PROFILING WITH HELPER THREADS

Takamasa Tokunaga
OMRON Corporation
email: toku@mickey.ai.kyutech.ac.jp

Toshinori Sato
PRESTO, JST
email: toshinori.sato@computer.org

ABSTRACT
Dynamic optimization technology improves its target system on the fly according to the system status. There are some studies on microprocessors to dynamically improve performance and energy efficiency. We are investigating a software-based dynamic optimization mechanism on a multi-threaded processor to improve performance. Optimization is performed on binary codes. To optimize binary codes dynamically, profile information is required. Gathering profiles suffers overhead, and thus low overhead scheme is strongly required. This paper proposes to utilize helper threads to gather profile information with little overhead. Simulation results are also presented.

KEY WORDS
simultaneous multi-threading, helper threads, thread level parallelism, instruction level parallelism, dynamic optimization

1 Introduction

In order to extract more instruction level parallelism (ILP), modern microprocessors rely on speculative execution or multi-path execution techniques. One of their drawbacks is that they execute useless instructions. Thus, they are justified only under the condition where a number of functional units are idle. Actually, these techniques have so far contributed to processor performance. However, deep speculation will fall in diminishing return as instruction issue width increases, since most of the instructions injected to the execution core might be worthless.

These considerations lead us to explore a new way to extract beneficial instructions. Idle functional units are efficiently utilized by multi-threading. An application and an optimizer are executed simultaneously on a multi-threaded processor. The optimizer improves performance of the application on-the-fly. We call this technique Condor[5]. Note that we will not claim that the total amount 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 to simultaneous multi-threading (SMT)[7]. Therefore, we are investigating our Condor architecture on the top of the SMT processor.

This paper focuses on how to gather profile information for dynamic optimization via Condor on-the-fly. It is required that the profiling is accurate and free from severe overhead. We will show that the traditional profiling technique suffers serious performance loss and thus will propose to exploit helper threads to reduce the loss. The organization of this paper is as follows: Section 2 briefly explains Condor architecture. Section 3 shows performance overhead due to profiling. Section 4 proposes to utilize helper threads for profiling. Section 5 describes evaluation methodology. Section 6 presents evaluation results. Finally, Section 7 concludes the paper.

2 Overview of Condor

This section briefly explains the Condor architecture. Its details can be found in [4, 5, 8]. It combines the dynamic optimization technique and multi-threading, where an application program and an optimizer are executed simultaneously. On Condor, instructions concerning the application and those concerning the optimizer are simultaneously issued to functional units. Functional units which are idle when executing the application are utilized for dynamically optimizing the application binary. Condor extracts more ILP from the optimized binary. When the ILP of the original binary is small, there are a number of idle functional units, which are used for the dynamic optimization. Otherwise, it is not necessary to optimize the binary.

Figure 1 shows the difference between the previously proposed dynamic optimization technique and Condor. In this case, a loop is the candidate for optimization. Figure 1(a) explains the previously proposed technique such as Dynamo[1], and (b) describes 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 previously proposed dynamic optimizer denoted as (a), the condition initiating the optimization is that a loop frequency is beyond a threshold, for example 50 times. During optimization, the application stops executing. After the optimization is completed, the application again starts

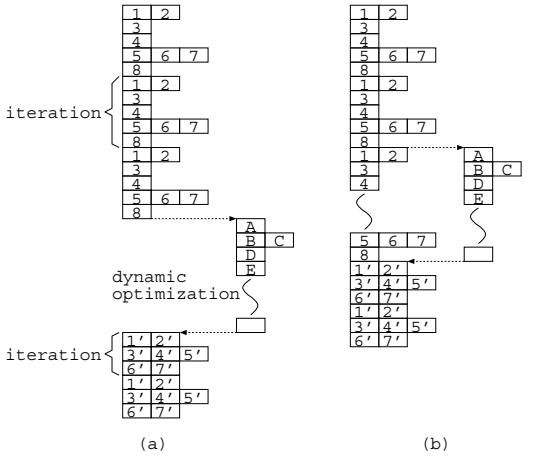

Figure 1. Condor dynamic optimization

execution. 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.

