

Preliminary Evaluation of a Binary Translation System for Multithreaded Processors

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

Department of Information Science, Faculty of Engineering,
Utsunomiya University
e-mail: kim@is.utsunomiya-u.ac.jp

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.

For the multithreaded processors to be widely accepted, we must solve the following two problems. First, who (what) produce multithreaded codes? And second, we should abandon plenty of existing (single-thread) binary codes if their source codes are not available. 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.

Based upon these background, we have proposed a binary translation and run-time optimization system[1] for multithreaded processors. Our system translates existing single-thread codes into multithreaded ones both statically and dynamically in a machine-independent fashion.

Binary Translation System

Figure 1 illustrates the configuration of our proposed system. In Figure 1, The **STO (Static Translation and Optimizer)** executes the static binary translation and optimization, and the **DTO (Dynamic Translation and Optimizer)** performs the dynamic (run-time) binary translation and optimization. Both sub-systems input the source bi-

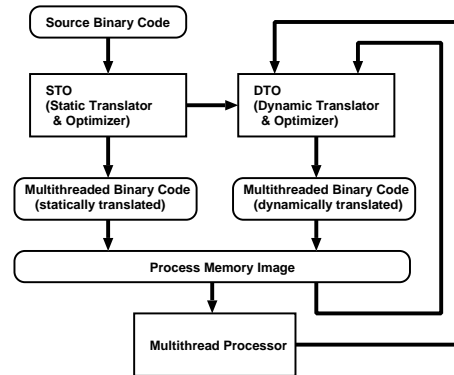


Figure 1: System Configuration

nary code and translate it to multithreaded one. STO's mission is to prepare translated binary codes for the multithreaded processor before the program is executed. However, it could not complete the translation. The remaining translation should be done at run-time by DTO. 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. Then, DTO substitutes the original code to the optimized one.

STO acquires useful information such as code analysis information, control- and data-flow information, during the analysis of input binary code. DTO can make full use of these information. This reduces overheads in run-time optimization.

In order to evaluate the basic idea of the binary translation from single-thread codes into multithreaded ones and their optimization, we have built a pilot translation system[2]. Figure

2 shows the block diagram of the pilot system.

In Figure 2, **MultiThread Code Generator** translates source binary code into multithreaded one using the thread pipelining model[3]. And it also performs various optimizations such as loop-unrolling. Finally, it generates the multithreaded partial binary code and the resulting partial binary code is merged into the original (single-thread) binary code by **Binary Patcher**. Binary Patcher merges two binaries and inserts the 'jump' instructions to the top of the original code region within the original binary code. Thus, the translated code is executed in the multithreaded manner, while the rest of the program is executed in the single-thread manner.

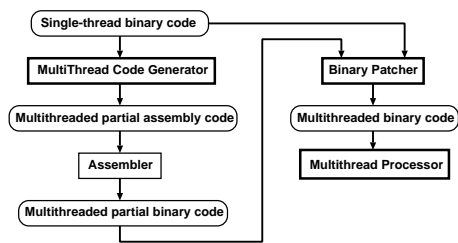


Figure 2: Pilot System

Evaluation

We evaluated the effectiveness of the binary-level multithreading system using an inner product calculation program.

We have assumed that the target multithreaded processor follows the architecture of SIMCA[3], that is a simulator based on the thread pipelining model. Original binary code is generated by *gcc* cross compiler (version 2.7.2.3) with “-O2” option. Performance was measured as execution cycles of the hot-spot loop by using the SIMCA. By comparing the number of execution cycles, speed-up ratio was calculated.

Figure 3 illustrates the evaluation result for the inner product calculation program. In Figure 3, we can find that speed-up ratio is limited by unrolling factor. The inner product calculation contains the small amount of operations. Thus thread pipelining overheads could not be hidden unless sufficient loop-unrolling is applied. This result reveals that the efficiency in thread pipelining heavily depends on the grain size of calculation.

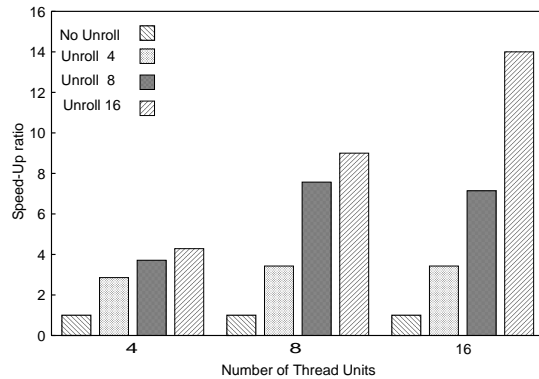


Figure 3: Inner Product Calculation

Conclusion

A pilot binary translator, that is a static part of our system[1], was built for the sake of preliminary evaluation. Using the inner product calculation program, we have evaluated the effectiveness of the binary-level multithreading system. 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, we are developing the dynamic part of the binary translation system. And we will continue to develop our system and show effectiveness in practical programs such as SPEC benchmarks.

References

- [1] 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).
- [2] K. Ootsu, T. Yokota, T. Ono, T. Baba, “A Binary Translation System for Multithreaded Processors and its Preliminary Evaluation,” *Fifth Workshop on Multithreaded Execution, Architecture and Compilation*, pp.13–22, December 2001.
- [3] J. Y. Tsai, J. Huang, and et al., “The Supertthreaded Processor Architecture,” *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures*, Vol. 48, No. 9, 1999.