An Online Profiling-Based Dynamically Adaptable Processor

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

Yoshimatsu, Norifumi
Fukuoka Industry, Science & Technology Foundation

Fujii, Yousuke
Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University

Eshima, Kazuhito
Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University

他

出版情報：The Proceedings of the 11th International CSI Conference, pp.520-523, 2006-01. Computer Society of Iran
バージョン：
権利関係：

An Online Profiling-Based Dynamically Adaptable Processor

Hamid Noori†, Yoshimatsu Norifumi††, Yousuke Fujii†, Kazuhito Eshima†, Makoto Yoshida††, Takeshi Soga††, Takanori Hayashida† and Kazuaki Murakami†

†Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University, 6-1 Kasuga-koen, Kasuga, Fukuoka, 816-8580, Japan
{noori, fujii, eshima, hayashida}@c.csce.kyushu-u.ac.jp
murakami@i.kyushu-u.ac.jp

†† Fukuoka Industry, Science & Technology Foundation
FLEETS (Fukuoka Laboratory for Emerging & Enabling Technology of SOC), 3-8-33, Momochihama, Sawara-ku, Fukuoka, 814-0001, Japan
{nyoshimatsu, yoshida, soga}@fleets.jp

ABSTRACT

This paper investigates a possible architecture to a dynamically adaptable processor. In this architecture, the running application is profiled and dynamic traces of high frequently executed loops (hot paths) are detected. The proposed online profiling methodology is mainly hardware-based so that overhead can be reduced as much as possible. Studying the behavior of branch and jump instructions, gathered by the profiler, guides us to the hot paths. To improve the performance for the next iterations, hot paths are optimized using dynamic software pipelining technique, which seems a suitable method for our simplified 8-way VLIW accelerator. To exploit the hardware accelerator, the binary code is rewritten. Some preliminary performance evaluations show speedup.

Keywords

Adaptive dynamic optimization, online profiling, VLIW accelerator, dynamic software pipelining.

1. INTRODUCTION

Dynamically linked libraries, micro-architecture specific features, inaccurate run-time profiles and using object-oriented language and techniques have limited the size of the scope available for static compiler analysis. Also dynamic optimization offers more opportunities for those applications whose behavior are input dependent. More recently, the use of dynamic code generation environments makes the applicability of heavyweight static compiler optimization techniques impractical. Meanwhile on the hardware side, progress of semiconductor technology enabled to design a large scale, complex functionality SoCs. As the complexity increases it is getting difficult to foresee circumstances which a system operates at its design time and it makes challenging to design a system optimizing performance, power consumption, and energy consumption for them.

Shifting optimizations to runtime can be a possible solution for these problems. Generally, in dynamic optimization, a binary program is executed on a processor system. The system monitors the executing binary, detects the frequently executed regions, optimizes those regions and exchange future occurrences of the original regions with the optimized version. The main processor or some extra hardware accelerator can be used for running the optimized traces. Recent work in dynamic optimization has shown that a run-time system can improve program performance and power consumption by performing optimizations that are difficult to deploy statically [4], [5], [6], [7], [8].

However there are some issues to realize the dynamic optimization in a system. It must have small impact on system's operation. It should be identified, where and how the optimizations are going to be applied. Also the system has to be able to update the executing optimized code, adaptively. The overhead of detection and optimization must be amortized by gaining better performance. This limits the scope of optimizations that can be done online, and makes the efficiency of the optimization infrastructure extremely critical. To overcome the overhead, there can be several solutions according to the application. For some applications that there is a gap between consecutive execution of the application, e.g. printer and mobile applications, the dynamic optimization can be done in this gap. For those applications without this gap, a training phase can be defined before the normal use. In the training phase, the system learns about the hot paths of the application, applies optimizations and prepares for executing optimized code for the next iterations. Using a hardware engine for optimization can be another solution.

This paper, proposes SysteMorph as a concept of a system which can be dynamically optimized. It optimizes systems dynamically and executes application programs adaptively based on application program's online profile information for processor based systems. SysteMorph consists of three elemental technologies those are (1) online profiling, (2) dynamic adaptive optimization, and (3) smart hardware.

