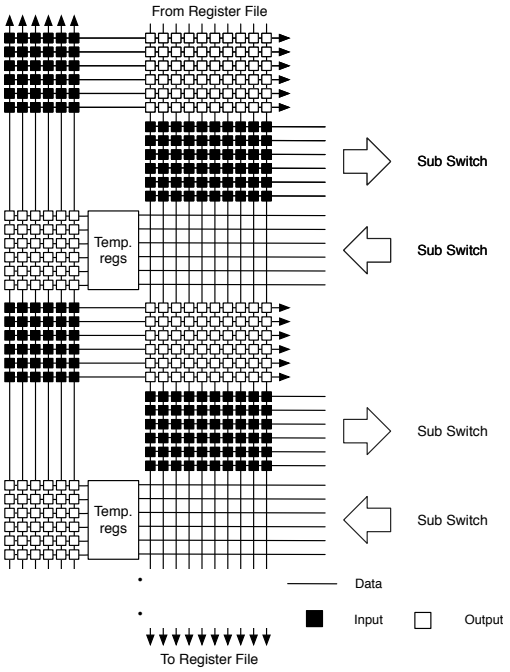

Fig. 5. Main switch array: configuring the connections between BBs configured in the sub switch array.

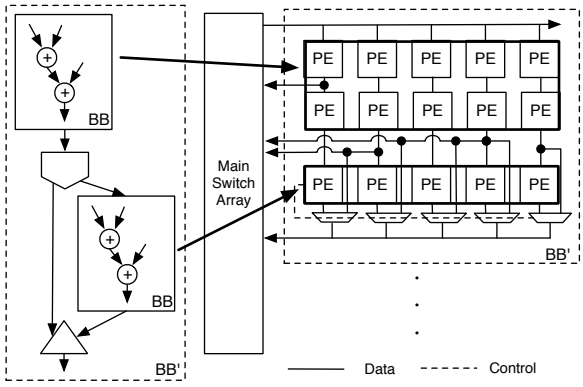

Fig. 6. Mapping BBs to the sub switch array.

that can be executed in parallel. In general, the instruction level parallelism of applications including many branches is known to be around five [7]; therefore, we set 5 PEs in a stage in this study. On the other hand, since instructions with data dependencies need to be mapped in different stages, the number of stages in the sub switch array set the maximum critical instruction path length of its executable task.

Since it is also expected that approximately one in five instructions would be a branch (BRA) or load(LD)/store(ST) instruction [7], we set left-most PE to include branch unit, and right-most PE to includeLD/ST unit, in addition to ALUs (the middle three PEs contains only ALUs).

If a PE in the stage requires the output of the PE in the previous stage, the input is connected to the output of the PE in the previous stage through the switches so that instructions can be executed in a chained manner. The inputs of each PE can be selected either data from the main switch array or output

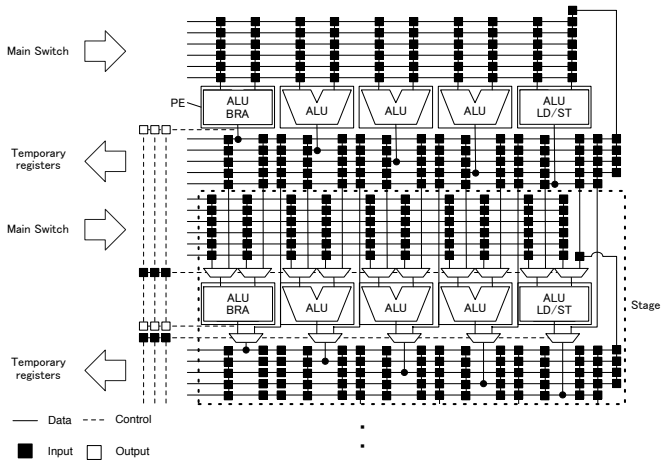

Fig. 7. Sub switch array: configuring the connections among PEs, and implements the data flows within BBs.

of the previous stage. LD/ST unit is an exception, in that it can get an output from the same stage. This is because it is highly likely that an ALU instructions mapped in the same stage generates a memory address for the LD/ST operation. Further, to avoid conflicts of multiple LD/ST operations, the LD/ST unit is selected sequentially, and only the selected unit can access the memory. The data from the data memory is stored in the external registers before switching contexts. The branch units output the conditions for context switching and conditional execution.

