Design Space Exploration for Implementing a Software-Based Speculative Memory System

Kohei Fujisawa, Kyoto Institute of Technology, Kyoto, Japan
Atsushi Nunome, Kyoto Institute of Technology, Kyoto, Japan
Kiyoshi Shibayama, Kyoto College of Graduate Studies for Informatics, Kyoto, Japan
Hiroaki Hirata, Kyoto Institute of Technology, Kyoto, Japan

ABSTRACT

To enlarge the opportunities for parallelizing a sequentially coded program, the authors have previously proposed speculative memory (SM). With SM, they can start the parallel execution of a program by assuming that it does not violate the data dependencies in the program. When the SM system detects a violation, it recovers the computational state of the program and restarts the execution. In this article, the authors explore the design space for implementing a software-based SM system. They compared the possible choices in the following three viewpoints: (1) which waiting system of suspending or busy-waiting should be used, (2) when a speculative thread should be committed, and (3) which version of data a speculative thread should read. Consequently, the performance of the busy-waiting system which makes speculative threads commit early and read non-speculative values is better than that of others.

KEYWORDS

Multithreaded Programming, Parallel Architecture, Speculative Multithreading (SpMT), Thread-Level Speculation (TLS)

1. INTRODUCTION

Shared-memory multiprocessors are commonplace in current computers, and multi-core processor chips are used not only in high-end servers but also in desktop and portable computers, even embedded ones. To benefit from core multiplicity, application programmers have to write parallel programs. They need to expose the inherent parallelism in an algorithm and express it explicitly, for example, by using a multithreading library.

Many techniques for parallelizing sequentially-coded programs have been developed (Zima, 1990). Most of them 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. In some simple cases, a compiler can automatically generate excellent codes executing a loop in parallel. However, when there are inter-iteration dependencies only in a small subset of iterations and other iterations have no dependency on each other, no compiler can parallelize the loop. To parallelize such loops, we proposed a concept called speculative memory (SM) (Hirata, 2016).

Shoji, 2016) of a small segment in a program is provided mainly by hardware mechanisms—with the partial support of compilers—and is invisible to programmers. But with SM, programmers can specify the speculative execution of loop iterations explicitly in their programs. The SM system creates a thread to execute an iteration of a loop. The results of the execution of the thread are committed if the thread does not violate the dependency on other threads executing earlier iterations. If it does, the SM system forces it to abort and re-start the execution of the iteration. With this aborting capability, SM does not always require the assurance that the loop is parallelizable. Consequently, SM gives programmers more opportunities to extract the parallelism of their programs.

In this paper, we present an implementation of the SM system by software only. This software implementation of SM (SSM) is the first step of our SM project. Although we intend to implement the core part of the SM system by hardware in order to optimize the performance in the future, we demonstrate here that our SSM system can achieve the performance improvement of a program even if it is executed on a conventional multi-core processor, which has no special-purpose hardware mechanism for SM.

2. SPECULATIVE MEMORY

Figure 1 shows a program segment making a binary search tree (BST). Assume that character string data such as personal names are stored in a linear linked list and the variable list points to the first node of the list. Nodes in the list are linked by using the structure member next. For each node in the list, we call the function add_toBST(), which inserts the node into the partial BST. Nodes in the BST are linked by using the structure member left or right. When exiting from the loop in the function main(), the variable tree must have pointed to the root node of the BST that includes all nodes in the linear linked list.

It is difficult to parallelize this loop in the function main() automatically because it cannot be assured that the iterations have no dependency on each other. But since a node is always added to the leaf node of the partial BST, it may be possible to add nodes to different leaf nodes non-sequentially. So by using SM we can rewrite this program to the program shown in Figure 2. (The SM library functions used in Figure 2 are described in this paper’s Appendix.)

We call the initial (default) thread of a process the main thread. So in Figure 2 the main thread executes the function main(). Before starting the parallel execution of loop iterations, the main thread calls the SM library function sm_init_nthreads() in order to create threads we call subthreads. We designed the SM system so that each subthread executes an iteration of the parallelization target loop. Subthreads themselves are not “speculative” at the level of a multithreading library or an operating system. With regard to the speculation state, we distinguish two types of subthreads. One is the primary thread, which executes the earliest iteration of the loop non-speculatively. The other subthreads are speculative threads, which executes iterations in speculative fashion.

In the body of the parallelization target loop, the main thread gets the pointer to the descriptor of an idle subthread by calling the SM library function sm_next_thread(). If there is no idle subthread, the main thread waits in sm_next_thread(). After that, the main thread invokes the speculative execution by calling the SM library function sm_start_speculation(). That is, the main thread forces the subthread selected by sm_next_thread() to start the speculative execution of the user function add_toBST(), which is given as an argument of sm_start_speculation(). The arguments for the function add_toBST() are also specified as arguments of sm_start_speculation(). Since control is returned from sm_start_speculation() without waiting for the subthread to complete the execution of add_toBST(), the main thread can start the next invocation of the speculation immediately.

A subthread executes the function add_toBST(), which is given as the argument of sm_start_speculation(). To detect inconsistencies, we must mark the memory accesses that might cause data hazards. In the program of Figure 2, accesses to the members left and right of Node and to the header variable tree are marked with sm_load_ptr() or sm_store_ptr(). These SM library functions
Control Automation to Reduce Costs of Control
[www.igi-global.com/article/control-automation-to-reduce-costs-of-control/103316?camid=4v1a](www.igi-global.com/article/control-automation-to-reduce-costs-of-control/103316?camid=4v1a)

Developing Map-Based and Location-Aware Collaborative Applications for Mobile Users
[www.igi-global.com/chapter/developing-map-based-location-aware/66497?camid=4v1a](www.igi-global.com/chapter/developing-map-based-location-aware/66497?camid=4v1a)