Exclusive Squashing for Thread-Level Speculation

Alvaro García-Yágüez
Dpto. de Informática
Univ. de Valladolid, Spain
alvarga87@gmail.com

Diego R. Llanos
Dpto. de Informática
Univ. de Valladolid, Spain
diego@infor.uva.es

Arturo González-Escribano
Dpto. de Informática
Univ. de Valladolid, Spain
arturo@infor.uva.es

ABSTRACT

Speculative parallelization is a runtime technique that optimistically executes sequential code in parallel, checking that no dependence violations appear. In this paper, we address the problem of minimizing the number of threads that should be restarted when a data dependence violation is found. We present a new mechanism that keeps track of inter-thread dependencies in order to selectively stop and restart offending threads, and all threads that have consumed data from them. Results show a reduction of 38.5% to 81.8% in the number of restarted threads for real application loops and up to a 10% speedup, depending on the amount of local computation.

Categories and Subject Descriptors

D.1.3 [**Concurrent programming**]: Parallel programming

General Terms

Experimentation

Keywords

Loop-based Parallelization, Speculative Parallelization.

1. BACKGROUND

Speculative parallelization (SP), also called Thread-Level Speculation (TLS) or Optimistic Parallelization [4] assumes that sequential code can be optimisticallly executed in parallel, and relies on a runtime monitor to ensure that no dependence violations are produced. A dependence violation appears when a given thread generates a datum that has already been consumed by a successor in the original sequential order. In this case, the results calculated so far by the successor (called the offending thread) are not valid and should be discarded. Early proposals [3, 5] stop the parallel execution and restart the loop serially. Other proposals stop the offending thread and all its successors, re-executing them in parallel [1, 2].

Our contribution keeps track of inter-thread dependencies, restarting only threads which are known to have consumed values from the offending thread. We have implemented this solution upon a state-of-the-art, software-based speculative parallelization scheme, and evaluate it with five different applications that present non-analyzable loops at compile time.

HPDC'11, June 8–11, 2011, San Jose, California, USA.
ACM 978-1-4503-0552-5/11/06.

2. HANDLING DEPENDENCE VIOLATIONS

If a dependence violation occurs, the runtime monitor should decide what to do with the parallel execution. This lead us to a design space for squashing policies that can be resumed in the following alternatives: (1) Stop parallel execution, simply discarding the parallel work done so far and restarting the loop serially; (2) Stop and restart the offending thread and all its successors (we call this alternative "inclusive squashing"); (3) Only stop the offending threads, and recursively, successors that have consumed *any* speculative variable generated by them (we call it "exclusive squashing"); and (4) to keep track of the dependence graph of every single speculative variable being read or written to squash only those threads that have consumed wrong values ("perfect squashing"). In this paper we have implemented an exclusive squashing mechanism, comparing its behaviour with an implementation of inclusive squashing described in [1]. As far as we know, this is the first working proposal of exclusive squashing in the bibliography.

3. EXCLUSIVE SQUASHING MECHANISM

Our exclusive mechanism extends a previous inclusive proposal [1] with a boolean $W \times W$ matrix called *consumer_list* that keeps track of inter-thread dependences. Therefore, once a violation dependence arises it is possible to selectively decide whether a thread should be restarted or not.

With the aim of explaining the general behavior of this new solution, Fig. 1 shows an execution example. Assuming we have W running threads, a possible sequence of events will be:

1. Thread W speculatively loads element D3 from the speculative structure (event 1.1 in the figure). As none of its predecessors have accessed the value yet, datum D3 would be forwarded from the reference value. Afterwards thread 2 loads the same element D3 forwarding it from the reference value (event 1.2).

2. Thread 2 speculatively writes element D1 to the speculative structure (event 2). After writing the new value, thread 2 searches for dependence violations (not shown). None of its successors have used that datum yet so no squashes are performed.

3. Thread 3 speculatively loads element D1. To do so, it searches backwards to find the most up-to-date value available. Thread 2 has the value, so thread 3 writes in *consumer_list[3][2]* to mark that it will consume a

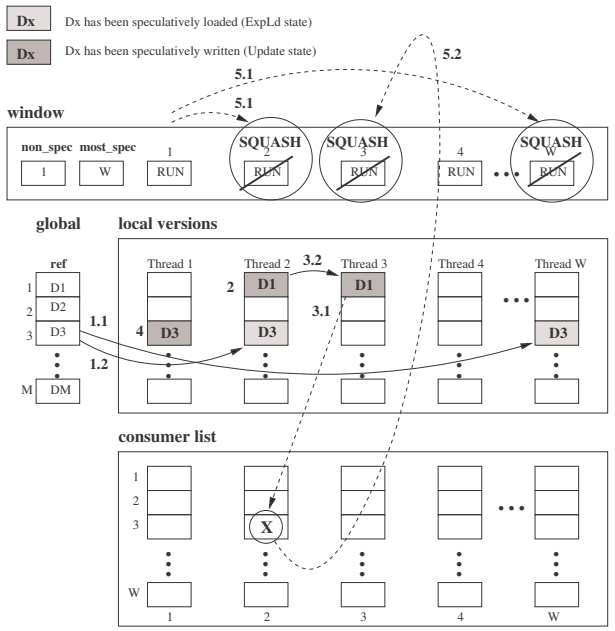

Figure 1: Execution example (see text). AM data structure is not shown for simplicity.

