Decoupling Computation and Result Write-Back for Thread-Level Parallelization

Hiroaki Hirata, Kyoto Institute of Technology, Japan
Atsushi Nunome, Kyoto Institute of Technology, Japan
https://orcid.org/0000-0003-1911-3075

ABSTRACT

Thread-level speculation (TLS) is an approach to enhance the opportunity of parallelization of programs. A TLS system enables multiple threads to begin the execution of tasks in parallel even if there may be the dependency between tasks. When any dependency violation is detected, the TLS system enforces the violating thread to abort and re-execute the task. So, the frequency of aborts is one of the factors that damage the performance of the speculative execution. This article proposes a new technique named the code shelving, which enables threads not to need to abort. It is available not only for TLS but also as a flexible synchronization technique in conventional and non-speculatively parallel execution. The authors implemented the code shelving on their parallel execution system called Speculative Memory (SM) and verified the effectiveness of the code shelving.

KEYWORDS

Code Shelving, Multithreading, Parallel Execution Model, Parallel Processing, Parallel Programming, Thread-Level Parallelism, Thread-Level Speculation

INTRODUCTION

Shared-memory multiprocessors are commonplace in current computers, but to benefit from the multiplicity of processing hardware units, programmers must write parallel programs. But it is a tremendously larger burden for a programmer to write an efficient parallel program—that is, multithreaded program—than to write a sequential program—that is, single-thread program. Nowadays the parallelization technology at the instruction level has been almost fully developed, and so for the single-thread program, compilers automatically extract instruction-level parallelism and optimize the program by code scheduling. On the other hand, at the thread level, current state-of-art compilers can merely parallelize only the limited and easy type of loops. Some compilers may be armed with the function to enable a programmer to specify thread-level parallel execution in a program. This is very helpful for a programmer to avoid of describing some typical codes for thread handling explicitly, but the essential parallelization still remains to be responsible to a programmer. And the possibility of specifying the parallel execution is also very limited.

DOI: 10.4018/IJSI.2020070102

Copyright © 2020, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
The authors have been developing a parallelization system based on a concept named speculative memory (SM) (Hirata, 2016; Fujisawa, 2017; Fujisawa, 2018). SM was originally developed for thread-level speculation, but now, its objective is not only to enable speculative execution of multiple threads but also to provide a broader framework to enlarge the possibility of thread-level parallelization. So, SM also supports purely parallel execution as well as speculatively parallel execution at the thread level. That is, in SM, speculative execution is merely one of the important choices for parallelization. The authors have already implemented the first version of software-based SM (SSM) as an SSM library, at the same time while they are developing a hardware-based SM (HSM) system. On the other hand, they also keep exploring functional extensions of SM. This article is concerning one of such extensions, code shelving.

The code shelving is a mean to remove the negative effect on the execution performance due to the inter-thread dependency. It was presented for the first time in the literature (Matsunaga, 2019) by the authors. That literature presented preliminary results of the performance evaluation, whereas this article presents new results of the performance evaluation, which is different from the previous literature in the following two points. First, this article optimizes programs so that the code shelving can be used more efficiently. Second, in the previous literature, the SM library included codes for collecting statistic information helpful for the performance evaluation, and this enlarged the parallelization overhead. In this article, such codes are all removed, and so pure effectiveness of the code shelving can be evaluated. What is more, this article analyzes and discusses about the effectiveness of parallelization using code shelving more deeply.

BACKGROUND

Many techniques for parallelizing sequentially-coded programs have been developed (Zima, 1990). Most of them, which are intended to be used by parallelizing compilers, analyze the dependencies among iterations of a loop in a program and execute iterations only if it is assured that they have no dependencies on each other. When any one of the iterations may depend on other iterations—even if many other iterations are independent—however, no compiler can parallelize the loop.

Thread-level speculation (TLS) (Hirata, 1992; Marcuello, 1999; Hammond, 2000; Vijaykumar, 2001; Steffan, 2005; Praun, 2007; Tian, 2008; Mehrara, 2009; Hertzberg, 2011; Odaira, 2014; Shoji, 2016) is an approach to parallelize a loop such a conventional and conservative compiler cannot parallelize. TLS assumes there are no dependencies among the iterations of a loop and make threads execute those iterations speculatively in parallel. The TLS system must trace the memory locations each thread accessed to detect the dependency dynamically. For example, if a thread $T_a$ executing an earlier iteration writes to the memory location from which another parallel thread $T_b$ executing a later iteration reads, it will be apparent that there is a dependency between the two iterations. And if $T_b$ reads the value before $T_a$ writes, this should be detected as a dependency violation. If the TLS system detects such a violation, it enforces the thread—$T_a$ in this case—which will be violating the dependency to abort and restart. Only the threads that did not violate dependencies can commit—that is, write back their speculatively modified data to the memory. The commitment should be performed in program order. Thus, the serialization of the commitment is one of the differences from purely parallel execution.

To shorten the execution time of a program by using TLS, it is essential that threads seldom abort in practice. One simple method to prevent threads from aborting may be the insertion of the synchronization codes, but this has the two following drawbacks. First, the waiting time for the synchronization can degrade the performance. Second, in the case that it is difficult to statically analyze which iteration of a loop depends on another iteration, it is impossible to insert a pair of the synchronization codes for posting and waiting. As a result, the insertion of the synchronization codes cannot be a fundamental solution since TLS is originally a solution for the case that the dependency among iterations is not statically analyzable.
A Decision Tree Analysis of a Multi-Player Card Game With Imperfect Information
Masato Konishi, Seiya Okubo, Tetsuro Nishino and Mitsuo Wakatsuki (2018).
*International Journal of Software Innovation* (pp. 1-17).
www.igi-global.com/article/a-decision-tree-analysis-of-a-multi-player-card-game-with-imperfect-information/207722?camid=4v1a

A Cross-Platform Architecture with Intelligent Agents for Dynamic Processes and Services Composition
www.igi-global.com/chapter/a-cross-platform-architecture-with-intelligent-agents-for-dynamic-processes-and-services-composition/135222?camid=4v1a