The sub switch array is provided with a mechanism to handle conditionally executions. A program generally includes multiple short forward branch instructions, and using contexts for these branches adds unnecessary overhead. By translating short forward branches into conditional executions in the sub switch array, the number of contexts can be reduced. For this purpose, the sub switch array is able to send branch unit's conditional results to subsequent stages. Also, selectors were added to the outputs of the PEs at each stage. Depending on the condition result, the PE could pass the output of the previous stage to the next stage, or it could pass its own output.

The PEs are linked as a chain based on dependencies among instructions. Because the maximum length of a chain for each context sets the execution time of the context, the context's execution time becomes variable. Therefore, an execution cycle count for a context, measured in fixed clock frequency, is different from a context to the other.

C. Context and state transition controllers

Fig. 8 shows a context controller, which controls the configuration of the main switch array. When the main processor decodes a RAC instruction, the configuration loader reads the configuration information from the instruction memory and stores it in the context memory. When the main processor decodes a RAR instruction, it stores the initial state specified by the instruction in the STATE register (Fig. 8). Then, the context is switched, and the datapath of the main switch array is reconfigured. When the datapath is configured, it forwards the RF data to the sub switch array, and the CDDS accelerator starts the computations. The execution times for each context is assigned to the context as the number of

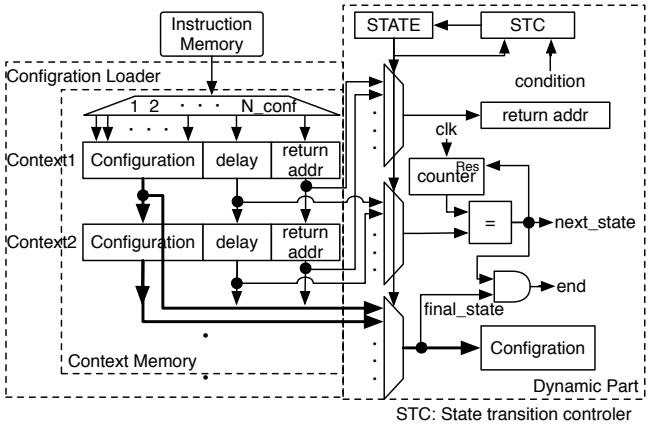

Fig. 8. Context controller: controlling the configuration of the main switch array.

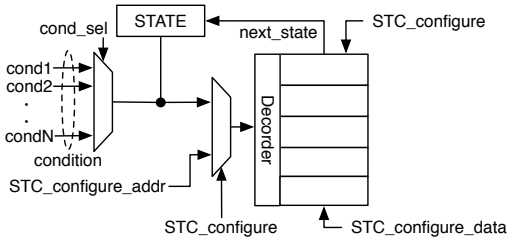

Fig. 9. State transition controller: determining the next state based on the current state which is stored in STATE register, and the conditional result.

cycles, and the accelerator runs for that length of time. During the execution time, the state transition controller (shown in Fig. 9, also STC in Fig. 8) determines the next state based on the current state which is stored in STATE register, and based on the condition result. The state transition function that determines the transition destination is stored in the memory. The conditional result is selected according to the context (cond_sel in Fig. 9) from outputs of the branch units of all stages. The context controller has the return addresses of the main processor, and stores this address in the program counter in the end of the final state.

V. Preliminary evaluation

We have designed a small test CDDS accelerator, conducted its performance and power estimation, then compared it with a main processor. First of all, Lattice Mico32 (LM32) [8] from Lattice Semiconductor, which employs a typical 32bit RISC architecture, is chosen for the main processor, because its RTL model is available as well as its capability for instruction extension. As for the CDDS accelerator, we set up 10 stages and 9 contexts. As explained in Section 4.B, it means this specific CDDS accelerator can handle a task up to 50 instructions, 10-instructions-long critical path, and 4 branches. Each stage of the main switch array has 5 temporal registers, and can forward 4 data to the sub switch array. Conditional execution was not implemented in this test design for simplicity.

