Improving Speculative Loop Parallelization via Selective Squash and Speculation Reuse

Santhosh Sharma Ananthramu
Indian Institute of Technology, Kanpur
sharma.santhosh@gmail.com

Deepak Majeti *
Dept. of Comp. Sci.
Rice University
dm14@rice.edu

Sanjeev Kumar Aggarwal and Mainak Chaudhuri
Indian Institute of Technology, Kanpur
{ska,mainakc}@cse.iitk.ac.in

ABSTRACT

Speculative parallelization is a powerful technique to parallelize loops with irregular data dependencies. In this poster, we present a value-based selective squash protocol and an optimistic speculation reuse technique that leverages an extended notion of silent stores. These optimizations focus on reducing the number of squashes due to dependency violations. Our proposed optimizations, when applied to loops selected from standard benchmark suites, demonstrate an average (geometric mean) 2.5x performance improvement. This improvement is attributed to a 94% success in speculation reuse and a 77% reduction in the number of squashed threads compared to an implementation that, in such cases of squashes, would have squashed all the successors starting from the oldest offending one.

Categories: D.1.3 [Software]: Programming Techniques–parallel programming

General Terms: Design, Performance

Keywords: Thread-level speculation, Mis-speculation overhead

1. INTRODUCTION

The basic principle of software-only speculative parallelization [1, 2] is to execute the iterations of the loop concurrently speculating that there is no carried true dependence. However, violation of data dependency, if any, is caught on the fly with the help of additional data structures and book-keeping information. The book-keeping overhead and the lost cycles due to mis-speculations leading to squashes can impose a significant performance penalty. In this paper, we squarely focus on improving the performance of software-only speculative parallelization by reducing the cycles lost in mis-speculations.

Our solution for reducing the squash overhead involves two components, *Selective Squash* and *Speculation Reuse*. Both the components are enabled by the simple observation that retaining the state (store values) generated by a thread during the last mis-speculated iteration can be helpful in the subsequent re-execution of the iteration. Traditional speculative parallelization techniques discard these states before re-execution is initiated. We propose another improvement related to making the commit operation less lock-intensive.

*This work has been done while Deepak was at IIT Kanpur.

PACT'10, September 11–15, 2010, Vienna, Austria.
ACM 978-1-4503-0178-7/10/09.

2. FRAMEWORK

Our framework is designed after the proposals in the baseline [2]. The global array and the index matrix are same as in the baseline. The version matrix must now accommodate both read and write values of a datum. A new state called the OLD state must be introduced to help us retain the speculatively generated states. A datum in a MOD (speculatively written) state transitions to OLD_MOD state when the thread associated with the sliding window slot suffers from a squash. The access matrix must now handle the OLD states.

3. SELECTIVE SQUASH, SPECULATION REUSE AND OPTIMIZING COMMIT

We have already discussed the newly introduced OLD state in the access matrix which helps us retain the speculatively generated versions during a mis-speculated execution of a task. Our implementation attempts to reuse these versions in the re-execution that immediately follows the squash.

A *silent store* is a write to memory that does not alter the value already present at the target address X.

An *extended silent store* is a write to memory that produces the same value at the target address X, as in the previous mis-speculated execution of the same task.

In the next definition, we present our selective squash protocol. This protocol leverages the extended silent stores to reduce the number of squashes.

Extended silent store-based squash A thread t producing a value for a target address X either through a silent store or an extended silent store is not required to check its successors for dependence violation involving X if t forwards versions in the OLD state to a requesting successor.

This means that a dependence violation check is invoked only if a speculative version for a variable is generated for the first time or if the generated version does not qualify as a silent store or an extended silent store.

We now turn to our second contribution, namely, speculation reuse. We have already discussed how speculation reuse helps us define extended silent stores and how we use extended silent stores to reduce the volume of squashes. However, the extended silent stores make use of only the OLD write versions. A second benefit of speculation reuse is that it can consume the speculatively written private versions from the previous mis-speculated execution as long as the versions have not expired due to a store from a predecessor.

This reduces the interference with shared memory values being committed and opens up the opportunity of reducing the re-execution latency as long as the thread scheduler is made aware of this.

Through a careful implementation, we make the commit protocol as lock-free as possible. Our implementation needs one critical section for allocating a new window slot as opposed to two in the baseline. The following protocol achieves this and we will refer to it as the quasi-lock-free commit protocol. First, we copy valid write versions to shared memory. We then clean up the access matrix structures and mark the window slot FREE. We now make the next slot non-speculative.

