Preliminary Performance Evaluation of an Adaptive Dynamic Extensible Processor for Embedded Applications

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

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

Preliminary Performance Evaluation of an Adaptive Dynamic Extensible Processor for Embedded Applications

Hamid Noori
Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University
6-1 Kasuga-koen, Kasuga, Fukuoka
816-8580 Japan
+81-92-583-7620
noori@c.csce.kyushu-u.ac.jp

Kazuaki Murakami
Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University
6-1 Kasuga-koen, Kasuga, Fukuoka
816-8580 Japan
+81-92-583-7620
murakami@i.kyushu-u.ac.jp

ABSTRACT

In this research we investigate an approach for adaptive dynamic instruction set extension, tuning processors to specific applications after fabrication.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other architecture styles – *adaptable architectures.*

General Terms

Performance, Design, Experimentation.

Keywords

Adaptive Dynamic Processor, Instruction Set Extension, Online Profiling, Hot basic Block, Custom Instruction.

1. INTRODUCTION

This research describes an approach for adaptive dynamic instruction set extension, tuning processors to specific applications. The processor has two modes: training mode and normal mode. New instructions are detected and added after production at a preliminary mode (training mode). In this methodology there is no need to a new compiler and extra opcodes for extended instructions. The application-specific custom instructions are extracted from the frequently executed parts of the code (hot basic blocks) at training mode. The custom instructions are executed on a reconfigurable coarse grain accelerator at normal mode. Hot basic blocks are detected by a simple hardware (profiler) which monitors the Program Counter (PC) of the processor.

2. GENERAL OVERVIEW OF THE 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).

The processor has been designed and developed by modifying a general single-issue, in-order RISC processor. There are three main units that have been augmented to the base processor: a profiler, a coarse grain reconfigurable accelerator and an arbiter.

The processor has two modes: training mode and normal mode. In training mode using the profiler the processor learns about application-specific custom instructions and then generates the proper and necessary configuration data. The processor tries to find hot basic blocks (HBBs) and then looks for the custom instructions in these critical regions. In the 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 custom instructions on the accelerator.

The reconfigurable coarse grain accelerator is a matrix of ALUs. We assume that each ALU of the accelerator can support 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 instruction can be executed by the accelerator. The accelerator does not support floating-point instructions. The outputs of ALUs in each row are fully connected just to the inputs of the ALUs in the subsequent row. The inputs of accelerator are directly connected to the outputs of the registers of the register file, so it does not need to read the registers from register file. This technique has also been used in Chimaera[1]. The accelerator has a two-level configuration memory: a multi-context memory and a cache. Multi-context memory can keep several (four or eight) contexts. Switching between different contexts is very fast (usually takes several cycles). The configuration of custom instructions which are most probable to be executed in near future are kept in the multi-context memory and the rest are saved in the cache. The required configurations are loaded from multi-context memory to the cache using a direct memory access controller (DMAC).

The role of the arbiter is that to select between the accelerator and the processor functional unit outputs. The arbiter 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 arbiter can predict which custom instruction will be executed at what time. It can also determine when to load new configuration from cache to multi-context memory and when to switch between different contexts of the multi-context memory. It has been assumed that custom instructions will take multi cycles to be executed on the accelerator. The accelerator has been supposed to have variable delay according to the number of rows which is needed by the mapped custom instruction.

3. HOT BASIC BLOCK DETECTION

A basic block is a sequence of instructions that terminates in a control instruction. HBBs are basic blocks that are executed more than a specified threshold. The custom instructions are extracted using these hot basic blocks. Start address of hot basic blocks are determined by the profiler monitoring the program counter (PC). In each cycle, the profiler compares the current value and the previous value of PC. If the difference of these two values is greater than the instruction length, it reveals that a taken branch or jump has occurred. The profiler has a table. For each entry of the table there is a counter. When the difference of current PC and previous PC is greater than one instruction length, 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. This method can not detect those HBBs that follow mostly not taken branches. To solve this issue we made following changes to software routines.

In the new algorithm after detecting the start addresses of the HBBs and generating the table similar to the previous method, the hot basic blocks are read from the object code. Control instructions terminate reading HBBs. For each HBB, its last instruction which is a control instruction is checked. If the last instruction is branch, the branch target address is detected and saved. The counter of the current HBB and the start address of not taken part are also saved. This counter shows that how many times this branch has been executed. Because jump instructions are always taken, they can be detected by looking into the table. Therefore we have to check just the branch instructions to see which direction of branch (taken, not taken or both) is hot.

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 the BTA is in the current HBB list, then it is ignored. Otherwise the BTAs of the new list are 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 HBB 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 HBB. This process is executed again for every new HBB and continues until no new HBB is found.

The maximum number of detected basic blocks for utilized applications of Mibench[2] is 2149. This number also shows the required entry for profiler table. By using replacement policies which replace low frequent basic block's start addresses with higher frequent ones in the profiler table, smaller profiler table is needed. For the applications we used for evaluation, the maximum number of basic blocks that are executed more than 512 (this number is much smaller than floor threshold for detecting hot basic blocks) is 459. Therefore a profiler table with 512 entries seems enough and suitable for our HBB detector.

4. PRELIMINARY PERFORMANCE EVALUATION

To do a preliminary performance evaluation of the architecture, the dataflow graph (DFG) is generated for each detected hot basic block. Custom instructions are extracted from the DFGs. Each custom instruction can have at most one control instruction (branch or jump) and one store instruction. The custom instruction can not contain multiply, divide, floating-point or load instructions.

Simplescalar[3] tool set (PISA configuration) and Mibench benchmark have been used for the experimental setup. The sim-safe tool of Simplescalar was modified to generate the sequence of PCs of the committed instructions. The output of the modified sim-safe is applied to our tool, in which PCs are monitored and the profiler table is created. Using the profiler table, the start addresses of hot basic blocks are detected. The hot basic blocks are read from the object code. After reading hot basic blocks from object code and generating the DFG, the custom instructions are determined and mapped on the accelerator. To be able to make larger custom instructions, sometimes it is necessary to move the unsupported instructions up or down. Although this change does not modify the logic of the application, a new object code should be generated.

At the current state the number of inputs, number of outputs and width and depth of the accelerator have assumed to be infinite. The accelerator has been supposed to have a variable delay. Because only the first and last row of accelerator need to read and write to register file, it has been presumed that the first row of the accelerator takes one clock cycle and the other rows take 0.5 clock cycle for execution. The result shows speedup ranges from 7.8% to 52% for the examined applications comparing the baseline processor.

5. REFERENCES

[1] Ye, Z. A., Moshovos A., Hauck S., and Banerjee P. Chimaera: a high-performance architecture with tightly-coupled reconfigurable functional unit. *In the proceeding of 27th International Symposium of Computer Architecture (ISCA)*, 2000, pages 225-235.

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

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