

A Binary Translation System for Multithreaded Processors and its Preliminary Evaluation

Kanemitsu Ootsu, Takashi Yokota, Takafumi Ono, and Takanobu Baba

Department of Information Science, Faculty of Engineering, Utsunomiya University
7-1-2 Yoto, Utsunomiya-shi, Tochigi, 321-8585 Japan.

phone&fax: +81-28-689-{6284, 6290} e-mail: {kim, yokota}@is.utsunomiya-u.ac.jp

Abstract

Thread level parallelism (TLP) is a key technology to coming generation of high performance processors. Although it provides higher processing capability, the loss of compatibility with existing processors is a crucial issue. This research is motivated by the following two points: (1) TLP requires multithread programming which is rather difficult for ordinary programmers, or complexed compilation technologies that can exploit multithread parallelism, and (2) existing binary codes should be executed efficiently on multithreaded processors. In this paper, we first propose a binary translation system, that translates existing binary codes to multithreaded ones and optimizes them dynamically during execution. The system inputs the original binary codes and translates them to internal RTL representation. It analyzes the structure of the program and applies multithreading to loop bodies in a thread pipelining manner. A pilot binary translator, that is a part of the proposed system, was built for the sake of preliminary evaluation. Evaluation results illustrate effectiveness of the system.

Keywords: *binary translation, thread level parallelism, multithreading, thread pipelining, run-time optimization.*

1 Introduction

Thread level parallelism (TLP) is one of the most promising key issue to high-performance processor architecture in the next generation. Present state-of-the-art technologies, such as superscalar, out-of-order, speculative execution, and value prediction, are successful in keeping continuous compatibility with conventional processor's instruction set architecture (ISA). And even in different architectures, i.e., VLIW (very long instruction word), a sort of binary translation technology is adapted so that the processor looks like conventional ISA from users view.

On the other hand, TLP essentially requires multithreaded machine codes to exploit full ability of the architecture. Because of the discontinuity of binary code compatibility, we can find the following two problems.

First, who (what) can produce multithreaded codes? Most programmers are not so skilled to make their application fully multithreaded. To this problem, further compiler technologies are required for automatic multithreading of an original application program in the near future. Second one is rather practical, i.e., we should abandon plenty of existing (single-thread) binary codes if their source codes are not available. The single-thread binary codes could run on multithreaded processors, although, they can receive no performance gain from TLP. Thus these two problems prevent TLP from being widely accepted. As a realistic solution to the problems, we focus our approach on the efficient reuse of existing binary codes on a multithreaded architecture that exploits rich TLP.

In this paper, we propose a binary translation and run-time optimization system.[14, 15] We first introduce binary translation technology that translates existing single-thread code to multithreaded ones. Source binary codes are analyzed, decomposed into threads, and then mapped onto the target architecture.

We then introduce dynamic (run-time) optimization of the translated codes. Because of lack of source code information, static analysis has some limitations: e.g., distinction of instruction words and data is not clear, and the target addresses of indirect jumps remain unknown.

The rest of this paper is organized as follows. We first discuss design principles to realize our ideas in Section 2, where we make some basic assumptions and discuss system requirements. Then we propose a binary translation and optimization system in Section 3, where basic components and their functions are discussed. Section 4 describes static optimizer in detail and Section 5 shows the preliminary evaluation. Section 6 presents related works which aim at binary translation or optimization. This section clarifies the standpoint of the proposed system and thus its unique features. Finally, we conclude this paper in Section 7.

2 Design Principles

2.1 Multithreading by the Thread Pipelining Model

In order to run a single-thread binary code efficiently on a multithreaded processor, logical structure embedded within the source binary code is extracted and the program is restructured to a set of threads. For the following discussion, we make an assumption on the target multithreaded architecture.

Needless to say, single-thread code follows a sequential programming manner. Although the ideal objective is to exploit all possible parallelism inherent in the program, it is not realistic for binary code inputs. We have started discussion with a simple idea: we payed our attention to *loop* structures.