In contrast, under the Condor architecture, the application continues to be executed during its optimization. Only idle functional units are used for the optimization. When the optimization is completed, the processor terminates the original binary and transfers its control flow 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 loop frequency is considerably small, the optimization can be discarded.

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

3 Profiling overhead

Profiling has an overhead on execution time. This section evaluates the overhead using a single-threaded supersclar processor. For profiling, we use ATOM tool[6]

3.1 ATOM

ATOM (Analysis Tools with OM) tool is a framework for building program analysis tools implemented on the Alpha AXP under OSF/1[6]. ATOM tool kit includes a package consisting of several profiling tools such as `prof`, `pixie`, and `third degree`. Using these tools enables us to analyze a program. It is also possible to build any user-customized tools using ATOM. Figure 2 shows how to gather profile information. A user prepares an application program, an instrumenting file (`tool.inst.c`), and an analysis file (`tool.anal.c`). The application program is

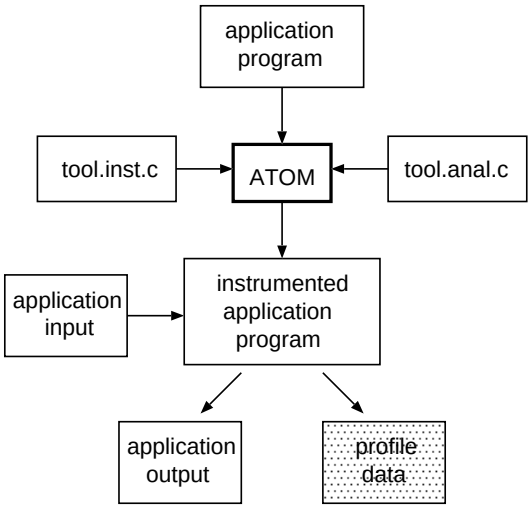

Figure 2. ATOM Tool

Table 1. Processor configuration

L1 I-cache	16KB, 32B blk, direct-mapped
L1 D-cache	16KB, 32B blk, 4-way
L2 cache	unified, 256KB, 64B blk, 4-way
Int reg.file	32 registers
FP reg.file	32 registers
Fetch queue	4 instructions
Inst. window	16 instructions
Ld/St queue	8 instructions
Func. units	iALU x 4, iMULT/DIV x 1
	fALU x 4, fMULT/DIV x 1
	Ld/St x 2
Br.predictor	512-set, 4-way BTB
	2048-entry bimodal
Fetch width	4 instructions
Decode width	4 instructions
Issue width	4 instructions
Commit width	4 instructions

the object of profiling. ATOM injects codes for profiling into the application program. The instrumentation file is implemented using C language. It specifies where the application program is to be instrumented such as `program`, `object`, `procedure`, `block`, and `instruction` via ATOM-API. The analysis file is also implemented using C language. It consists of analysis routines, which are invoked by the instrumentation file. ATOM combines these three files to build a customized instrumentation tool.

3.2 Results

In order to measure the profiling overhead, we execute instrumented application program on a single-threaded processor simulator. The profile includes information on branch instructions. We can find where

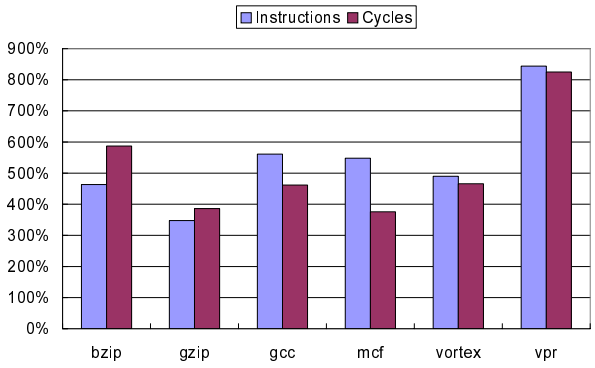

Figure 3. Profiling overhead

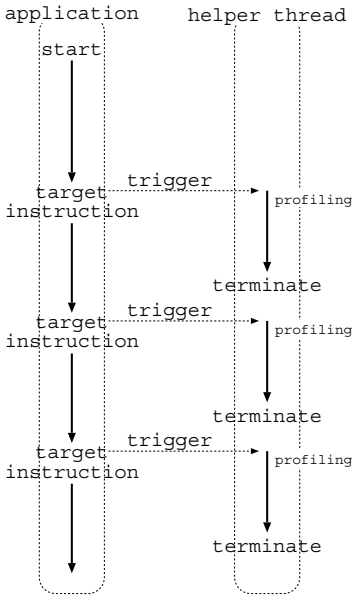

Figure 4. Profiling flow on SMT

every branch jumps and how many times it jumps via profile.

