An Adaptive Dynamic Extensible Processor

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

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

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

出版情報：IEICE Technical Report, CPSY2005-29. 105 (453), pp.13-18, 2005-12. IEICE
バージョン：
権利関係：

社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE TECHNICAL REPORT
CPSY2005-29(2005-12)

An Adaptive Dynamic Extensible Processor

Hamid Noori Kazuaki Murakami and Koji Inoue

Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University, 6-1 Kasuga-koen, Kasuga, Fukuoka, 816-8580, Japan

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

Abstract This paper describes an approach for adaptive dynamic instruction set extension, tuning processors to specific applications. These new instructions are generated after production. The processor has two modes: training mode and normal mode. The application-specific instructions are extracted from the critical portions of the code detected by a profiler at training mode. At normal mode they are executed on a reconfigurable coarse grain accelerator. The sequencer decides when, which custom instruction should be executed. In this methodology there is no need to a new compiler and extra opcodes. Two methods are proposed for finding critical regions of the code.

Keyword Extensible Processor, Hot Spots, Online Profiling, Custom Instruction, Reconfigurable Accelerator

1. Introduction

Although availability of tools, programmability, and ability to rapidly deploy general purpose processors (GPPs) in embedded systems are good reasons for the common use of GPPs in embedded systems, they do not offer the necessary performance.

The application-specific nature of embedded system creates new opportunities to customize processor architecture for a particular application. Application specific instruction set processors, or ASIPs, have the potential to meet the challenging high-performance demands of embedded applications. The synthesis ASIPs traditionally involved the generation of a complete instruction set architecture (ISA) for the targeted application. However, this full-custom solution is too expensive and has long design turnaround times.

Another method for providing enhanced performance in processors is application-specific instruction set extension. By creating application-specific extensions to an instruction set, the critical portions of an application's dataflow graph (DFG) can be accelerated by mapping them to custom functional units. Though not as effective as ASICs, instruction set extensions improve performance and reduce energy consumption of processors. Instruction set extensions also maintain a degree of system programmability, which enables them to be utilized with more flexibility. The main problem with this method is that there are significant non-recurring engineering costs associated with implementing them. The addition of instruction set extensions to a baseline processor brings along with it many of the issues associated with designing a brand new processor in the first place.

In the design of embedded systems-on-chip, the success of a product generation depends on the efficiency and flexibility to accommodate future design changes. Flexibility allows system designs to be easily modified or enhanced in response to bugs, market shifts, evolution of standards, or user requirements, during the design cycle and even after production which also means increase in design productivity. Efficient implementations are required to meet the tight cost, timing, and power constraints present in embedded systems. Efficiency and flexibility are critical, but usually conflicting, design goals in embedded system design. While efficiency is obtained through custom hardwired implementations, flexibility is best provided through programmable implementations.

This paper presents a work-in-progress of adaptive dynamic extensible processor architecture. In this methodology the processor looks for the custom instructions (CIs) in the frequently executed portions of the code detected during online profiling. The processor has been developed by augmenting extra hardware to a single issue RISC processor and applying some modifications in the baseline processor architecture.

This paper is organized as follows: The general overview of processor architecture will be described in Section 2. In Sections 3 we propose our critical regions detector. Paper is closed by conclusion and future work.

2. General Overview of Processor Architecture

By *Adaptive* we mean that the processor can tune itself to the running applications. And we claim it is *Dynamic,* because instruction set extension is done after production

and even at run-time in the gap between two consecutive executions of the application (e.g. printers, cell phone and etc). Instructions set extensions are going to be done fully automatically and transparently.

2.1. Architecture

Our extensible processor has been designed and developed by adding/modifying some units to/of a general RISC processor. Figure 1 depicts the added/modified sections to/of the baseline processor, which has been proposed as a preliminary architecture for the extensible processor. Because it is a newborn project, details of the architecture still are not decided. The figure just tries to show a general view of the whole idea and concept.