The idea is very natural. A programmer tends to follow a sequential program (thread) depicted in a one-dimensional space. In such situations, a parallel structure is expressed as a loop. In other words, a loop structure contains inherent parallelism. Thus, it is appropriate that each iteration in the loop is converted to a thread. In many cases, a loop structure contains many iterations, and thus enables us to exploit the maximum parallelism.

Thread pipelining model[1] best fits to our purpose described above. Figure 1 shows partial structure of the multithreaded processor based on the model. Each thread generated by binary translation is mapped to a thread execution unit in order. Communication unit and Memory Buffer handle inter-thread control and dependencies, respectively.

Basically, each iteration corresponds to a thread, and threads are executed in a pipeline manner. Figure 2 illustrates the thread pipelining[1]. Each thread consists of four stages: Continuation, Target Store Address Generation (TSAG), Computation, and Writeback.

The Continuation stage introduces loop variables and necessary data so as to be used in the thread. After the Continuation stage completes, the succeeding thread is invoked. The TSAG stage checks dependencies of shared data between threads. Addresses of shared data are notified to Memory Buffer, which detects access dependencies between threads by monitoring addresses. The Computation stage does the peculiar calculation assigned to the thread. After completing the Computation stage, a thread terminates its life in the Writeback stage. The Writeback stage cannot be started until the preceding threads' Writeback stages are completed.

2.2 Single- to Multi-thread Binary Translation

As described above, we have introduced a thread pipelining concept to our system. This pipelining is a fundamental requirement in the proposed system. It is totally different from existing translation systems in that it converts single-thread code to multithreaded one whereas others translate to single-thread. In other words, ordinal translators don't change program structure, although, our system should arrange the analyzed program structure to achieve the best fit to the thread pipelining model.

Source binary code is first analyzed with its logical structure, and the structure is re-organized following the thread pipeline manner. Then, target machine code is generated. During analysis of program structure (i.e., control- and data-flow) and re-configuration phase, abstract representation is required. We have introduced RTL (register transfer level) representation, which is used internally in the translator. Actual translation procedure is displayed in Section 3.

2.3 Necessity of Run-time Optimization

In principle, binary translator converts a loop structure to a set of threads. This requires accurate analysis of program structure, however, the translator could not find full information because of lack of source code information. For example, indirect branch hides the target address of iteration. So, some loop structures may

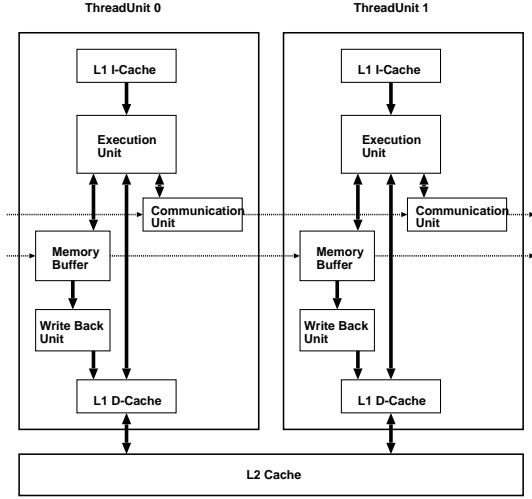


Figure 1: Thread Execution Units

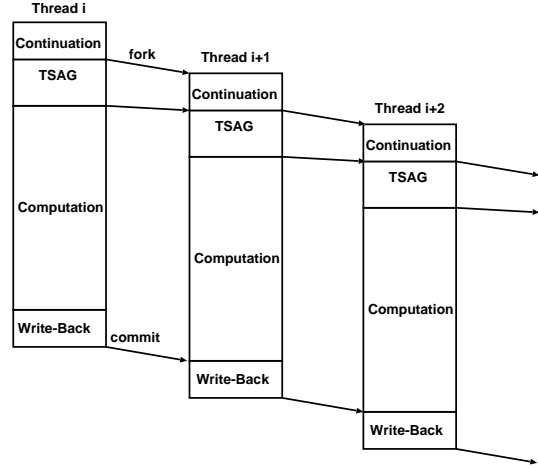


Figure 2: Thread Pipelining Model

remain unfound even when the analysis phase is completed. These loop structures cannot be converted to multithreaded code, and they do not appear until the translated codes run.

