Continuous Speculative Program Parallelization in Software

Chao Zhang[‡], Chen Ding[†], Xiaoming Gu[†], Kirk Kelsey[†],
Tongxin Bai[†], Xiaobing Feng[⋆]

[†]Department of Computer Science, University of Rochester
[⋆]Key Laboratory of Computer System and Architecture,
Institute of Computing Technology, Chinese Academy of Sciences
[‡]Intel China Research Center

Abstract

This paper addresses the problem of extracting coarse-grained parallelism from large sequential code. It builds on *BOP*, a system for software speculative parallelization. *BOP* lets a user to mark *possibly parallel regions (*PPR*)* in a program and at run-time speculatively executes *PPR* instances using Unix processes. This short paper presents a new run-time support called continuous speculation, which fully utilizes available parallelism to tolerate differences in *PPR* task size and processor speed.

Categories and Subject Descriptors D.3.4 [*Software*]: PROGRAMMING LANGUAGES—Processors

General Terms Languages, Performance

Keywords Software speculative parallelization

1. Introduction

The initial *BOP* system used *batch speculation*, which spawns k tasks on k processors and stops new spawning until the current batch of tasks finish. Batch speculation can cause hardware underutilization for at least four reasons.

- *Uneven task size* In the extreme case, one task takes much longer than other tasks, and all $k - 1$ processors will be idle waiting.
- *Inter-spawn delay* The delay is caused by the time spent in executing the code between *PPR* blocks. With such delay, even tasks of the same size do not finish at the same time, again leaving some processors idle.
- *Sequential commit* Virtually all speculative systems check correctness in the sequential order, one task at a time. Later checks always happen at a time when the early tasks are over and their processors are idle. In fact, this makes full utilization impossible with a single-group activity window.

PPoPP'10, January 9–14, 2010, Bangalore, India.
ACM 978-1-60558-708-0/10/01.

- *Asymmetrical hardware* Not all processors have the same speed. A hyperthreaded core has a different speed when running one or two tasks. As a result, identical tasks may take a different amount of time to finish.

In this short paper, we describe *continuous speculation*, which starts a speculation task whenever a processor becomes available. For full details of the system, its evaluation, and a discussion of related work, we refer the reader to a full technical report [2].

2. Continuous Speculation

2.1 Basic Concepts

Process-based speculation A process has several useful properties for speculative execution. A process can be used to start a speculation task anywhere in the code. It is easier to monitor when using the page protection mechanism. Processes are well isolated from each other, allowing easy error recovery. Modern OS performs copy-on-write, which enables on-demand data replication.

Group commit In a process-based implementation, a correct speculation task must explicitly copy the changed data to later tasks. For a group of k concurrent tasks on k processors, each task can copy its data in two ways: either to the next task or to the last task of the group. The latter choice has the minimal, linear cost, where each changed datum is copied only once. We call this choice *group commit* or *group update*.

Implicit synchronization A parallel task should finish before its result is used. The use point is statically specified in languages such as OpenMP, Java, and Cilk. For *BOP* the first use point of a *PPR* task happens at the first parallel conflict, which is detected automatically by the speculation substrate.

Understudy-based recovery Speculation may fail in two ways: either a speculative task incurs a conflict, or it takes too long to finish. *BOP* overcomes both types of failures with the use of an *understudy* task. The understudy task starts from the last correct state and non-speculatively executes the program alongside the speculative execution. It executes at the same speed as the unmodified sequential code. Parallel execution succeeds if it incurs no conflict and finishes faster than the understudy.

2.2 Dual-group Activity Window

A *dual-group activity window* divides the active tasks into two groups based on their spawning order in such a way that every task

in the first group is earlier than any task in the second group. When a task in either group finishes execution, the next task enters the window into the second group. When all tasks in the first group complete the group commit, the entire group exits, and the second group stops expanding and becomes the first group in the window.

For a speculative system that requires group commits, the dual-group activity window is both a necessary and sufficient solution to maximize hardware utilization. Next we briefly describe four components of the design.

Token passing For a system with p available processors, we need to reserve one for competitive recovery (as discussed later in this section). We use $k = p - 1$ *processor tokens* to maintain k active tasks. The first k tasks in a program form the first group. They start without waiting for a token but finish by passing a token. Each later task waits for a process token before executing and releases the token after finishing. To divide active tasks into two groups, we use a *group token*, which is passed by the first group after a group commit to the next new task entering the activity window. The new task becomes the first task of the next group.