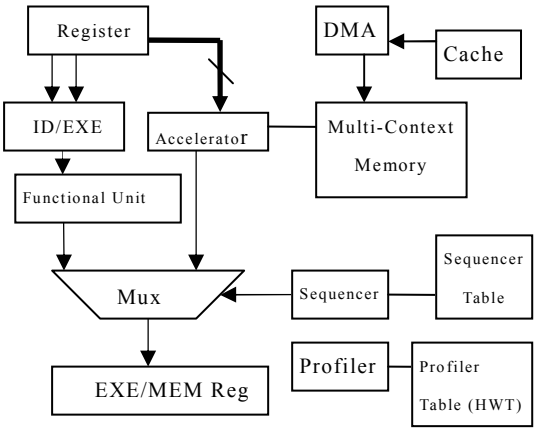

Fig. 1. General overview of the architecture

For simplicity, it has been assumed that the baseline processor is a single issue RISC processor. Firstly, we are going to evaluate the general idea. If it is practical then it will be merged it with more powerful embedded processors. There are three main units that have been augmented to the baseline processor: a profiler, a coarse grain reconfigurable accelerator and a sequencer.

This processor has two modes: training mode and normal mode. In training mode the processor learns about application-specific instructions and then generates the proper and necessary configuration data. In normal mode, it can still keep on learning but it can not generate new configuration. It just uses the configured architecture for running the application including the execution of CIs on the accelerator.

At the training mode, the processor starts profiling the applications. It tries to find critical regions of the code and then looks for the CIs in these hot regions. Some memories are required as a table for keeping the profiling information. This information will be used for generating CIs. Profiler has a simple hardware for monitoring and generating base knowledge about hot regions and more complicated software for handling the remaining tasks. More details about the profiler will be given in Sections 3 and 4.

As it has been also mentioned in [1], [3], according to the size of data in the processors, a matrix of ALUs seems efficient and reasonable enough for accelerating dataflow subgraphs as CIs. Using coarse grain reconfigurable accelerators demand for less configuration memory. Also mapping instructions on them will be easier. Although fine grain accelerators are more flexible, they are slower comparing coarse grain ones. We assume that each ALU of the accelerator supports all fixed-point instructions of the baseline processor except multiplication, division and load. It has also been presumed that at most one store and one control (branch or jump) instruction can be executed by the accelerator. Therefore the accelerator can change the program counter (PC).

For the preliminary performance evaluation, we suppose that, the accelerator does not have any limitation on inputs and outputs. It can have as many inputs as it needs and can write to register file as many as it requires. But there are some assumptions for the interconnect network. The outputs of ALUs in each row are fully connected just to the inputs of the ALUs in the subsequent row. Also at current phase, the depth and width of the accelerator have presumed to be infinite. Focusing on more details about the accelerator architecture will include in our future work.

The inputs of accelerator are directly connected to the outputs of the registers of the register file. Therefore, accelerator does not need to read the data from the register file. This idea has been used before in Chimaera [2]. The accelerator has a two-level configuration memory: a multi-context memory and a cache.

The multi-context memory can keep for example four configurations for the accelerator. Switching between configurations in multi-context memory will be very fast (within several clock cycles). The configurations of custom instructions that are most probable to be executed in near future are stored in the multi-context memory and the others in the cache. Using two-level configuration memory will hide the overhead of loading new configuration from cache to multi-context memory.

Utilizing a direct memory access (DMA) controller, loading multi-context memory from cache can be done in parallel with application execution. The number of

custom instructions is limited by the size of these memories. At current status there is no presumed value for size of these memories. Specifying proper size for memories and relation between multi-context memory, cache and DMA are left as future work.

The sequencer mainly determines the microcode execution sequence by selecting between the accelerator and the processor functional unit outputs. It has a table in which the start addresses of dataflow subgraphs, which are going to be executed as custom instructions on the accelerator, are specified. By comparing the PC and the contents of the table, the sequencer can distinguish which custom instruction will be executed at what time. Therefore the sequencer will know when it should switch between the outputs of the accelerator and the functional unit. Using this table, the accelerator can also decide when to switch between different contexts of multi-context memory and when to start loading new configuration from cache to the multi-context memory. The sequencer knows the required clock cycles for each CI execution. The table of the sequencer is initialized according to the locations of the custom instructions in the object code at the training mode, when they are generated.

One of the advantages of using the sequencer is that it obviates the need for adding new opcodes for the custom instructions. When training finishes and processor enters the normal mode, the sequencer starts monitoring the PC. When it reaches to the start address of a custom instruction, it switches from functional unit to the accelerator. Because the outputs of registers in register file are already available at the inputs of the accelerator, it does not need to read the registers. The sequencer will send the result of accelerator to the next stage after waiting for determined number of clock cycles (the custom instruction are assumed to be multi-cycle) and again switches to the output of functional unit. The required clock cycles for executing each custom instruction are determined at the training mode after mapping the custom instructions on the accelerator. These values will be used by sequencer.