The concept of SysteMorph is described in Section 2. Section 3 discusses the SysteMorph online hotpath profiler. In Section 4 proposed SoC architecture of SysteMorph is described. Some preliminary performance evaluations are shown in Section 5 and the paper is closed with conclusion and future work.

2. SYSTEMORPH CONCEPT

SysteMorph is a feedback directed dynamic optimization technology mainly adapted in processor based systems (Fig. 1). With SysteMorph, a system can be optimized in terms of performance, power consumption, and energy consumption using target application program's run-time profile information. SysteMorph technology consists of the following three elemental technologies: Online profiling, Adaptive dynamic optimization and Smart hardware.

The online profiler collects run-time information at binary level and exploits it for optimization/acceleration. The information is sent to the optimizer as feedback. The profiling information helps the system to detect, e.g. threads, a hot instruction sequence, hot loops and etc.

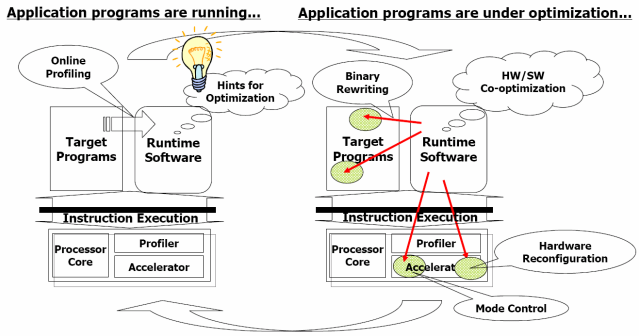

Figure 1. SysteMorph Concept

The adaptive dynamic optimization is a technology to optimize the system dynamically including software and hardware based on the profile information. It can cover various optimizations for hot traces, fusion of instructions (for dynamic reconfigurable processors), reconfiguring the reconfigurable parts of the system and etc. After optimizations, optimized binary code and modified configurations are rewritten. Dynamic optimization can be done at different periods. For example, the system can have a training phase before starting its normal operation. In the training phase, the user can apply desired inputs and let the system learn about the hot paths, optimize them and exchange the code for exploiting the accelerator. In the other case, if there are any gaps between the consecutive executions of the application, dynamic optimization can be applied at those periods. In the these cases, the overhead of dynamic optimization can be ignored.

The smart hardware technology is a hardware configuration technology to perform hardware reconfiguration or mode change to apply the adaptive dynamic optimization. Basically it contains a general purpose or dynamic reconfigurable processor, some hardware for profiler and accelerator.

3. ONLINE HOT PATH PROFILER

The online profiling is a technology to collect information of application program's behaviors at run-time, while off-line profiling collects whole information regarding to application program's execution after running the application program. The issue of the online profiling is how to get precise profile while the profiling does not affect execution of an application program. The advantage of online profiling is that it can get a profile for a running application program which behavior depends on its inputs.

3.1 Branch History Hot Path Profiler

Any serial code can be considered as many linked basic blocks. A basic block is a sequence of instructions ending in a control instruction. All instructions except the last instruction are non-control instructions. Our online profiler finds dynamic traces of basic blocks of loops. The online profiler has a hardware profiler assistant and hot path finder software routine. The profiler assistant monitors the executed instructions. When it encounters a branch, it stores the address of the branch and its target address in the branch history table (which is a buffer that can be a FIFO or a CAM). The branch history table has four fields: basic block start address, branch instruction address, branch target address and executed count. The last field shows, the execution frequency of basic block. When the buffer is filled, the main processor is interrupted.

The interrupt routine, containing the hot path finder, starts analyzing the branch history table and find closed paths by dynamically linking the basic blocks. The closed paths will be considered as a hot path if they have been executed more than a defined threshold and their length is less than a maximum length. Using a hardware assistant for profiler helps to reduce the overhead of profiling and makes dynamic optimizations more effective. Hot paths will be passed to the optimizer routine for optimizations.

Because whole hot path is used to apply optimization taking advantage of loop execution, precise hot path prediction is crucial. BH (branch history) hot path profiler is an online profiling method that can provide hot path prediction based on each branch instructions history.