Thus, to exploit full parallelism of the program, it is necessary that behavior of the program is monitored and that ‘hot’ portion is translated to multithreaded code. This methodology is a kind of run-time optimization. Like the binary translation introduced in the previous section, run-time optimization requires re-structuring of the program (i.e., converting single-thread code to multithreaded one) where other run-time optimization techniques do not essentially affect program structure.

3 The Binary Translation and Optimization System

3.1 Systems Logical Structure

As discussed above, in order to execute existing binary codes on next-generation multi-threaded processor, the system requires following two phases: (i) binary translation and static optimization and (ii) run-time optimization. Figure 3 illustrates the configuration of the proposed system.

In Figure 3, STO (Static Translation and Optimizer) executes the phase (i) and DTO (Dynamic Translation and Optimizer) performs (ii). The figure includes Multithreaded Processor, whose basic architecture is described in Section 2.1 and Figure 1. The processor’s basic ISA is not limited to some specific architecture since the original binary codes may be translated according to the target architecture by STO and DTO.

STO inputs the sequence of the source binary codes and translates them to the target binary code. If the program requires dynamic linked libraries (DLLs), STO prepares the necessary libraries and links. Resulting executable binary image is put into the main memory and the processor executes the executable.

During the processor executes the translated binary code, the behavior of program is monitored. We introduce profiling techniques for monitoring. We assume that the processor has additional mechanisms that reduce profiling overheads.

DTO uses the profiling information and observes actual behavior of the application program. When detecting a buried ‘hot’ loop, it begins binary translation and optimization. It substitutes the single-thread ‘hot’ part by the translated multithreaded code, and thus accelerates total execution.

3.2 Static Translation and Optimization (STO)

As shown in Figure 3, STO inputs the source binary code and translates it to multithreaded code. Once STO reads the source binary code, the code is translated into an internal representation. The representation is, in principle, abstracted in a machine independent RTL (register transfer level) form. The internal representation

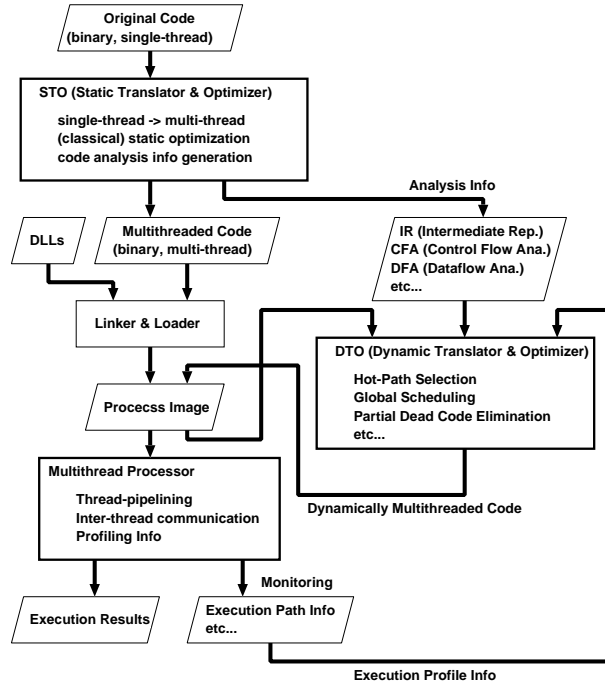


Figure 3: Proposed System Diagram

enables STO to do powerful optimization as taken in ordinal compilers/optimizers. In the proposed system, STO generates threads in the internal representation level.

