Exclusive Squashing for Thread-Level Speculation

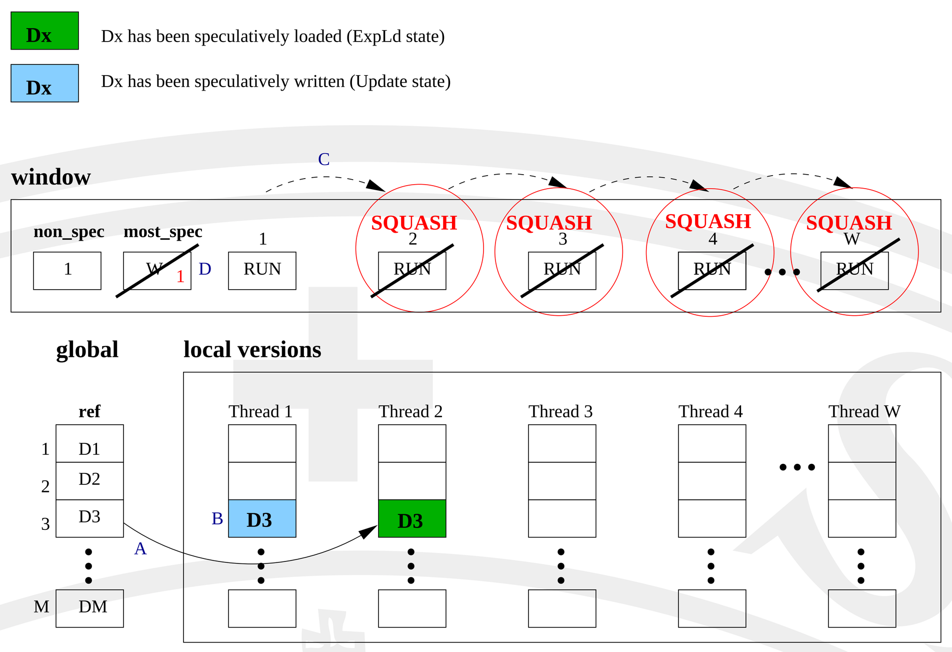
Alvaro García-Yágüez, Diego R.Llanos, and Arturo González-Escribano
Universidad de Valladolid, Spain
alvarga87@gmail.com, diego@infor.uva.es, arturo@infor.uva.es

Introduction

Speculative parallelization aims to extract loop and task-level paralelism when a compile-time dependence analysis can not guarantee that a given sequential code is safely parallelizable. Speculative parallelization optimistically assumes that the code can be executed in parallel, and relies on a runtime monitor to ensure that no dependence violation is produced.
If the runtime monitor detects a dependence violation, the runtime monitor should decide what to do with the parallel execution:

- **Restart serially**. Discarding the parallel work done so far and restarting the loop serially [1]
- **Inclusive Squashing IS**. Restarting the offending thread and all its successors [2, 3]
- **Exclusive Squashing ES (our proposal)**. Only offending threads, and recursively, successors that have consumed *any* value generated by them are restarted.
- **Perfect Squashing**. Only offending threads, and recursively, successors that have consumed *wrong* values generated by them are restarted.

Inclusive squash [3]