To detect hot path(s) BH profiler collects address information on each executed branch instructions as a tuple. The tuple consists of branch instruction address (BIA), branch target address (BTA), and basic block start address (BSA). The tuple is stored in the branch history table. (Fig. 2). When a branch instruction is executed a tuple is constructed then compared with tuples stored in the branch history table. If there is a hit, associated count is incremented otherwise the new tuple is written in the branch history table. Also a separated table called backward branch history table keeps branch target address (BTA) only for backward branches.

BSA	BIA	BTA	count
...	...	...	...

BTA	count
...	...

Figure 2. Branch history table and Backward branch history table

When a count in an entry of the backward branch history table exceeds a threshold value, that BTA entry is picked as a start address of a hot path. The branch history table is referred to find hot path sequence. BSA addresses in the branch history table are compared with BTA. If matched BSA is found, BTA in the same entry is picked to search next BSA. If multiple BSAs are found counts are compared and BSA which has bigger count is picked. Finally if BTA is smaller than BSA and the BTA is matched the hot path head address, current sequence BTA is identified as a hot path. By using BSA to index branch history table, the search is simplified.

4. SYSTEMORPH UTILIZING A VLIW ACCELERATOR AND TRACE BASED DYNAMICALLY SOFTWARE PIPELINING

Fig. 3 illustrates the proposed architecture for implementing SysteMorph. It contains a main processor, online profiler, optimizer, binary rewriter and an accelerator. The main processor is a simple RISC processor with 32 32-bit registers. The application is run on the main processor.

The optimizer routine uses Trace based Dynamic Software Pipelining (TSWP) technique for optimizing hot paths. Software pipelining has been shown to be an effective technique for scheduling loop intensive program on VLIW processors [1]. The principle behind software pipelining is to overlap or pipeline different iterations of the loop body in order to exploit parallelism.

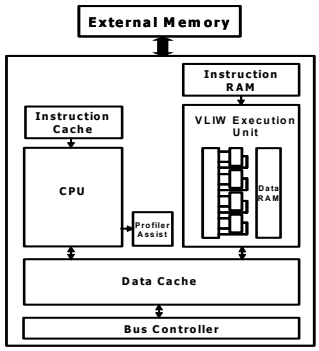

Figure 3. Proposed architecture for SoC implementation of SysteMorph

However loops including conditional branches are difficult to handle for the software pipelining because there are multiple paths of execution to schedule. To address this problem Trace based Dynamic Software Pipelining (TSWP) is proposed. Figure 4 shows outline of TSWP technique. TSWP applies to hot paths which are detected based on online profile information. In this technique, the high frequently executed path of the loop is selected among different available paths. Then, it is optimized, using software pipelining and executed speculatively. In the case of branch misprediction, the compensation code, which is generated by the optimizer, is run.

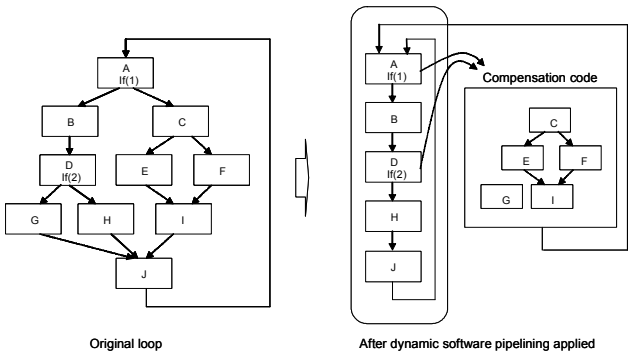

Figure 4. Outline of Dynamic Software Pipelining

When the optimized code is generated, it is passed to the rewrite binary procedure. In this procedure, the new code is rewritten and the first instruction of the hot path is replaced by a call instruction to the start address of the new generated instructions. The optimized code contains all necessary instructions for initializing the accelerator register file. Binary rewriting makes the instructions generated by the optimizer, executable by the accelerator. After binary rewriting, the interrupt handler finishes and control returns to the target application. For the next time when the loop including the hot path is executed, the optimized code will be executed speculatively. If the predicted path is not hot path, the compensation code will be executed.

The accelerator is a simplified 8-way five-stage VLIW processor which excludes many unnecessary instructions such as memory management and instructions for supporting the OS. This simple architecture of the accelerator helps it to have a higher performance and lower power consumption and faster clock frequency. Similarities between the opcodes of main processor and accelerator facilitate the binary translation and optimization. To obtain better performance using TSWP, 128 32-bit registers have been used by the accelerator.