Basically, STO works before the program runs. Its mission is to prepare translated binary code for the multithreaded processor before the program is started, however, it could not complete the translation. The reason is that no clear distinction is made between instruction code and data and that run-time information is buried. For example, an indirect jump operation hides its branch target address and thus prevents further analysis. Another example is the self-modifying code that determines its own execution code at run-time, so STO cannot know exactly what is to be done in the program. The remaining translation should be done at run-time and DTO handles it.

During analysis of input binary code, STO acquires useful information: code analysis information, control- and data-flow information. DTO does full use of these information. This reduces overheads in run-time optimization. STO inserts profiling codes so that DTO can collect proper information at low cost.

3.3 Dynamic Translation and Optimization (DTO)

DTO's major objective is run-time optimization (Figure 3). Unlike STO, DTO runs concurrently with the execution of application. It is invoked at proper intervals during application execution. DTO collects profiling information and monitors the program behavior. After detecting a hot-path, DTO arranges global scheduling, eliminates redundant codes, and applies possible optimization methods[2]. Then, DTO substitutes the original code to the optimized one.

Profiling codes are not removed by the DTO optimization. This means that profiling continues until the application is terminated. So DTO can apply further optimization incrementally and it can follow the change of program behavior. The DTO approach is similar to profile-guided compilation[3]. Since DTO can collect more detailed information, it should achieve deeper optimizations.

Source binary codes may contain self-modifying codes. STO cannot handle such codes since actual codes are determined at run-time. Thus, DTO should provide similar functions that STO does: i.e., the series of binary translation and optimization processes. Actually, input of source binary code, translation to internal representation, and control- and data-flow analysis should be processed by DTO.

4 Binary Translation Method

We have built an experimental binary translation software in order to estimate the effectiveness of the proposed system. The pilot translation system is to be a part of STO in the proposed system. This section introduces binary translation methods employed in the pilot system.

4.1 Basic Algorithm

As introduced in the previous section, the binary translator inputs binary codes and outputs multithreaded one. The translator performs the following steps:

- (1) inputs source binary code and translates to internal representation,
- (2) determines basic blocks, analyzes control-flow, and detects loop structure,
- (3) analyzes data-flow, detects loop variables and inter-loop dependencies,
- (4) converts loop structure to multithread codes in a thread pipelining fashion, and
- (5) generates target machine code from internal representation.

The translator reads the source binary code. It begins code analysis from the starting address specified in the binary code. Input code is translated into the internal representation in order.

The internal representation categorizes instructions into six groups: alu operation, inter-register transfer, jump, branch, load/store, and other operations. Each instruction category has its unique operand expression. Figure 4 shows a part of instruction stream converted into the internal representation. The internal representation forms a list structure. Once the input binary codes are read and translated into the representation, succeeding processes, (2) to (5), are performed on the representation.

In the step (2), the translator determines basic blocks and analyzes control flow. It seeks back-edges, i.e., backward jumps/branches, in the internal representation. A back-edge is an important hint to mine a loop structure. The translator tries to find a path from the target address of a back-edge to the back-edge itself. If the path exists, it constitutes a loop structure.

In the step (3), the translator analyzes data-flow in the loop structure. It presumes loop variables used in the loop. The present pilot system finds the loop variables by increment of integer variables. The translator can analyze multiplexed loops.

The translator modifies the internal representation according to the result of multithreading operation (in the step (4)). Step (5) generates the target machine codes from the internal representation.

Next, we will explain step (4), the key part of the translator, in more detail.

In order to exploit sufficient parallelism by multithreading, we have found the following two requirements: (i) an interval of thread invocation should be shortened, and (ii) the synchronization time in resolving dependency should be reduced.

To solve (i), the Continuation stage prepares loop variables used in the succeeding thread. The values of loop variables are computed in the preceding thread. A newly created thread can start the execution of its loop body.

To reduce synchronization overheads due to inter-thread dependency (ii), the translator tries to move ‘load’ and ‘store’ instructions of shared data backward and forward in the Computation stage, respectively.