We use a timing simulator, `sim-outorder`, from SimpleScalar/Alpha tool set (version 3.0)[2]. We measure six programs from SPEC2000 benchmark. The processor configuration is summarized in Table 1.

Figure 3 shows simulation results. The left bar indicates the percentage of the increase in the number of dynamic instructions after the instrumentation by ATOM. The right one indicates the percentage of the increase in execution cycles after the instrumentation. It is observed that the number of dynamic instructions is increased by the factor of 4.4 to 9.4 and that execution cycle is also increased by the factor of 4.8 to 9.2. From these results, we found that the traditional profiling method suffers a significantly large execution overhead.

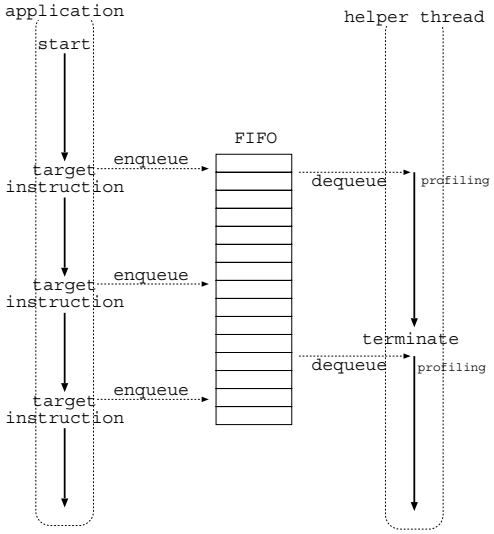

Figure 5. Thread generation using queue

4 Profiling with helper thread

As we have seen in Section 3, profiling suffers a large execution overhead. Even if performance of an application program is improved by dynamic optimization, total performance may not be improved if its profiling has a substantially large overhead. On the other hand, the quality of profile information is also important. Even if the overhead is considerably small, the profile information is useless for dynamic optimization if it is inaccurate. From these considerations, we have come up with to investigate any low-overhead and accurate profiling technique.

In order to attain the goal, we propose to execute both the application and the profiling codes simultaneously on an SMT processor. The profiling code is executed as a helper thread that is independent of the application thread. The helper threads are executed on functional units that are idle when only executing the application. These characteristics enable to gather accurate profile information with a little impact on application performance. Figure 4 shows the profiling flow. During the main thread (application thread) is running, detecting a target instruction triggers the helper thread (profiling thread). Each helper thread corrects dynamic information on the target instruction.

We are considering two models of the trigger process; `Queue` and `Threaded` models. Next, we explain the details of the models.

4.1 Queue model

The `Queue` models use a FIFO queue to strictly satisfy the order of helper threads. Under this model, only one helper thread is executed at a time, and hence at

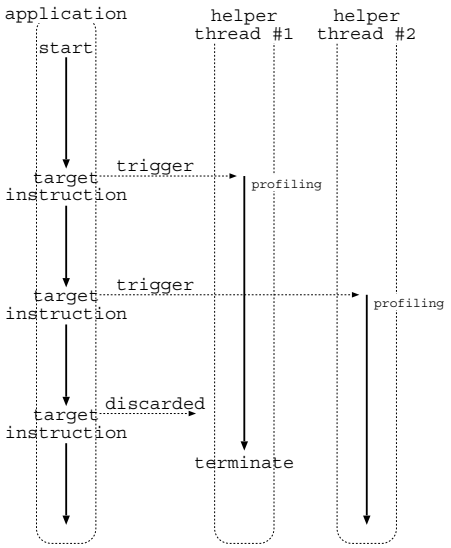

Figure 6. Threaded generation without queue

most two thread are executed on an SMT processor. Figure 5 shows the `Queue` model. When every helper thread is triggered, it is registered in the FIFO queue. When it is on the top of the FIFO and its predecessor thread is finished, it starts execution. If the FIFO is full, the triggered helper thread is discarded.

