The KIT COSMOS Processor: Introducing CONDOR

Toshinori Sato  
Department of Artificial Intelligence  
Kyushu Institute of Technology  
680-4 Kawazu, Iizuka, 820-8502 Japan

Itsujiro Arita  
Department of Artificial Intelligence  
Kyushu Institute of Technology  
680-4 Kawazu, Iizuka, 820-8502 Japan

Abstract In this paper, we propose a microprocessor architecture which efficiently utilizes next-generation semiconductor technology. While the technology makes it possible to integrate a lot of functional units on a single chip, contemporary microprocessors can not exploit much instruction level parallelism so that the units are wasted. Our proposal based on Simultaneous Multi-Threading (SMT) increases the number of beneficial instructions and thus the functional units work efficiently. As well as application programs, a binary code translator which dynamically optimizes the applications on-the-fly is executed on the SMT. We call this mechanism CONcurrent Dynamic Opti

mizer (CONDOR). Based on the CONDOR architecture, performance of the applications is improved as well. We are currently studying the KIT COSMOS processor which utilizes CONDOR. This paper describes some ideas behind the architecture and the goal of this study.  

Keywords: simultaneous multi-threading, instruction level parallelism, dynamic optimization

1 Introduction

For past decades, advancing semiconductor technology has increased the number of transistors on a single chip. The Semiconductor Industry Association predicted that by 2010, 800M transistors will be integrated on a single chip [4]. Contemporary processors cannot utilize such an unprecedented number of transistors efficiently, because increasing the number of functional units cannot contribute to processor performance due to low instruction level parallelism (ILP) in programs such as desktop applications.

In order to extract more ILP, modern processors rely on speculative execution or multi-path execution. In the case of control flow speculation based on branch prediction [22], instructions which are predicted to be possibly executing in the near future have to be squashed when the prediction is incorrect. This only increases useless computing. In the case of data flow speculation based on value prediction [19], misspeculated instructions should be invalidated and reissued in order to revert processor state to a safe point when the speculation initiates, increasing redundant execution of an instruction.

On the other hand, a multi-path processor simultaneously executes code from both taken and not-taken outcomes of a branch [30]. This eliminates misspeculation penalties of branch prediction. When the branch is resolved, one of the two paths is valid and the other has to be discarded. Thus, processor resources are wasted by the discarded instructions. Confidence estimation [13] controls the explosion of the wasted instructions by determining when to diverge both paths, but cannot be perfect.

From the above observations, we conclude that both the speculative execution and the multi-path execution extract ILP at the risk of execution of useless instructions and thus they are justified only under the condition where a number of functional units are idle. Actually, these techniques contribute to processor performance so far. However, deep speculation will fall in diminishing return as instruction issue width increases, since almost worthless instructions are injected execution core.

These considerations lead us to explore a new way to extract beneficial instructions. Killing functional units are efficiently utilized by multi-threading. An application and an optimizing program execute simultaneously on a multi-threaded processor. The optimizing program improves performance of the application on-the-fly. We call this technique CONcurrent Dynamic Optimizer (CONDOR) [27]. Note that we will not emphasize that the total number of ILP is increased by multi-threading. Instead, more ILP is extracted from the single application due to the dynamic optimization. This execution model is similar with Simultaneous Multi-Threading (SMT) [11, 28]. Therefore, we investigate a processor architecture utilizing CONDOR on the top of the SMT processor, which we call COSMOS processor.

The organization of this paper is as follows. Sections 2 and 3 describe the SMT processor and the dynamic optimization, introducing the CONDOR architecture. Sections 4 and 5 explain to-do matters for realizing the CONDOR
architecture and the COSMOS processor. In Section 6, related works are surveyed. Finally, our conclusions are summarized in Section 7.

2 Simultaneous Multi-Threading

In this section, we describe SMT architecture [28], which is the basis of the CONDOR architecture. Recently, COMPAQ announced that EV8 microprocessor will be based on SMT [11].

In an SMT processor, multiple threads are able to share resources. Executing multiple applications simultaneously on the SMT processor makes possible to exchange ILP and thread level parallelism (TLP). For example, when ILP of an application executing on the processor is small, the processor encounters other threads, improving resource utilization.