We have conducted RTL design of the CDDS accelerator as well as LM32 (only minor modification), and also conducted their physical design on TSMC 0.18um CMOS

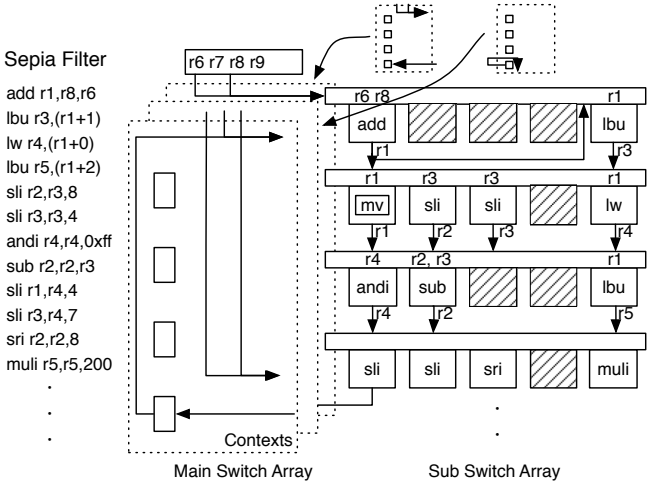

Fig. 10. Sepia filter mapped on the CDDS accelerator for evaluation.

process library using industry standard tools (see Table I). Expected operational clock frequency from post-layout timing analysis is 100MHz for both of them (see also Table I). As for instruction and data memories, we have set their size as 32-bit×4k-word (16KB) each, which is within a typical range for embedded processors. Since it is rather difficult to include memory access directly into post-layout power simulation, we have added memory access power consumption afterwords, which is calculated as 0.475mW/MHz (obtained from [9] for this size of memory) multiplied by the memory access average frequency obtained from the RTL simulation.

TABLE I. Tools and Design Results

Logic Synthesis		Synopsys Design Compiler
Layout/Post-layout Timing Analysis		Cadence Encounter
Post-layout Logic Simulation		Mentor Graphics ModelSim
Post-layout Power Estimation		Synopsys Design Compiler
Architecture	Gate count	Clock frequency [MHz]
CDDS Accelerator	561,279	100
LM32	30,459	100

A. Benchmark programs

As for benchmark programs, we have chosen three tiny codes, sepia filter, crc32, and sbox (a key function in AES), often used for preliminary evaluation of a new reconfigurable architecture. Binary codes for LM32 is compiled from respective C source codes by its software design environment. Configuration information for the CDDS accelerator is generated successfully, via manual translation (CDDS tool set is not available yet), from the disassembled LM32 binary codes.

For example, Fig. 10 shows how sepia filter is mapped on the CDDS accelerator. Instructions in an original code (shown left) are assigned to PEs in the sub switch array. Because the sepia filter contains one branch instruction, the number of contexts is 3. Hatched PEs are the PEs mapped with no instructions. Table II summarizes basic characteristics of these benchmarks. Here PE utilization, which is calculated as [the number of used PEs/the number of used stages/5], is ranged from 36% to 50%: which essentially says 1.8 to 2.5 instructions are executed in parallel (in a same stage) in the CDDS accelerator in this evaluation.

TABLE II. MAPPING RESULTS ON THE CDDS ACCELERATOR.

Application	# of inst.	# of branches	PE utilization [%]	# of contexts
sepia filter	22	1	44	3
crc32	18	2	36	5
sbox	25	2	50	5

B. Power consumption

Fig. 11 shows the average power consumption of the CDDS accelerator and LM32 when running each of the benchmark applications. In total, the CDDS accelerator consumes only 1/3 to 1/5 power compared to LM32. The main reason is in instruction memory, to which LM32 must access every cycle, while the CDDS accelerator does not at all during execution time. The difference in logic power consumption is relatively small. As a side note, both of the designs have been fully clock gated to conduct fair comparison.