4.2 Threaded model

Under the `Threaded` model, each helper thread is executed as an independent hardware context. Because the number of hardware context that can be executed simultaneously is limited, the last triggered thread is discarded when the context is fully used. Figure 6 shows the `Threaded` model in the case where the number of helper threads is limited to two.

4.3 Helper thread example

This section presents a example of helper threads. In this example, dynamic information on branch instructions is profiled. Target instructions are conditional branches listed below.

- conditional branches
  - `beq bne blt bgt bge blbc ble blbs`
- floating-point conditional branches
  - `fbeq fbne fblt fbgt fbge fble`

Among these branch instructions, we explain `beq` and `fbgt` here. For other instructions, please refer Alpha instruction set architecture manual. `bqe` means branch equal zero, and has the following format;

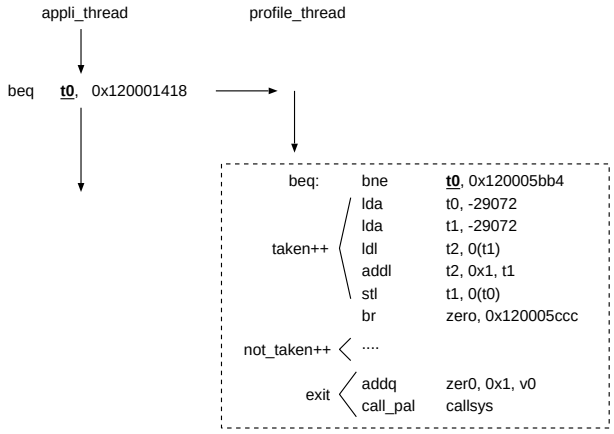

Figure 7. Profile thread for beq

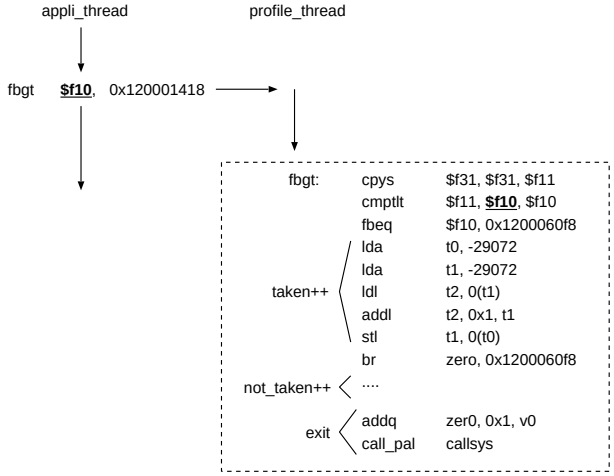

Figure 8. Profile thread for fbgt

```
beq    i-register, address
```

If the value in the integer register, `i-register`, equals zero, instruction flow jumps to `address`. `fbgt` means branch grater than zero, and has the following format;

```
fbgt   fp-register, address
```

If the value in the floating-point register, `fp-register`, is greater than 0.0, the instruction flow jumps to `address`.

Next, we explain helper threads for these two branches. Figures 7 and 8 are the helper threads for `beq` and `fgbt`, respectively. First, we explain the helper thread for `beq`. When a `beq` instruction is issued in the main (application) thread, this helper thread is triggered. It is assumed that the value of the argument is provided with the helper thread. Using the value, `bne` instruction at the first line is executed. Hence, when the target instruction `beq` is taken, the `bne` is not taken. From the second to the sixth instructions

count the number of taken branches. Otherwise, another lines, which are not shown in the figure, count the number of not-taken branches. The last two lines are the epilogue of the helper thread.

Next, we explain the helper thread for `fbgt`. Using the value of the argument, first three instructions are executed. If the target instruction `fbgt` is taken, the `fbqeq` in the third line is not taken. From the forth to the eighth instructions count the number of taken branches. Otherwise, another lines, which are not shown in the figure, count the number of not_taken branches. The last two lines are the epilogue of the helper thread.