5. PRELIMINARY PERFORMANCE EVALUATION

Figure 5 shows the results of running SysteMorph hot path detector for some of MiBench programs [2]. The line corresponds to the right Y-axis which shows the number of detected hot paths in the applications and the bars correspond to the left Y-axis which illustrate the total instructions of all the hot paths seen in the application. According to the results, in most cases there are tiny parts of the code which have been detected as hot paths (less than 30 hot paths have been detected except two cases). Mostly the hot paths contain less than 50 instructions. This information shows that hot path optimizer will be called not so much and there will not be much overhead. Also it shows that the optimized code is small enough to be fitted in a small memory instruction for accelerator which allow efficient execution without instruction fetch using main memory.

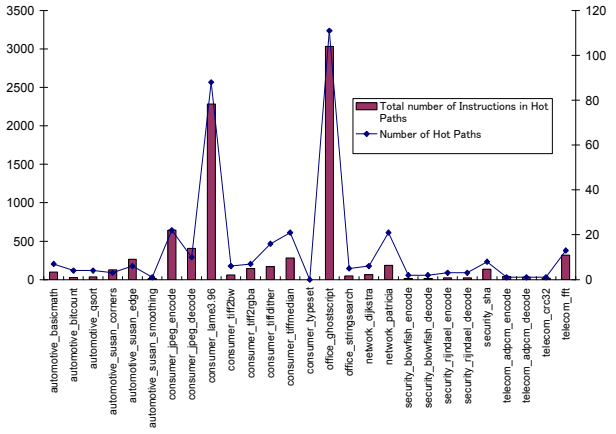

Figure 5. Number of hot paths and their total instructions for MiBench

To have a preliminary performance evaluation of SysteMorph, two experiments have been done. A single issue RISC processor (model A) has been compared to a single issue RISC processor, once augmented with a 4-way VLIW accelerator (model B) and once with an 8-way VLIW accelerator (model C). SimpleScalar [3] tool set has been used as the simulator and Mibench programs as our benchmark. The 4-way accelerator has three integer/branch units and one load/store unit. The 8-way accelerator has 7 integer/branch units and one load/store unit. Each instruction has been supposed to take one clock cycle. Each processor can execute at most one branch instruction in each cycle. To evaluate the speedup, the application was run on model A (sim-safe of Simplescalar tool set was utilized) and clock cycles were counted, so called N_{all}. Then the hot path detector was run for the output of the sim-safe, which is sequence of executed instructions on the processor, and hot paths were detected. Fig. 6 shows the percentage of the total executed instructions covered by hot paths. As it can be seen, in most cases, hot paths are critical regions of the code, which cover a big part of total executed instructions.

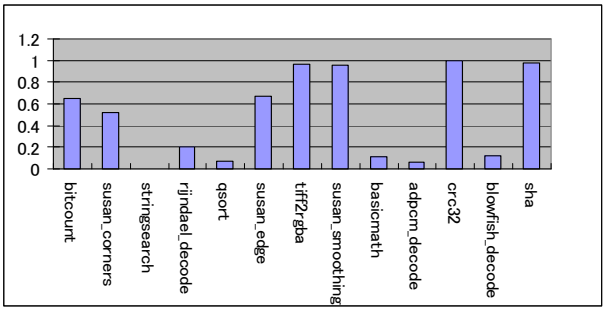

Figure 6. Percentage of executed instructions covered by hot paths

For the first experiment, hot paths were optimized using list scheduling and in the second experiment they were optimized by traced-based dynamic software pipelining. The speedup was calculated using: $speedup = N_{all} / ((N_{all} - N_{HP}) + (N_{OPHP}))$ (Eq. 1). N_{all} is the total clock cycle counts for executing the applications on the single issue RISC processor. N_{HP} is the clock cycle counts for executing hot paths on the single issue processor without applying any optimization. And N_{OPHP} is the number of clock cycles for executing optimized hot paths (hot paths were optimized by list scheduling and TSWP) on the accelerator. Figure 7 and 8 show the speed up gained by applying list scheduling and TSWP to the hot paths for 4-way and 8-way accelerators, using Eq. 1. According to the Fig 7 and Fig 8, TSWP can not always improve the speedup comparing to list scheduling. Due to tight dependences and load/store instructions for some applications in their hot paths, it is hard to overlap consecutive iterations of the loop. But for the applications that have less dependences and have no or few load/store instructions in their hot paths, TSWP boosts the performance more. In this phase of evaluation, the overhead of optimization and switching from processor to accelerator and reverse have been ignored. Also the overhead of hot paths misprediction recovery has not been considered. At runtime, hot paths are logically, sequential code. Therefore, jump and branch instructions in hot paths are treated exactly like other instructions.