value from thread 2 (event 3.1). After that, thread 3 forwards datum D1 from thread 2 (event 3.2).

4. Thread 1 speculatively writes element D3 (event 4).

5. After writing the value, thread 1 starts the search for potential dependence violations. Since thread 2 have consumed the value, a squash operation takes place. Firstly (event 5.1), it squashes all threads that have incorrectly consumed the value D3 (in our example, threads 2 and W). Then, it looks at the *consumer_list* columns that correspond to slots owned by squashed threads, looking for any thread that has consumed a value from them. In our example (event 5.2), thread 3 is also squashed, and its *consumer_list* column is also checked. No additional squashes are performed.

6. This squash operation generates three bubbles in the sliding window, that will be assigned in order to the three threads that have been squashed (not shown).

In this example an inclusive squashing would produce $W - 1$ re-executions while exclusive squahing only restarts three threads. Note that thread 2 and thread W have been squashed because they have consumed an incorrect value. However, we have also squashed thread 3 because it has consumed value D1 from thread 2. This value D1 may be indeed correct, but to ensure this point it is neccessary a perfect squashing policy, an operation that we consider too costly.

4. EXPERIMENTAL RESULTS

We have evaluated our solution with five different applications. The first three applications have non-analyzable loops that do not suffer from dependence violations at runtime. From the PERFECT Club Benchmark suite, we have chosen `accel_10` from TREE, `muldeo_200` and `muldoe_200` from MDG. From SPECfp2000 benchmark, we have selected `interf_1000` from WUPWISE. The results show a slowdown that reaches 11.6% for TREE and as 16.2% for WUPWISE, comparing with the inclusive mechanism. In MDG both schemes perform equally well.

We have also evaluated two applications that present a high number of dependence violations: The construction of the two-dimensional Convex Hull (2D-Hull); and the construction of the two-dimensional Delaunay Triangulation (2D-DT). In 2D-Hull, exclusive squashing mechanism leads to a 38.5% reduction in the number of squashes for 16 processors. 2D-DT reaches a reduction of 81.83% in the number of squashes. However, these reductions are not noticeable in terms of speedup. In the case of 2D-Hull, any change in the solution being processed will invalidate sooner or later the work of all successors, making the inclusive policy more practical. In the case of 2D-DT, the application spends a small time in local computations comparing to the global-data accesses. We have overloaded the main loop with additional local computation, increasing the sequential iteration time by 10×, 23× and 41× factors. For these overloaded versions, the exclusive squashing mechanism improves the inclusive scheme by 7.7% to 10.0% in terms of speedup.

5. SUMMARY

Our experimental results with exclusive squashing show a reduction in the number of squashes ranging from 10% for 4 threads up to 85% for 16 threads. We have also found that the usefulness of this mechanism in terms of speedup heavily depends on the cost associated to discard potentially valid work. Our current work in this field is to extend the spectrum of applications evaluated to further support this claim.

Acknowledgments

The authors would like to thank Dr. Marcelo Cintra and Dr. Belén Palop for many helpful discussions on this topic. This research is partly supported by the Ministerio de Educación y Ciencia, Spain (TIN2007-62302) and Junta de Castilla y León, Spain (VA094A08). Part of this work was carried out under the HPC-EUROPA2 project (project number: 228398), with the support of the European Community - Research Infrastructure Action of the FP7.

6. REFERENCES

[1] Cintra, M., and Llanos, D. R. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP 2003* (San Diego, CA, USA, 2003), ACM, pp. 13–24.

[2] Dang, F., Yu, H., and Rauchwerger, L. The R-LRPD test: speculative parallelization of partially parallel loops. In *IPDPS '02* (2002).

[3] Gupta, M., and Nim, R. Techniques for speculative run-time parallelization of loops. In *ACM/IEEE Conf. on Supercomputing* (San Jose, CA, 1998), pp. 1–12.

[4] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P. Optimistic parallelism requires abstractions. In *ACM PLDI '07* (San Diego, CA, USA, 2007), ACM, pp. 211–222.

[5] Rauchwerger, L., and Padua, D. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *ACM PLDI '95* (La Jolla, CA, USA, 1995), pp. 218–232.