The other advantage of the sequencer is the elimination of the penalty because of the absence of accelerator configuration data for the executing CI, in the multi-context memory. When the sequencer expects the execution of a CI in near future, it checks for the existence of the corresponding configuration data for the CI in the multi-context memory. If the configuration is available, the dataflow subgraph will be executed as a CI on the accelerator, otherwise the original code will be executed on the functional unit (the logical sequence of the binary code is not modified). Therefore in the case of absence of configuration of a custom instruction we will only miss the expected speedup for that instruction and there will be no penalty.

It has been assumed that CIs will take multi cycles to be executed on the accelerator. After mapping the CIs on the accelerator it will be known how many clock cycles it will be required for execution. Therefore we have supposed that we can have an accelerator with variable delay if we are able to read the output at the right time.

Using the proposed architecture and hardware obviate the need for developing a new compiler and adding new opcodes and instructions to the instruction set.

It could be also possible to utilize both the processor functional unit (PFU) and accelerator in parallel. Custom instructions (CIs) mostly need multi cycles for execution. By proper selection of custom instruction and start time of execution, it is possible to deploy the accelerator and PFU in parallel. In this case the performance can be enhanced more. We leave this issue for future work.

2.2. Functionality

We believe that it is fundamental to handle the instruction set extensions in a fully automated manner. On the other hand we are looking for a transparent post-production extensible embedded processor which does not require a new compiler. But automated instruction-set extension tools are time consuming so that they can not be applied dynamically while the processor is doing its duties. Therefore two modes have been defined for the processor: training mode and normal mode. In the training mode, the user will run desired applications with favorite inputs. The applications are executed on the processor using only the processor functional unit as usual. The system also starts online profiling. According to the information gathered by the profiler, critical regions are detected. More information about the profiler and critical regions detection will be given in following sections.

Custom instructions are generated utilizing frequently executed regions. Each CI can have at most one control (branch or jump) and one store instruction. The custom instruction can not contain multiply, divide, floating-point or load instructions. Then the control/dataflow graph (DFG for the first method and CDFG for second method) is generated for the detected

critical regions. In some cases, to make CIs larger, the sequence of instructions in the object code should be moved, which should not cause any change in the logic of the application. For example, sometimes by moving backward or forward some unsupported and independent instructions (e.g. load or divide) in the object code; larger CI could be created. In these cases we would need to do binary rewriting for some parts of the object code. Doing some dynamic optimizations on the instruction sequence such as constant propagation will reduce dataflow height and increase ILP.

For some embedded applications, there are some gaps between consecutive executions of the applications (e.g. printer and cell phones). In these systems the processor can keep on profiling even at normal mode. When the application execution finishes, it switches to training mode and applies the tools again to update or optimize more the custom instructions. For those embedded applications which do not have this feature, the processor enters the training mode at first once, and then it switches to the normal mode. In the training mode, also data for accelerator configuration memories are generated and loaded into the cache and multi-context memory. The sequencer table is initialized in this mode too.

After finishing CI generation in training mode the processor switches to the normal mode. In the normal mode, using the accelerator, its configuration data, the sequencer and its table, the CIs are executed on the accelerator. The sequencer monitors the PC and compares to its table entries. When it detects that a CI is going to be executed in near future, it checks whether the corresponding configuration is available in the multi-context memory or not. If it exists, the multi-context memory selects the proper configuration; the sequencer switches from processor functional unit to the accelerator, waits for specified clock cycles and let the accelerator finishes the execution of the custom instruction, then switches again to the processor functional unit. If the configuration is not available in the multi-context memory and there is enough time, the configuration data will be loaded to multi-context memory from cache. Otherwise the original code will be executed on the processor functional unit as usual. Each CI generates proper PC after its execution finishes, considering original sequence execution, so that processor can continue from correct address.

All the processes that are done in training mode can be done statically too. Therefore we can propose two methodologies, in one methodology, the processor has a training mode, so it needs some hardware for profiling and in other method training mode can be performed completely at static time. In this case, the processor does not need extra hardware for profiling. In this method the applications will be run using a simulator (e.g. simplescalar toolset) at static time then the profiler and remaining processes will be executed. The data for configuration memories and sequencer table will be generated statically. At the time the processor wants to start, first the generated data should be loaded to the configuration memories and sequencer table, then the processor starts its job.