6. CONCLUSIONS AND FUTURE WORK

In this paper, a dynamically adaptable architecture based on online profiling information has been proposed. This architecture tries to obtain better performance by accelerating the frequently executed dynamic trace of loops (hot paths) of the applications. The hot paths detection is performed, according to the binary code online profiling information. The profiler is hardware based which decreases the overhead. Hot paths identification is done by studying the behavior of branch/jump instructions. Hardware accelerator is used for executing the hot paths. Binary rewriting provides the feasibility for running the hot paths on the accelerator. Running hot path detector for MiBench has shown that mostly for each application the number of hot paths is less than 30 and their length is approximately 50 instructions. Due to these numbers little overhead for system and small instruction memory for accelerator will be expected.

Among different architectures we have focused on an architecture which contains a hardware/software profiler and an 8-way VLIW accelerator. Dynamic software pipelining is used as the dynamic optimization technique. Besides optimizing the code, the compensation code for hot path recovery is generated. The preliminary performance evaluations show improvement, though it's rough and more exact evaluation is going to be done. We are trying to turn over profiling and optimizing to the hardware as much as possible to reduce the overhead of runtime software while not losing the flexibility of the optimizer.

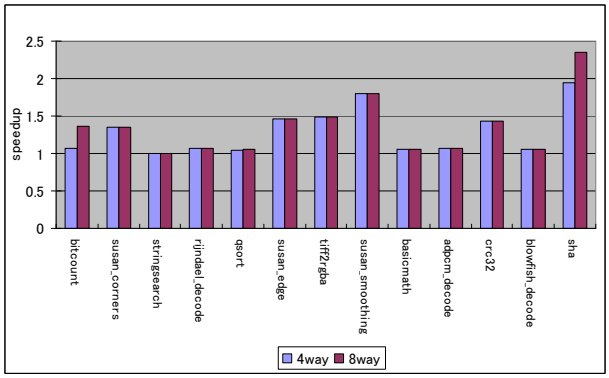

Figure 7. Speedup gained by applying list scheduling to hot paths

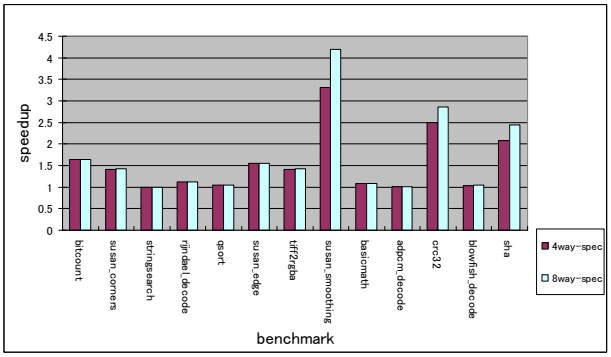

Figure 8. Speedup gained by applying TSWP to hot paths

7. REFERENCES

[1] Lam, M. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *In Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, Jun. 1998.

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

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

[4] Rosner R., Almog Y., Moffie M., Schwartz N. and Mendelson A. Power Awareness through Selective Dynamically Optimized Traces, *ISCA'04*, 2004.

[5] Black B. and Shen J. P. Turboscalar: A High Frequency High IPC Microarchitecture. *ISCA27*, June 2000.

[6] Klaiber A., The Technology Behind Crusoe Processors. *Transmeta Corporation White Paper*, 2000.

[7] http://www.cs.ucr.edu/~vahid/warp/

[8] Patel S. and Lumetta S., rePlay: A Hardware Framework for Dynamic Optimization, *IEEE Trans. on Computers*, 50(6), pp 590-608, June 2001.