After the Continuation stage, addresses of inter-thread dependent data are registered in Memory Buffer in the TSAG stage. Execution of the consecutive TSAG stages cannot be overlapped since the stage determines shared data. Thus, at the entrance of the stage, the processor waits for completion signal from its preceding thread, and at the exit of the TSAG stage it sends completion signal to its succeeding thread. These inter-thread communications are handled by Communication Unit, shown in Figure 1.

Most calculations in the original loop body are executed in the Computation stage. Inter-thread dependencies are registered at the TSAG stage and Memory Buffer monitors all memory accesses. It handles inter-thread synchronization in a producer-consumer manner.

The Writeback stage writes calculation results onto memory. Since the resulting data should be stored in the semantic order, the stage cannot be overlapped between threads. Thus at the start-point of the stage, the processor should wait for a termination signal from its predecessor. After completion of this stage, the thread terminates.

Following the thread pipelining model, as shown in Figure 2, each thread consists of four stages. Three out of four stages cause overheads and only Computation stage performs the peculiar calculation in the iteration. Thus, to exploit sufficient performance on this model, the Computation stage should be long enough to hide overheads caused in other stages. We have introduced a loop unrolling technique as a solution. The succeeding section discusses the effectiveness.

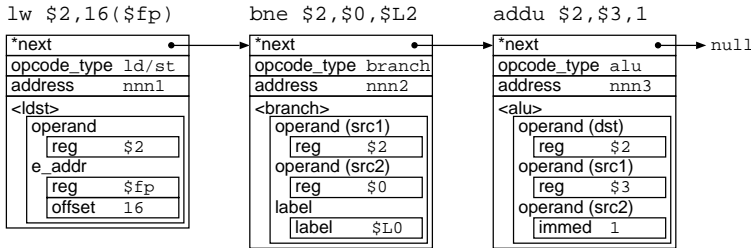


Figure 4: Internal Representation Example

4.2 Translation Example

Figure 5 illustrates a simple example of translation. The instruction stream, listed in the left side, is a part of the source binary code (disassembled for display use). In our pilot translator, only a hot-path (i.e., loop) is translated to multithreaded code as described in Section 4. The right side list shows the translated output of the hot-path.

To avoid complexity in evaluation processes, the original ISA is extended by adding several thread control instructions so that the codes can be run on the target multithreaded architecture.

A thread pipeline begins with **bstr** instruction. In the Continuation stage, the thread calculates the loop variable used in its succeeding thread from its own variable (address ($\$fp+16$) in Figure 5), stores the result by **sttsw** instruction, and then generates the succeeding thread by **lfrk**. Note that a loop variable is accessed via ($\$sp-8$) in this example.

In the TSAG stage, dependent address of ($\$fp+48$) is registered to Memory Buffer by **altsd** instruction. For proper synchronization among neighboring threads, **wtsagd** and **tsagd** instructions are used.

We can find a load instruction that fetches data from ($\$fp+48$). When the instruction is executed, Memory Buffer detects the memory access and execution stalls until the preceding thread updates the data.

In the Writeback stage, **estr** instruction writes calculated data into memory and the thread terminates.

5 Preliminary Evaluation

5.1 Experimental Translation System

To evaluate the basic idea of binary translation to multithreaded codes and their optimization, we have built a pilot translation system. Figure 6 shows the block diagram. The objective of the system is to evaluate the STO functions which was described in Section 3.2. MultiThread Code Generator in Figure 6 follows all the translation steps (1) to (5) described in Section 4.1.