5 Methodology

We implemented our simulator using an SMT processor, `ss_smt`[3], which is a derivative of SimpleScalar/PISA tool set[2]. We modified it to execute Alpha binaries. Figure 9 shows its block diagram and Table 2 summarizes the processor configuration.

The original `ss_smt` only executes multiple threads from independent programs, and thus we modified it to execute multiple threads from a single program. This requires a hardware mechanism to trigger a new thread in the SMT processor. We implemented it in our modified `ss_smt`. It is assumed that triggering a new thread requires no cycle overhead and that all register values required in the new thread is delivered. All concurrent threads share functional units, instruction window, load and store queue, and instruction and data caches. However, each thread is given a separate program counter, instruction fetch queue, register file, branch predictor, and instruction and data TLBs. Instructions from a single thread are fetched every cycle. We use the ICOUNT[7] to prioritize instructions fetch from different threads.

The following assumptions are made in this evaluation. While each thread has its own branch predictor, every helper thread does not utilize it but predicts every branch as not-taken. Misspeculated branches in the main (application) thread are not the target of profile. A simulation is terminated when the main (application) thread is finished, and thus some helper threads might be killed.

6 Results

This section presents simulation results of performing the example described in Section 4.3. Table 3 shows the baseline performance when the processor executes only one application program. Instructions per cycle (IPC) is used as the metric to evaluate performance. It is observed that the IPC values are considerably smaller than the number of functional units for almost every program. This means there are idle functional

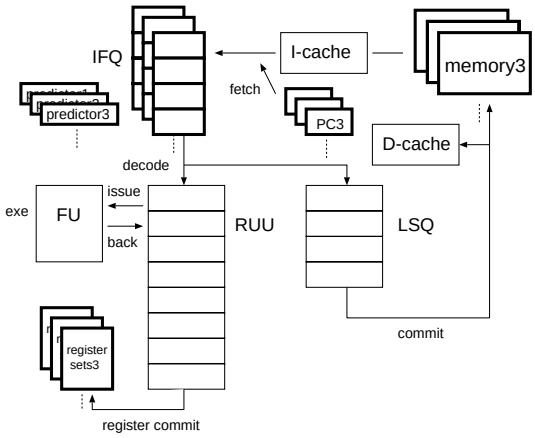

Figure 9. ss_smt simulator

Table 2. Processor configuration

L1 I-cache	16KB, 32B blk, 2-way
L1 D-cache	16KB, 32B blk, 4-way
L2 cache	unified, 256KB, 64B blk, 4-way
Int reg.file	32 registers $*$ # of threads
FP reg.file	32 registers $*$ # of threads
Fetch queue	4 instructions $*$ # of threads
Inst. window	16 instructions
Ld/St queue	8 instructions
Func. units	iALU x 4, iMULT/DIV x 1
	fALU x 4, fMULT/DIV x 1
	Ld/St x 2
Br.predictor	512-set, 4-way BTB
	2048-entry bimodal
Fetch width	4 instructions
Decode width	4 instructions
Issue width	4 instructions
Commit width	4 instructions

units enough to execute helper threads. We measure the increase in execution cycles to evaluate the execution overhead due to profiling. We also measure coverage in available profile. It is defined as the ratio of the number of actually executed helper threads over the total number of triggered ones. It is used as the metric to evaluate accuracy of profiling.

We evaluate both models introduced in Section 4. As explained, the `Queue` model executes just one helper thread as well as a main thread at a time. For evaluating the `Threaded` model, we examine three models which can execute up to 2, 4, and 8 threads, respectively. We use `Threaded-2`, `Threaded-4`, and `Threaded-8` to denote them.