3. Detecting Critical Regions of the Code

In the first method, CIs are extracted from hot basic blocks (HBBs). A basic block is a sequence of instructions that ends in a control instruction. Therefore the basic block has one branch or jump instruction which is the last instruction. HBBs are basic blocks that are executed more than a specified threshold. Start address of HBBs are determined by the profiler hardware monitoring the PC. In every clock cycle, the profiler hardware compares the current value and the previous value of PC. If the difference of these two values is greater than the instruction length, a taken branch or jump has occurred. The profiler hardware has a table with a counter for each entry. In the case of taken branch/jump detection by profiler hardware, the profiler table is checked. If the target address (current PC) is in the table, the corresponding counter is incremented, otherwise current PC is added as a new entry and the corresponding counter is initialized to one. In our future work we are looking for hardware/software implementation of this profiler with very low overhead and determining a proper size for the table.

After application execution and profiling finish, the profiler software reviews the table and selects the entries (target addresses) with counters more than a specified threshold. These addresses determine the start address of HBBs. Using these addresses, the HBBs are read from the object code. Reading an HBB is terminated as a control instruction is seen. In this method, if the branch instruction of hot basic block is mostly no taken (more than the threshold), the following basic block of this HBB will be also hot, but by looking just to the profiler table it can not be detected.

To solve this issue, after detecting the start addresses

of the HBBs and reading the HBBs from the object code, their last instructions are checked. If the last instruction is branch (not jump), the branch target address, the counter of the current HBB and the start address of not taken part are saved in a new list. The counter shows that how many times the branch has been executed. Jump instructions are always taken, so their target can be detected by looking into the table. Therefore we have to check just the branches to see if the not taken direction is also hot or not.

After saving these values for detected HBBs in the new list, all branch target addresses (BTAs) of the new list, are checked to see if they are in the HBB list or not. If a BTA is in the current HBB list, then it is ignored. Otherwise the branch target address of the new list is searched in the profiler table. If the BTA of the new list can be found in the profiler table, then the counter of the profiler table is subtracted from the corresponding counter of the BTA of the new list. The counter of the profiler table shows how many times the branch is taken and the counter of the new list shows how many times the branch instruction of the HBB is executed. By comparing the result of subtraction to the threshold value it can be distinguished if the not taken direction is hot or not. If it is hot, the not taken start address is added to the HBB list as a new entry otherwise it is ignored. If the BTA of the new list can not be found in the profiler table, it means that this branch is always not taken which means that the not taken part is hot. In this case, the not taken start address is added to the HBB list as a new entry. This algorithm is run again for every new HBB entry and continues until no new HBB is found. All of these tasks are done by profiler software.

3.1. Preliminary Performance Evaluation

To do a preliminary performance evaluation, Simplescalar tool set (PISA configuration) [5] and Mibench [4] (a free, commercially representative embedded benchmark suite) have been used. We use the sample inputs for the benchmarks. The sim-safe tool of Simplescalar was modified to generate the sequence of PCs of the retired instructions. The output of the modified sim-safe is applied to our profiler, in which PCs are monitored and the profiler table is created. Using the profiler table, the start addresses of HBBs are detected. The HBBs are read from the object code; CIs are extracted from them and mapped on the accelerator. At this phase, the custom instruction selection and mapping is done manually.

To calculate speedup, we use a single-issue, in-order pipelined architecture with 100% cache hit rate. Each instruction is supposed to be executed in one clock cycle.

As it has been mentioned before, the accelerator has been assumed to have a variable delay according to the height of mapped custom instruction. It has been presumed that the first row of the accelerator takes one clock cycle and the other rows, which do not have register read and write take 0.5 clock cycle for execution. For example, suppose that there is a CI containing nine instructions. After mapping this CI on the accelerator, it takes three rows of the accelerator. The first row takes one clock cycle and the second and third rows take 0.5 clock cycle. Therefore it takes two clock cycles for the CI to be executed on the accelerator. For N times execution of this CI; (9-2)*N clock cycles will be saved. Using these assumptions, preliminary performance evaluation was done.

In figure 2 the percentage of CIs for different length is shown. The numbers which appear after the applications' name show the threshold value used for selecting HBBs. Due to the results; in most cases more than 70% of CIs have less than 6 instructions. The longer the CI, the more performance enhancement can be obtained. Therefore we should look for some methods to be able to increase the CI's length.