In Figure 6, MultiThread Code Generator translates source binary code to multithreaded one. Hot-path is manually guided to the translator for evaluation purpose. Resulting partial binary code is merged into the original (single-thread) binary code by **Binary Patcher**. Binary Patcher removes hot-spot codes from the original binary code, and inserts translated multithreaded code. Thus the hot-path is executed in the thread-pipelining manner and the rest of the program is executed in a single thread. At this timing, necessary run-time libraries are linked. Then, the prepared multithreaded binary code is executed in a simulator.

source binary code		translated code	
\$L0:	lw \$2,16(\$fp)	/* Continuation Stage */	bstr
	lw \$3,20(\$fp)		lw \$2,16(\$fp)
	slt \$2,\$2,\$3		sw \$2,-8(\$sp)
	bne \$2,\$0,\$L2		lw \$2,-8(\$sp)
	j \$L1		addi \$2,\$2,1
\$L2:	l.s \$f0,16(\$fp)		addi \$3,\$fp,16
	cvt.d.w \$f0,\$f0		sttsw \$3,\$2
	mov.d \$f12,\$f0		lw \$2,-8(\$sp)
	jal sin		lw \$3,20(\$fp)
	l.d \$f2,48(\$fp)		slt \$2,\$2,\$3
	add.d \$f0,\$f2,\$f0		bne \$2,\$0,\$L3
	s.d \$f0,48(\$fp)		j \$ST_END
	lw \$3,16(\$fp)	\$L3:	lfrk
	addu \$2,\$3,1	/* TSAG Stage */	wtsagd
	move \$3,\$2		addiu \$2,\$fp,48
	sw \$3,16(\$fp)		alted \$2
	j \$L0		tsagd
\$L1:		/* Computation Stage */	l.s \$f0,-8(\$sp)
			cvt.d.w \$f0,\$f0
			mov.d \$f12,\$f0
			jal sin
			l.d \$f2,48(\$fp)
			add.d \$f0,\$f2,\$f0
			s.d \$f0,48(\$fp)
			addiu \$3,\$fp,48
			addiu \$4,\$fp,48
			lw \$2,0(\$3)
			sttsw \$4,\$2
		/* Writeback Stage */	\$ST_END

Figure 5: A Simple Example of Binary Translation

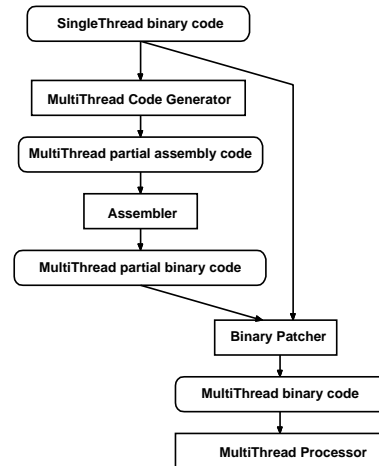


Figure 6: Block Diagram of the Pilot System

5.2 Evaluation Environment

We have assumed that the target multithreaded processor follows the architecture of SIMCA[4]. SIMCA is a simulator based on thread pipelining model and matches to our evaluation purpose.

Original binary codes are compiled by *gcc* cross compiler for SIMCA. The compiler's version is 2.7.2.3 and "-O2" option is applied. Application programs are (a) integral calculation in a 'sin' trigonometric function using a trapezoidal equation and (b) inner product calculation.

Performance was measured as execution cycles of the hot-path by using the SIMCA simulator. Original binary code was executed on SIMCA and the number of execution cycles of the hot-path was measured. Similar evaluation was done for the translated code. By comparing the number of execution cycles, speed-up ratio was calculated.

In this preliminary evaluation, the number of thread units were assumed to be 4, 8, and 16. Furthermore, the loop-unrolling technique was applied to each application program; the measured unrolling factors were 4, 8, and 16 and no unrolling was measured for comparison purpose.

5.3 Evaluation Results

Figures 7 and 8 illustrate evaluation results for integral and inner product calculation applications, respectively.

In the integral calculation (Figure 7), the system gains performance linearly to the number of thread units. Unrolling factor does not affect the performance except 'no unroll' case.

In the inner product calculation (Figure 8), we can find that speed-up ratio is limited by unrolling factor. In 'no unroll' case, speed-up ratio is around 0.9 in spite of the number of thread units. We can find the similar phenomenon in the 'unroll 4' case. In the 'unroll 8' case, speed-up is achieved when 8 thread units are used. However, the performance saturates in the 16 thread units case. We can recognize linear speed-up in the 'unroll 16' case.