Figure 1 explains how processors exploit functional units [11]. The vertical and horizontal axes denote time and instruction issue slots respectively. Figures 1(a)–(d) are for a superscalar processor, a chip multi-processor (CMP), an interleaving multi-threaded processor, and an SMT processor, respectively. Each square corresponds to an issue slot and a white one is an empty slot. The superscalar processor cannot utilize its resources if plenty of ILP are not included in a single application. The CMP can not exploit the issue slots when it executes sequential parts in the application, where there are not any TLP. The interleaving multi-threaded processor can utilize vertically empty issue slots, but it cannot do horizontally empty ones. On the other hand, the SMT processor can issue instructions into both vertical and horizontal slots efficiently.

One of the characteristics which make the SMT attractive is that SMT processors are realized with slight modification to modern superscalar processors. What have to be changed are as follows:

• multiple program counters for multiple threads
• large register files with high bandwidth
• instruction retirement mechanism dedicated for individual thread
• branch predictor considering thread identifiers

On the other hand, it is not necessary to modify functional units, caches, and address translation mechanisms.

3 Dynamic Optimization

The dynamic optimization technique is different from static optimization which a compiler performs in that it optimizes a binary when its corresponding application program executes [2]. Since instruction set is maintained during the optimization, any special hardware is not requested to support it. In addition, the dynamic optimization does not translate input source code as JIT compilers and does not require any annotations in the input source code like dynamic compilation systems. The dynamic optimization is only one step of total optimization process.

The dynamic optimization technique is effectively applicable to program sequences which compilers have difficulties to find statically. For example, the dynamic technique makes it possible to optimize a binary including dynamically-linked libraries beyond procedure boundaries. Recent trends of advancing object oriented programs and of software distributions in the form of dynamic linked libraries (DLLs) are favorable winds for the dynamic optimization. Note that the dynamic optimization will not replace the static optimization, but there is a complementary relationship between them.

The dynamic optimization technique can utilize profile information gathered on-the-fly. Based on the information, instruction traces frequently executing are selected for optimization. It is expected that small portions of whole program dominate its execution time, and thus even optimizing only the small portions contributes to total performance. During gathering profiles, the program is interpreted by the dynamic optimizer. Note that the instruction set of interpretation is equivalent to the instruction set of the target machine. The original program is maintained unchangedly, and the processor transfer its control flow to a region in memory space where optimized binary is stored.

Figure 2 explains dynamic optimization flow. There are seven steps.

1. The optimizer interprets the original binary.
2. The optimizer gathers profile information on-the-fly.
3. According to the profile information, a frequently executing trace is optimized. Meanwhile, execution of the original binary stops.
4. The optimizer resumes the execution after the optimization is done.
5. When a branch to an optimized trace is detected, the processor transfers its control flow to the optimized trace.
6. The processor executes the optimized binary.
7. When a branch to an unoptimized portion is detected, the interpretation starts again.

3.1 Concurrent Dynamic Optimizer

This section explains the CONDOR architecture. Shortly, it combines multi-threading and dynamic optimization technique, and an application program and a binary optimizer execute simultaneously on an SMT processor. Instructions of the application and those of the optimizer are concurrently issued to functional units. Functional units which are idle when executing the application are utilized for dynamically optimizing the application binary. It is found that desktop applications have TLP of at most 1.5 [12], and thus the SMT processor has a context for CONDOR. CONDOR extracts more ILP from the optimized binary. When ILP of the original binary is small, there are a number of idling functional units, which is used for the dynamic optimization. Otherwise, it is not necessary to optimize the binary.

Figure 3 shows the difference between ordinary dynamic optimization technique and CONDOR. In this case, a loop is the candidate for optimization. Figure 3(a) explains the ordinary technique, and (b) describe CONDOR. The vertical axis denotes time. Each square corresponds to an instruction. The original application program consists of instructions 1, 2, 3, ..., and the optimizing program (or CONDOR) consists of instructions A, B, C, ... After optimization, the original application becomes instructions 1', 2', 3', ... .

In the case of the ordinary dynamic optimizer denoted as (a), the condition initiating the optimization is that the frequency of the loop is beyond a threshold, for example 50 times. During optimization, the application stops executing. After the optimization has done, the application starts execution again. Since the period for the optimization is an overhead for the application, it is important to decide which portions of the application should be optimized. This means that the threshold must be large.