Fig. 12 shows the LM32 and the CDDS accelerator power breakdown in circuit types for sepia filter for example (crc32 and sbox power breakdowns are quite similar to this one). In order to get more insights into the nature of logic power consumption, instruction/data memory accesses are excluded in this figure. In LM32 (a), the register portion, including RF, pipeine registers, etc., accounts for two times larger proportion than the combinational portion. In CDDS accelerator (b), on the other hand, the combinational portion accounted for 3/4 of the total power.

Fig. 13(a) and 13(b) show another breakdown of logic power consumption in functional blocks (again, excluding memory accesses) for the same application. LM32 power breakdown shows a typical characteristics of a general purpose processor. The CDDS accelerator consumes more than 70% of its power in its static part, while 6% is consumed in dynamic part. The clear difference between LM32 and the CDDS accelerator shown in Fig. 12 and Fig. 13 reveals that the CDDS architecture concept, i.e., executing a task statically (without using registers) as much as possible, is working quite successfully.

On the other hand, the CDDS accelerator power consumption during the configuration time for sepia filter is 59.5mW (47.5mW for instruction memory and 12mW for memory). Though it is relatively large, approximately twice the power of execution period, it is not a problem because the configuration is executed only once at the processor startup. For clarity, Fig. 13(c) additionally shows the power breakdown in functional blocks when this application is being configured onto the CDDS accelerator, where the percentage portion related to the configuration (configuration loader) is as large as 41%.

C. Performance and energy consumption

Fig. 14 shows performance improvement exhibited by the CDDS accelerator over LM32 (red line). It ranges from 1.4 to 2.2 depending on applications. Performance improvement in crc32 is relatively low since its instruction sequence is more heavily data dependent compared to the others (it also appeared in poorer PE utilization in Table II).

Needless to say, relative energy consumption is equal to relative power/performance, which is shown in green line in the same figure. The energy consumption of the CDDS accelerator is approximately 1/3 to 1/6 of that of LM32, showing that the CDDS accelerator has achieved the better performance/power in comparison with the main processor, as we have targeted.

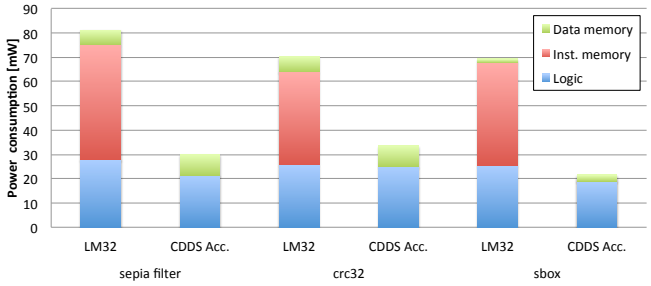

Fig. 11. Estimated power consumption for each application.

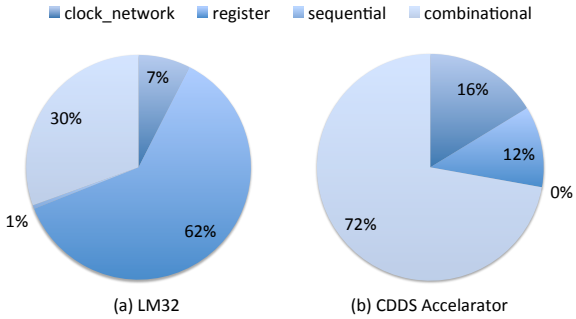

Fig. 12. Breakdown of power consumption for sepia filter.

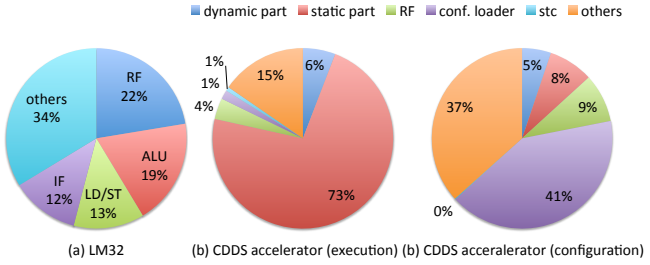

Fig. 13. Power consumption of each module for sepia filter. (a) LM32, (b) The CDDS accelerator in execution phase, and (c) The CDDS accelerator in configuration phase.