Figure 10 presents the increase in execution cycles over that of the single-threaded model without profiling. For each group of four bars, the bars from left to right indicate results for the `Queue`, `Threaded-2`,

Table 3. Baseline results

program	bzip	gzip	gcc	mcf	vortex	vpr
IPC	2.90	1.96	1.33	1.07	1.75	1.66

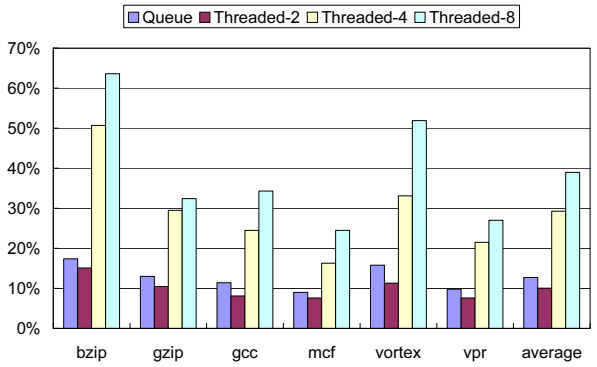

Figure 10. Profiling overhead

`Threaded-4`, and `Threaded-8` models, respectively. By comparing with the values shown in Figure 3, it is observed that the overhead due to profiling is significantly reduced by introducing helper threads. In the case of `Queue` model, it is less than 20% for all programs.

In contrast, in the case of the `Threaded` models, the overhead differs according to the total number of available threads. This is because as follows. As mentioned in Sections 4 and 5, some helper threads are discarded when hardware contexts are fully used by other threads or when the main thread is finished. Figure 11 shows the percentage of helper thread that are actually executed. In other words, it is the coverage of available profile. It can be observed that the coverage increases as the total number of available threads increases. Considering both Figures 10 and 11, it is found there is a tradeoff between the profiling overhead and accuracy. If 100% accuracy is required, the overhead is up to 63.6% while the average is 38.9%. However, this is still significantly smaller than that can be found in Figure 3.

In the case of `Queue` model, the coverage is as small as 44.6% on average, while the overhead is relatively small.

In summary, we found the followings;

- Introducing helper threads is an effective way to reduce execution overhead due to profiling.
- There is a tradeoff between profiling overhead and accuracy.

7 Conclusions

This paper proposed to use helper threads for dynamic profiling. Profile information is required for dynamic optimization technology, which we are investigating to boost processor performance. Traditional profiling on single-threaded processors suffers very large overhead. Our simulation showed it is up to 825%. Our proposed technique utilizes idle functional units for profiling, and thus the overhead is reduced. It is found that it can be reduced to 38.9% on average with almost 100% coverage. From the results, we found that profiling with helper threads is a very effective way to gather accurate profile information, which is utilized for dynamic optimization.

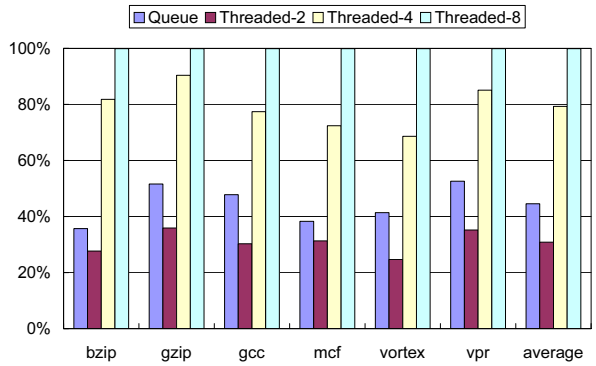

Figure 11. Coverage in available profile

References

[1] V.Bala, et al., Transparent dynamic optimization, HPL-1999-77, HP Lab., 1999.

[2] D. Burger, T. M. Austin: The SimpleScalar tool set,version 2.0, TR-1342, Computer Sciences Dept., Univ. of Wisconsin-Madison, June, 1997.

[3] R. Goncalves, et al., A simulator for SMT architectures: evaluating instruction cache topologies, SBAC-PAD, 2000.

[4] K. Morita, et al., The KIT COSMOS processor: a preliminary study on transparent software prefetching via dynamic optimization, 6th Workshop on Multi-threaded Execution, Architecture and Compilation, 2002.

[5] T. Sato, I. Arita: The KIT COSMOS processor: introducing CONDOR, PDPTA, 2000.

[6] A. Srivastava, A. Eustace: ATOM: a system for building customized program analysis tools, PLDI, 1994.

[7] D. Tullsen, et al., Exploiting choice: instruction fetch and issue on an implementable simltaneous multithreading processor, ISCA, 1996.

[8] T. Yamamoto, et al., The KIT COSMOS processor: an application of multi-threading for dynamic optimization, PDPTA, 2002.