On the other hand, under the CONDOR architecture, the application continues to execute during its optimization. Only idling functional units are used for the optimization. When the optimization finishes, the processor terminates the original binary and change to the optimized one. Hence, CONDOR is free from the overhead explained above. Since the overhead is negligible, it is possible to make the threshold for initiating the optimization relatively small. In addition, if it is found during optimization that the frequency of the loop is considerably small, the optimization can be discarded.

From the consideration above, it can be observed that the CONDOR architecture reduces the overheads for dynamic optimization technique efficiently.

3.2 Applications of CONDOR

This section explores application fields where CONDOR is effective. CONDOR is a general purpose instruction optimizer, hence it can do traditional optimization methods as modern compilers do. In addition, CONDOR can perform aggressive optimization based on profile information gathered on-the-fly as follows.

- Branch prediction: Some branch instructions are highly biased to one direction [6]. Such branches can be replaced by unconditional jumps or be removed completely. This improves utilization of branch prediction tables, increasing branch prediction accuracy. In addition, if branches are removed, basic block size are enlarged. These contribute to exploiting ILP.
- Prefetching: Based on profile information including cache miss histories, only frequently missed instructions can be selected for candidates of prefetching. In addition, prefetchings are initiated at optimal point to hide miss penalties since CONDOR knows how long memory access latencies are.

- Memory disambiguation: Serialization of a program caused by ambiguous memory dependences is one of the serious bottlenecks in modern superscalar processors [25]. It is difficult for compilers to detect such dependences, especially pointer-intensive ones. CONDOR can easily detect them dynamically, and can eliminate unnecessary serialization of store and load instructions.

- Value prediction: CONDOR also contributes to data speculation based on value prediction. It is important to select candidate instructions in order to improve prediction accuracy. On the other hand, hybrid predictors are utilized for increasing prediction coverage [26], improving efficiency of data speculation. Based on profile information, CONDOR decides which instructions should be predicted and also decides which component predictor of hybrid one is suited for an instruction.

- Replacing trace cache [23]: CONDOR can replace trace cache by a software cache. It is possible to perform trace scheduling on optimized binaries, and hence dynamic instruction traces are captured by CONDOR.

- Instruction packing: In SPECint95 benchmark, half of instructions have operands less than 16-bits [3]. Thus, in order to utilize wide bitwidth functional units efficiently, optimization of packing such instructions to a single SIMD-style instruction is useful. Let us see the following example. There is an instruction in a loop, whose two operands are less than 16-bits almost all the time. If an ALU is 64-bits wide, four instructions can be packed into an instruction. In this case, CONDOR unrolls the loop four times, and then four identical instructions are replaced by a SIMD-style instruction. The bitwidth of operands does not find until the instruction is encountered, thus compilers cannot do this optimization.

- Multi-path execution: CONDOR improves efficiency of multi-path execution. Base on profile information, it can select branch instructions which can not be predicted with high accuracies. These branches are converted to predicated instructions dynamically.

- Dynamic thread partitioning: Krishnan et al. propose software-based thread partitioning [18]. Single-threaded program is divided into multiple threads statically. CONDOR extends this scheme to do thread partitioning dynamically with the help of value prediction explained above. It is true that compilers can utilize static profile information. However, dynamic profile information, which compilers cannot utilize, is more effective than static one due to the following reasons.

  - Gathering profile information is tedious work for programmers. Thus, if it is automatically obtained when a program is executing, programmers can use their time for other useful work.

  - In the case of the dynamic profiling, a training data set and an actual data set are equivalent. Therefore, significantly accurate profile information is obtained.

Dynamic profiling is supported by performance counters. Modern processors have the performance counters which collect information such as the number of instructions and cache misses. Horowitz et al. [14] propose a mechanism by which hardware is able to inform cache misses to software. Such mechanism can reduce overheads caused by profiling. We are currently investigating how to utilize value prediction miss information gathered by the performance counters.

4 Software Considerations

In this section, we discuss several considerations for realizing CONDOR dynamic optimizer. First, we examine how to start CONDOR. Then, we investigate how to execute an application under control of CONDOR. And last, we explain how to optimize the application.

4.1 Starting CONDOR

As explained above, CONDOR utilizes profile information dynamically gathered. Thus, CONDOR has to obtain control of an application program when it starts. That is, executing the application is replaced by interpreting the application by CONDOR. In other words, CONDOR is a proxy handler. There are several ways to achieve this [2].