The integral calculation contains many operations enough to hide thread pipelining overheads. This leads near-linear speed-up according to the number of thread units.

On the other hand, the inner product calculation contains less operations than the integral calculation. Thus thread pipelining overheads could not be hidden unless sufficient loop-unrolling is applied.

These results reveal that efficiency in thread pipelining heavily depends on the grain size of calculation.

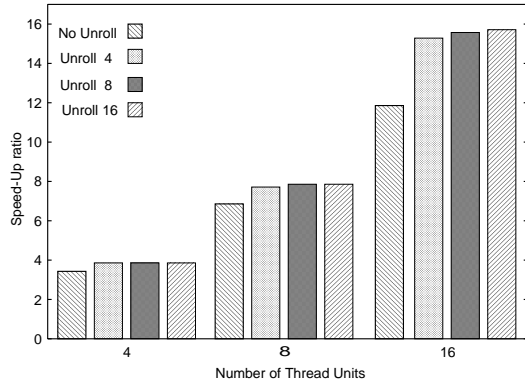


Figure 7: Speed-up Ratio in Integral Calculation

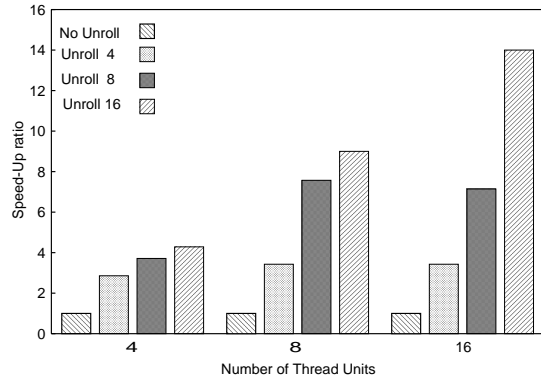


Figure 8: Speed-up Ratio in Inner Product Calculation

6 Related Works

In general, the major objective of binary translation is to execute existing binary codes based on different ISA.

FX!32[5] of Compaq is a translation subsystem in Windows NT for Alpha processor, that enables x86 win32 codes to run on Alpha platforms. It emulates x86 instructions and does binary translation into Alpha ISA codes. During idle time, The binary translation is executed with profiling results from the preceding emulation. Once the code is translated, the resulting native code is executed and earns high performance.

DAISY[6] of IBM translates well-used ISA codes, such as PowerPC and x86, so that programs run on the original VLIW processor. The system exploits instruction level parallelism (ILP). It does no emulation.

Transmeta's Crusoe[7] has similar mechanism to DAISY. Crusoe is based on VLIW and it has unique ISA. The processor runs CMS (Code Morphing Software) and the software dynamically translates x86 instructions to its internal ones. Different from DAISY, CMS translates only hot-spot codes and takes incremental optimization concurrently with program execution.

These systems listed above are for translation purpose into different ISA. Following systems aims at optimization.

Dynamo[8] of Hewlett-Packard translates PA-RISC binaries to PA-RISC codes for optimization purpose. Dynamo translates concurrently with emulation of PA-RISC instructions. From profiling results of emulation, it can find hot-spots and translates into optimized codes. The resulting codes are cached, thus, once the hot-spot is translated, the optimized codes are executed for acceleration.

Morph[9] of Harvard University does profiling under the cooperation with operating system, and it optimizes executed codes off-line using the results of profiling.

Deco[10] of Harvard University does run-time optimization and binary translation. Deco can re-translate optimized codes according to change of the program's behavior.

BOA[11] of IBM focuses EPIC-style approach, that aims at high clock frequency by simplified hardware, abandoned out-of-order superscalar mechanisms like PowerPC. BOA optimizes instruction scheduling for such architecture by using binary translation technology. Where DAISY translates only once, BOA continuously monitors the behavior of execution paths and does run-time optimization.

Java's HotSpotVM[12] collects profile information during interpretive execution. When it detects a hot-spot, the hot codes are translated into native codes. The VM uses the translated binaries so that it accelerates performance.