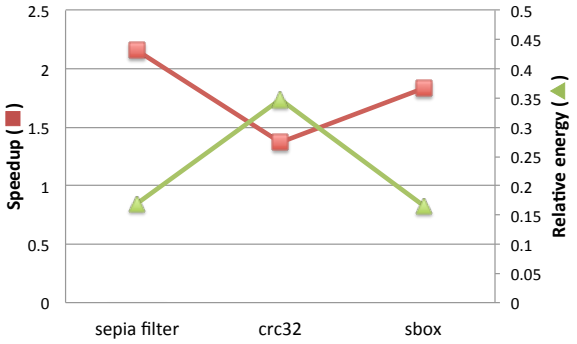

Fig. 14. Relative speedup and energy comparison between the CDDS accelerator and LM32.

VI. CONCLUSION AND FUTURE WORK

This paper has focused on an architectural study of a reconfigurable accelerator for low-power embedded processors, where maintaining versatility over applications and achieving better performance/power are the two major issues to be

addressed. We have claimed that exploiting dynamic reconfiguration in a restricted manner would be one way to successfully achieve both of them. CDDS architecture, which has been derived from this claim, is characterized by a reconfigurable datapath composed of static part and dynamic part, only the latter being dynamically reconfigured during execution time.

Our preliminary evaluation on a test CDDS accelerator design has shown promising results: it has shown approximately 3 to 6 times performance/power improvements on three benchmark programs. Though these programs are tiny ones, they include a few branches that can test versatility of this proposed architecture to some extent.

The proposed architecture is still in preliminary stage in various aspects: e.g., main/sub switch array structure design space need to be explored more systematically, power consumption need to be investigated into more detail to see potential in-efficiencies, testing on more realistic applications to get firm results, lacking a tool to generate CDDS configuration, etc. Our continued efforts on these and other aspects will be published elsewhere in future.

REFERENCES

[1] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz.: *Understanding sources of inefficiency in general-purpose chips*, *SIGARCH Comput. Archit. News*, 38(3):37–47, June 2010.

[2] S. Swanson and M.B. Taylor.: *Greendroid: Exploring the next evolution in smartphone application processors*, *Communications Magazine, IEEE*, 49(4):112 –119, april 2011.

[3] Francisco-Javier Veredas, Michael Scheppler, Will Moffat, and Bingfeng Mei.: *Custom implementation of the coarse-grained reconfigurable adres architecture for multimedia purposes*, In *FPL*, pages 106–111, 2005.

[4] N. Ozaki, Y. Yasuda, Y. Saito, D. Ikebuchi, M. Kimura, H. Amano, H. Nakamura, K. Usami, M. Namiki, and M. Kondo.: *Cool mega-arrays: Ultralow-power reconfigurable accelerator chips*, *Micro, IEEE*, 31(6):6 –18, nov.-dec. 2011.

[5] Yoshiki Saito, Toru Sano, Masaru Kato, Vasutan Tunbunheng, Yoshihiro Yasuda, and Hideharu Amano.: *A real chip evaluation of muccra-3: A low power dycamically reconfigurable processor array*, In *ERSA'09*, pages 283–286, 2009.

[6] Masato Motomura.: *A dynamically reconfigurable processor architecture*, *Microprocessor Forum, Oct. 2002*, 2002.

[7] David W. Wall.: *Limits of instruction-level parallelism*, *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):176–188, April. 1991.

[8] LatticeMico32 development tools, http://www.latticesemi.co.jp/products/designsoftware/micodevelopmenttools/index.cfm.

[9] 3.1 On-Chip SRAM http://www.csd.uoc.gr/~hy534/03a/s31_ram_bl.htm.