1. Modify the kernel loader.
2. Use ptrace to attach to the application program.
3. Extend the program's text segment in a separate copy of the program executable file.
4. Use a special version of cr0.

Bala et al. [2] chose the solution 4. However, we think the solution 1 is more desirable than the solution 4, since the solution 1 is truly transparent to the user. While it requires to modify
operating system, we have to build a complete new operating system for the SMT originally and thus it is not a serious hurdle.

4.2 Executing Application

Since any application program is executed under the control of CONDOR, threads executed on the COSMOS processor does not consist of the application and CONDOR but a piece of CONDOR performing interpretation and another piece of CONDOR performing optimization. Note that the interpreter does not have to be implemented as a virtual machine such as DAISY [9]. The virtual machine is easy to understand, however it can not realize the proxy handling. The COSMOS processor can not extract ILP from the application but from the virtual machine.

In order to minimize the overhead, the interpretation based on binary translation [7, 31] is selected. This scheme is proposed for fast simulation and originally the application executable is translated into the native instruction set of a processor where the simulator executes and buffered for the execution on the processor. Note that the instruction sets of the original binary and the translated one are equivalent for the CONDOR architecture. This removes the translation process using a dedicated hardware support. The requisite for dynamic profiling is to inform CONDOR when it should be performed. This can be easily realized as an exception. For example, branch instruction behavior is gathered as follows. When a branch instruction is encountered, an exception occurs as a CONDOR thread which performs dynamic profiling. The thread is executed simultaneously with the application on the SMT processor.

4.3 Performing Optimization

First, it should be decided where the optimized binary is stored. While it is possible to allocate software cache for the optimized binary without any hardware support [2], we save the optimized binary in a memory area which is not visible to the base architecture [9]. The memory area is mapped to a scratch-pad memory, named Turbo cache.

When a CONDOR thread is triggered by an exception, it updates profile information and then checks the threshold for initiating the optimization. As explained above, it is possible to make the threshold relatively small since the overhead for the initiation is negligible.

In summary, the concurrent dynamic optimization is performed as described in Figure 4. When an application starts, a CONDOR thread for setup intercepts the control. After the setup finishes, the application really starts. When a branch instruction encounters, a CONDOR thread is initiated as an exception handler and an optimization process is triggered if its condition is satisfied. The optimized binary is stored in the Turbo cache.

If the optimized binary is already in the Turbo cache when the CONDOR thread is initiated, the control moves to the Turbo cache. Concurrently, the original binary executing is squashed. Note that instructions following the branch instruction which initiates the CONDOR thread are not retired until the CONDOR thread finishes. When the control moves to any region which is not in the Turbo cache, the original binary re-starts. The exchange of contexts between the original and optimized binaries can be easily handled by the synchronization mechanism implemented in the SMT processor [29].

5 Hardware Considerations

As explained above, the COSMOS processor is based on the SMT processor. It is important to consider impact of hardware complexity on processor cycle time, since the SMT processor is a large-scale superscalar processor. In order to solve the problem, it is widely investigated to split the large processor into a number of small processing elements (PEs). There are two candidate implementations for this purpose.

* On-chip multi-processor
* Clustered superscalar processor

The on-chip multi-processor consists of a number of independent processors, each of which is a PE. One example implementation of the SMT processor on the on-chip multi-processor is the self-synchronizing parallel processing system using FIFO queue [1]. However, we are currently investigating to implement the COSMOS processor as a clustered superscalar processor.
In the case of the clustered superscalar processor, every PE shares an instruction supply mechanism with each other and has a distributed execution engine. When an execution result is delivered from an PE to another one, this data forwarding suffers larger latency than delivery inside an PE. Thus, it is desirable to issue a series of dependent instructions into an PE. In the case of the SMT processor, there are several independent threads on-the-fly and hence it is easily split instructions into each PE based on thread identifiers. Currently, we are investigating to implement the COSMOS processor on trace processor architecture [24]. CONDOR and each trace generated by CONDOR are executed on independent PEs.

6 Related Work

The CONDOR architecture proposed in this paper is strongly based on the SMT architecture [28]. Originally, the SMT was proposed to improve throughput of processor resources by executing multiple applications in parallel. Thus, when only one application executes on the SMT processor, it is difficult to exploit TLP. Following Tulsen et al., several extensions to the SMT are proposed.