4. EXPERIMENTAL RESULTS

We select five loops from the SPEC 2000, SPEC 2006, and PERFECT Club benchmark suites. From the SPEC 2006 suite, the loop at *approx_cont_mgau.c:279* of 482.sphinx3 and the loop at *innerc.c:4160* of 435.gromacs have a high dependence density. The loop at *blocksort.c:551* of 401.bzip2 has a moderate dependence density. From the SPEC 2000 suite, the loop at *vbrender.c:897* of 177.mesa exhibits a low dependence density. From the PERFECT Club suite, the loop *interf_1000* of MDG has a zero dependence density. The dependence density of these loops are expressed only qualitatively.

We run our experiments on a multiprocessor system which has four Intel quad-core E7330 Xeon processors (2.40 GHz 2x3 MB L2 cache). The shared main memory size is 32 GB and the system runs Red Hat Enterprise Linux Server 5.3. The applications are compiled with the Intel C++ and Fortran Compilers 11.0 with level 2 optimizations. We use the standard production data sets provided with the applications. OpenMP 2.0 is used to put the parallelization directives around the selected loops.

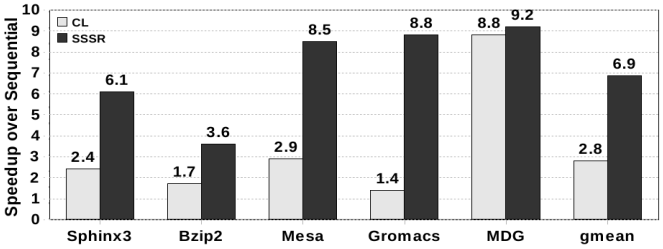

Figure 1: Performance on sixteen threads.

Figure 1 presents the self-relative speedup comparison between the baseline and our implementation for the selected loops. The baseline is labeled CL after the proposers [2]. Our implementation is labeled SSSR (Selective Squash Speculation Reuse). Note that SSSR also turns on our quasi-lock-free commit optimization. For each application loop, the speedup over a sequential execution of the loop is shown for sixteen threads. The last two bars present the geometric mean speedup over sequential execution for CL and SSSR on sixteen threads. For 177.mesa, 482.sphinx3, 401.bzip2 and 435.gromacs, SSSR is successful in dramatically improving the scalability compared to CL. Finally, for MDG, the two schemes deliver almost the same performance. Overall, the geometric mean speedup delivered by CL for these loops is 2.8, while SSSR exhibits a speedup of 6.9 on sixteen threads. This translates to a parallel efficiency of 0.17 in CL and 0.43 in SSSR for the selected loop nests.

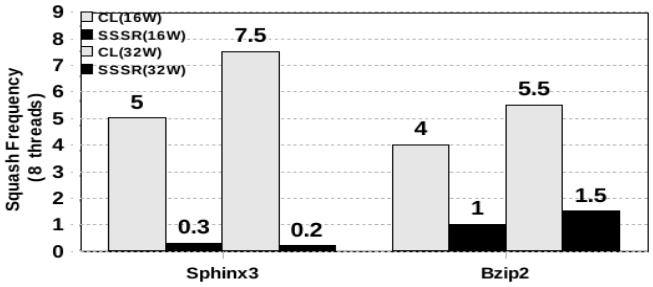

Figure 2: Frequency of squashes for 8 processors.

Figure 2 presents the frequency of squashes (ratio of number of squashes generated to the number of iterations) for the selected loops in sphinx3 and bzip2. The number of threads are set to 8 and two windows of size 16 and 32 are considered. The frequency of squashes generated by CL is much more than the squashes generated by SSSR. The frequency of squashes also increases with CL as the window size increases. This is because CL squashes all the successive threads. So a higher window size implies a higher number of resultant squashes. On the other hand, the squash frequency in SSSR is less because SS in SSSR squashes only those threads which have violated a dependency. The rest of the threads are spared. Finally, when using SSSR, the number of squashes tend to become independent of the window size. This makes SSSR more scalable.

Table 1: Reuse and squash statistics for SSSR

Attribute	sphinx3	bzip2	mesa	gromacs
Silent store	1%	0%	0%	0%
Ex. silent store	100%	0%	100%	84%
Successors spared	99%	75%	99%	35%

Table 1 further quantifies the benefits of speculation reuse and selective squash. Overall, we find that 94% of silent store versions and extended silent store versions can be reused, while 77% of successor threads can be spared by supporting selective squash.

5. CONCLUSIONS

In this paper, we have explored the benefits of selective squash and speculation reuse in the context of software-only speculative loop parallelization. These techniques are enabled by value-based dependence check, extended silent stores, and retention of speculatively consumed and generated versions in the last mis-speculated execution of a task. As a third contribution, we bring the commit operation closer to being lock-free.

6. REFERENCES

[1] L. Rauchwerger and D. Padua, "The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *PLDI '95*, pp. 218–232, 1995.

[2] M. Cintra and D. R. Llanos, "Design space exploration of a software speculative parallelization scheme," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 6, pp. 562–576, 2005.