UQBT[13] is a framework of retargetable binary translation. The system's unique point is that, theoretically, it enables any ISA codes translated into any other ISA. Currently it supports SPARC, x86, and Java bytecode.

All systems shown above assume single-thread code and none aims at performance enhancement by multi-threading.

7 Concluding Remarks

In this paper, we proposed a binary translation and optimization system that enables existing binary codes to run on the future multithreaded processors. We first discussed about the basic assumption on the target architecture and the essential requirements for single-thread binary codes to be translated to multithreaded codes.

The proposed system roughly consists of static translator and optimizer (STO) and dynamic translator and optimizer (DTO). STO initially translates an input binary code to the multithreaded one. DTO handles the dynamic behavior of the translated program and optimizes according to profiling results at run-time.

A pilot binary translator was built for the sake of preliminary evaluation. Programs used for evaluation are integral calculation in a *sin* trigonometric function using a trapezoidal equation and inner product calculation. The results show overheads in thread pipelining and, if each thread has sufficient calculation, the overhead can be negligible and the speed-up, linear to the number of thread units, is achieved.

At the present time, DTO is not completed. We will continue to develop the proposed system and show effectiveness in practical programs such as SPEC benchmarks.

Acknowledgement This research was supported in part by the Grant-in-Aid for Scientific Research (C) of Japan Society for Promotion of Science (JSPS) No.12680328.

References

- [1] J. Y. Tsai, J. Huang, and et al., "The Superthreaded Processor Architecture," *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures*, Vol. 48, No. 9, 1999.
- [2] D. F. Bacon, S. L. Graham and O. J. Sharp, "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys*, Vol. 26, No. 4, pp. 345–420, 1994.
- [3] M. D. Smith, "Overcoming the Challenges to Feedback-Directed Optimization," *Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, 2000.
- [4] J. Huang. "The SIMulator for Multi-threaded Computer Architecture(SIMCA), Release 1.2.," <http://www-mount.cs.umm.edu/Research/Ag-assiz/simca.html>.
- [5] R. J. Hookway and M. A. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation," *Digital Technical Journal*, Vol.9, No 1, pp. 3–12, 1997.
- [6] K. Ebcioglu, E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *Proceedings of 24th Annual International Symposium on Computer Architecture*, pp. 26–37, 1997.
- [7] A. Kaliber, "The Technology Behind Crusoe Processors," 2000, URL: <http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf>.
- [8] V. Bala, E. Duesterwald, S. Banerji, "Dynamo: A Transparent Dynamic Optimization System," *Proceedings of Programming Language Design and Implementation*, 2000.
- [9] X. Zhang, Z. Wang, and et al., "System Support for Automatic Profiling and Optimization," *Proceedings of 16th Symposium on Operating Systems Principles*, 2000.
- [10] E. Feigin, "A Case for Automatic Run-Time Code Optimization," Senior thesis, Harvard College, Division of Engineering and Applied Sciences, 1999.
- [11] S. Sathaye, P. Ledak, and et al., "BOA: Targeting Multi-Gigahertz with Binary Translation," *Workshop on Binary Translation (Binary99)*, 1999.
- [12] Sun Microsystems, "Java HotSpot™ Technology," URL: <http://java.sun.com/products/hotspot/>
- [13] C. Cifuentes and M. Van Emmerik, "UQBT: Adaptable Binary Translation at Low Cost," *Computer*, Vol. 33, No. 3, pp. 60–66, 2000.
- [14] K. Ootsu, T. Ono, T. Baba, "A Methodology for Multithreading with Binary Translation," *IPSJ SIG Notes*, Vol.2001, No.10, pp.41–46, January 2001 (in Japanese).
- [15] T. Ono, K. Ootsu, T. Yokota, T. Baba, "Preliminary Evaluation of Binary-Level Multithreading," *IPSJ SIG Notes*, Vol.2001, No.76, pp.183–188, August 2001 (in Japanese).