Wallace et al. [30] implement multiple path execution mechanism on the top of the SMT. Both possible targets of a conditional branch instruction are executed in parallel as individual threads on the SMT processor. When the branch instruction is resolved, the path (thread) which is not selected is squashed. In order to copy processor states when multiple paths are encountered, the SMT processor requires a large network between threads.

Chappel et al. [5] investigate to improve performance of an application thread (a primary thread) by allowing subordinate microthreads to support the primary thread. They call this mechanism Simultaneous Subordinate Microthreading (SSMT). For example, subordinate microthreads can assist the primary thread by prefetching data requested by the primary thread. CONDOR is different from the SSMT in the following reasons. (1) The SSMT requires a microcode RAM which keeps the microthreads. On the other hand, CONDOR is placed on the main memory space. (2) The SSMT does not optimize the application thread. In addition, it requires profile information in order to insert instructions which initiate the microthreads. CONDOR dynamically optimizes the application program using profile information gathered on-the-fly.

Keckler et al. [16] and Zilles et al. [32] propose to use multi-threading for exception handling. An exception handler is forked as an individual threads and executes simultaneously with its faulting threads on the SMT processor. This removes useless action squashing instructions in the faulting thread, improving performance of the application thread. Since exceptions occur rarely, the use of multi-threading is conservative.

There are two ways to optimize binary codes dynamically. One is based on hardware mechanisms, and the other is based on software optimizer. The hardware mechanisms include NCB [25], trace cache [23], and DIF [21]. They construct instruction traces on the fly for improving instruction fetch bandwidth. On the other hand, the CONDOR is an optimizing program.

Dynamo [2], Dynamic Rescheduling [8], DAISY [9] and Code Morphing [17] are based on software technology. Dynamic Rescheduling translates binary codes in order to keep compatibility between generations of VLIW machines. It does not optimize the binaries. DAISY translates binary codes for an RISC machine into a VLIW machine. Similarly, Code Morphing processors translate binaries for the x86 instruction set into those for a VLIW instruction set. CONDOR does not change the instruction set during optimization. DAISY and Code Morphing gathers profile information dynamically for translation [10, 17]. CONDOR also utilizes the dynamic profiling. One of the serious problems in DAISY is increasing cache miss frequency of the translated VLIW instructions [10]. On the other hand, CONDOR is free from the problem since it keeps instruction set during optimization. In addition, CONDOR has a potential to perform optimization improving cache utilization [15, 20]. Dynamo is a software optimizer and keeps instruction set during optimization. It also utilizes profile information gathered dynamically. CONDOR and Dynamo share these characteristics. However, CONDOR differs from Dynamo in when they initiate optimization. In the case of Dynamo, the application thread sleeps while it is optimized. On the other hand, CONDOR allows the application thread to continue during its optimization. CONDOR executes simultaneously with the application thread. Due to the policy, it is important for Dynamo to reduce overheads caused by the optimization. Thus, Dynamo is based on a heuristic that an optimization is initiated after the corresponding trace appears more than 50 times. CONDOR is not constrained by the heuristic. It observes resource utilization and thus can decide efficiently when a optimization should be initiated. Lastly, CONDOR is different from these proposals in that it is supported by the SMT hardware, simplifying management of each thread.

7 Summary

This paper has introduced the CONDOR architecture which can extract more ILP by in-
creasing beneficial instructions. The key behind the architecture is cooperation between hardware and software. An application thread and a subordinate thread, which optimize the application dynamically, are executing simultaneously on an SMT processor. Idling functional units are efficiently utilized and thus overheads caused by the optimization can be eliminated. Currently, we are studying the COSMOS processor based on the CONDOR architecture. One of the primary future studies dealing with CONDOR is performance evaluation. We have a plan developing a CONDOR prototype and a COSMOS simulation model. We expect that detailed evaluations demonstrate usefulness of the CONDOR architecture.

This paper introduced one aspect of our ongoing research in high performance microprocessor COSMOS at Kyushu Institute of Technology. More information is available at http://www.mickey.ai.kyutech.ac.jp/~tsato/cosmos/.

Acknowledgment

The authors thank Prof. Koichiro Tanaka at Kyushu Institute of Technology and Prof. Kazuhide Dogome at Kagoshima National College of Technology for their valuable comments on this study. This work is supported in part by the Grant-in-Aid for Scientific Research #12780273, and by a grant from Marubun Research Promotion Foundation.

References