Variable vs. fixed window size The activity window can be of a fixed size or a variable size. Token passing bounds the group size from below to be at least k, the number of processor tokens. It does not impose an upper bound because it allows the second group to grow to an arbitrary size to make up for the lack of parallelism in the activity window. Conceivably there is a danger of a runaway window growth as a large group begets an even larger group. To bound the window and group size, we augment the basic control by storing the processor tokens when the second group exceeds a specific size. The window expansion stops until the first group finishes and releases the group token. In evaluation, we found that token passing creates an "elastic" window that naturally contracts and does not enter a run away expansion. However, the benefit of variable over fixed window size is small in our tests.

Triple updates Upon successful completion, a speculative task needs to commit its changes to shared data. In a process-based design, this means copying modified pages from the task process to other processes. In continuous speculation, a modified page is copied three times in a scheme we call *triple updates*. Consider group g. The first update happens at the group commit and copies modified pages in all but the last task of g to the last task. The second update happens before group $g+2$ and copies the data changes in g to $g+2$. The third update of group g happens at the end of group $g+1$ and copies the data changes in g to $g+1$. In process-based design, inter-task copying is implemented by communication pipes. We need 3 pipes for each group. Since the activity window has two groups, we need a total of 6 pipes, independent of the size of the activity window.

Competitive recovery In continuous speculation, the understudy task is started after the first task, as illustrated in Figure 1. As the activity window advances, the understudy task is re-started after each task group. An understudy task is always active (after the first task) as long as speculation continues. As a result, it requires a processor constantly, leaving one fewer processor for parallel execution. We refer to this as the "missing processor" problem in continuous speculation.

A question is where to draw the finish line for the parallel-sequential race. Since the understudy is run with two speculation groups, it has two possible finish lines: the completion of the first group or the completion of both groups. We choose the second because favoring speculation means that it won't give up the competition as long as there is a chance the parallel execution may finish early. A victory by the understudy, on the other hand, means no performance improvement and no benefit from parallel execution.

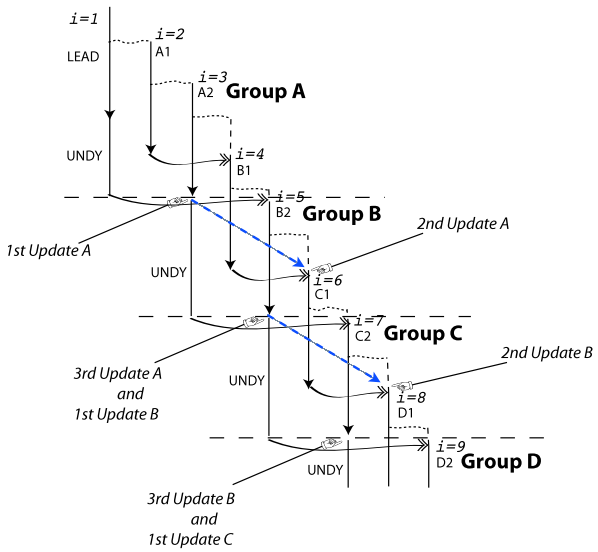

Figure 1. Example execution of 9 *PPR* tasks on 3 processors, showing the use of token passing and triple updates. Processor tokens are passed along solid lines with a double headed arrow. The group token is passed along dotted lines. The hands point to the triple updates by the first two groups. Continuous speculation maintains 2 active speculation tasks and 1 understudy task (marked "UNDY") at all times.

An example Figure 1 shows an example of continuous speculation. There are 9 *PPR* tasks, represented by vertical bars marked with $i = 1, \ldots, 9$. There are 3 processors $p = 3$, so there are $k = p - 1 = 2$ processor tokens. The first 3 *PPR* tasks form the first group, A. When a task finishes, it passes a processor token, shown by a line with a double headed arrow, to a task in the next group B. When group A finishes, it passes the group token, shown by a dotted line, to start group C. The tasks enter the activity window one individual at a time and leave the window one group at a time. In the steady state, it maintains two task groups with two speculation tasks active at all times. This can be seen in the figure. For example, the token passing mechanism delays task $B1$ until a processor is available.

After finishing, group A copies its speculative changes three times: at the end of A for the understudy of B, at the end of B for the understudy of C, and at the start of C for speculation of C and later *PPR*s. The understudy tasks are marked "UNDY". An understudy task is always running during the continuous speculation.

Acknowledgments

The research is supported by the National Science Foundation (Contract No. CCR-0238176, CNS-0834566, CNS-0720796), IBM CAS Faculty Fellowship, and a gift from Microsoft Research. The initial system design was inspired by RingSTM [1]. We also wish to thank Jingliang Zhang at ICT for help with the implementation of *BOP*-malloc.

References

[1] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of SPAA*, pages 275–284, June 2008.

[2] C. Zhang, C. Ding, K. Kelsey, T. Bai, X. Gu, and X. Feng. A language of suggestions for program parallelization. Technical Report URCS #948, Department of Computer Science, University of Rochester, 2009.