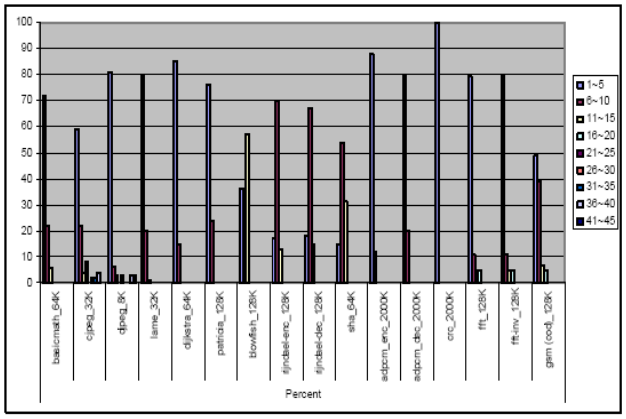

Fig. 2. Percentage of custom instruction according to their length

Table 1 illustrates more information about the CIs and performance enhancement. The second column shows the number of executed and profiled instructions. Third column contains threshold values for selecting hot basic blocks. In the next column, number of detected HBBs has been written. The numbers in the fifth column specifies how many CIs could be extracted from the HBBs.

Table 1. Information about custom instructions of Mibench

App.	Exe Instr (M)	Threshold (K)	No. HBB	No. CI	% Speedup	% code size	% exec time
basicmath	170	64	37	18	19.6	1.4	31.6
cjpeg	101	32	42	52	27	1.5	44
djpeg	25	8	22	32	31.5	0.8	48
lame	260	32	142	104	8.6	1.1	16
dijkstra	254	64	34	20	21.4	0.7	38.6
patricia	217	128	51	17	7.8	0.6	14.6
blowfish	260	128	18	28	33	2.7	59
rijndael (enc)	260	128	63	92	36	6.1	51.7
rijndael (dec)	259	128	63	78	36	4.5	51.7
sha	154	64	9	13	52	1.1	73
adpcm (enc)	260	2000	14	8	21	0.32	42
adpcm (dec)	265	2000	12	5	24	0.24	41
crc	265	512	4	2	20	0.1	44.9
fft	189	128	43	19	18.6	0.93	30
fft (inv)	190	128	43	19	18.6	0.93	30
gsm (cod)	265	128	34	41	25.1	1.53	47.2

Sometimes, HBBs are very long so that several custom instructions can be extracted from one hot basic block (e.g. in JPEG) and sometimes hot basic blocks are too small or contain unsupported instructions so that no custom instruction can be extracted (e.g. basicmath).The sixth column shows the speedup versus the baseline processor. As it was expected, applications such as sha, gsm, and rijndael that have longer CIs could reach better performance. The last two columns show the percentage of the code size and execution time covered by the custom instructions, respectively. As it can be seen, a very small part of the object code is executed for many clock cycles.

4. Conclusions and Future Work

We have presented a general overview of a novel architecture for an adaptive dynamic extensible processor. This processor is capable to add application-specific instructions to its instruction set after production.

The architecture is based on a single issue RISC processor. A profiler, a reconfigurable accelerator and a sequencer has been augmented to the baseline processor and some modifications have to be applied to the register file and pipeline intermediate registers.

This processor has two modes: training mode and normal mode. In training mode, using the profiler, it learns about CIs and generates the configuration data for its accelerator and initializes its sequencer table. In normal mode using the information generated in training mode, tries to run the CI on the accelerator. It does not need any new compiler and new opcodes for the extended instructions. CIs are extracted from critical regions of code. The preliminary valuation results speedup between 7.8% and 52% for some of Mibench programs.

Looking for a simple hardware implementation of profiler, developing tools for CDFG generation, HIS extraction and mapping custom instruction on the accelerator are some examples. Also more details about the architecture and functionality of modules such as accelerator and sequencer should be specified. The details of connection of accelerator to the register file are missing. The custom instructions have many inputs and need also many register file writing. These issues should also be handled. However, [6] shows that 4-input, 3-output patterns achieve close to maximal speedup. Executing primitive instructions and CIs on processor functional unit and accelerator in parallel can also be considered as another task. Determining the threshold value should be done dynamically. Finally a complete simulator framework is high required.

Acknowledgement

This research was supported in part by grant of the Cooperative Link of Unique Science and technology for Economy Revitalization (CLUSTER) of Ministry of Education, Culture, Sports, Science and Technology (MEXT) and Grant-in-Aid for Encouragement of Young Scientists (A), 17680005.

Reference

[1] N. Clark, et al., Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. MICRO-37, 2004.

[2] Z.A. Ye, et al., Chimaera: a high-performance architecture with tightly-coupled reconfigurable functional unit. 27th ISCA, pages 225-235, 2000

[3] N. Clark, et al., An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors. ISAC-32, 2005.

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

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

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