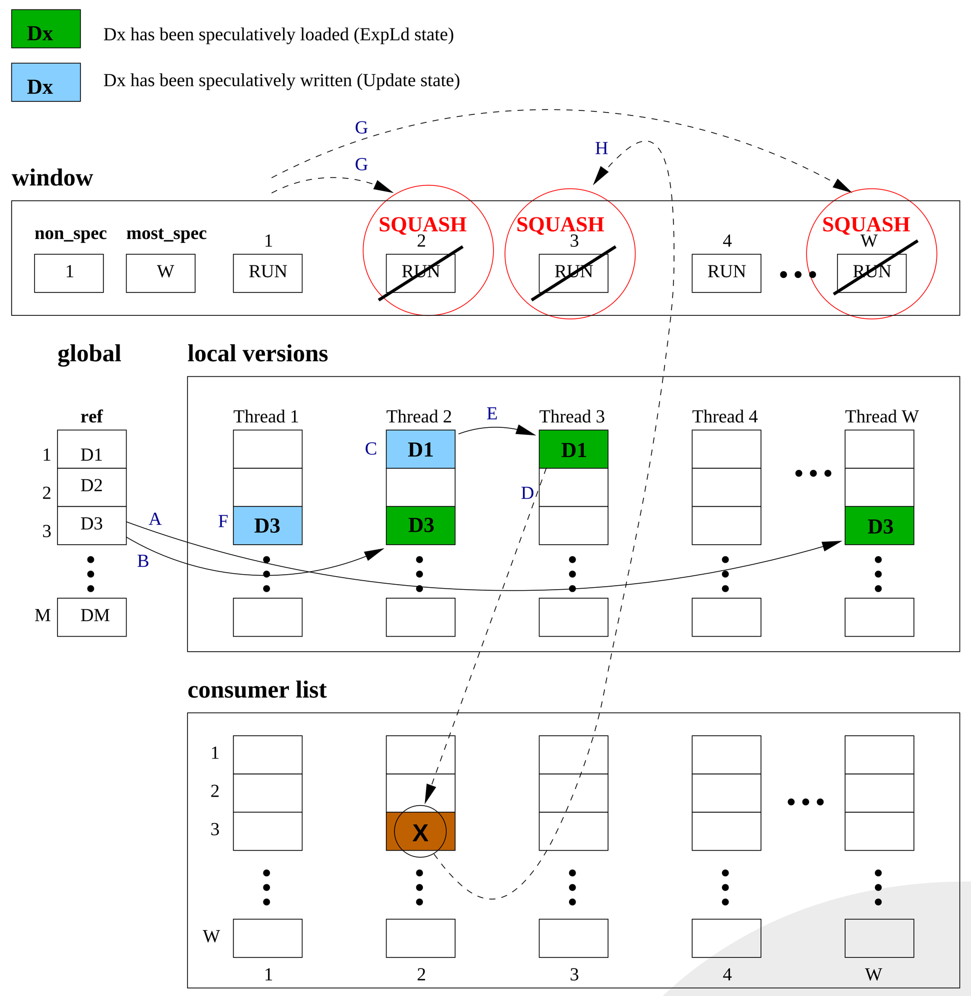


Ref. Original user data structure
Window. Holds the state of W slots where block of iterations are executed (FREE, DONE, RUNNING, SQUASHED)
Version. Stores W copies of Ref data

Execution example

A Thread 2 speculatively loads element D3 from the speculative structure.
B Thread 1 speculatively writes element D3.
C Since a dependence violation appears, Thread 2 and all successors are squashed.
D Most-speculative pointer is modified.

Our proposal: Exclusive squash

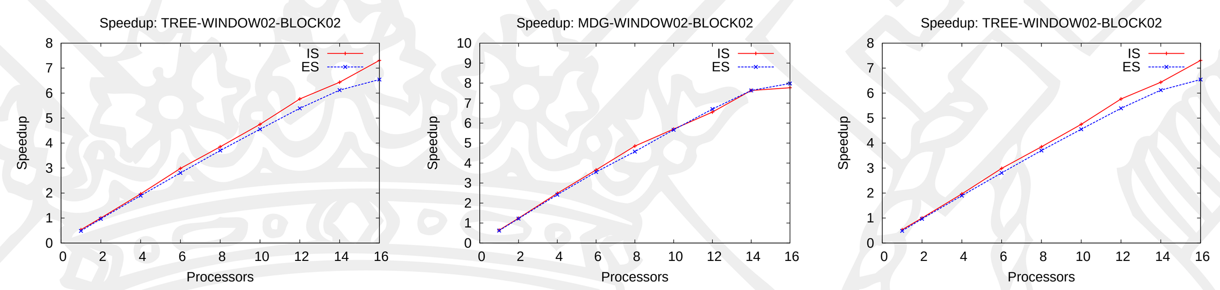


Execution example

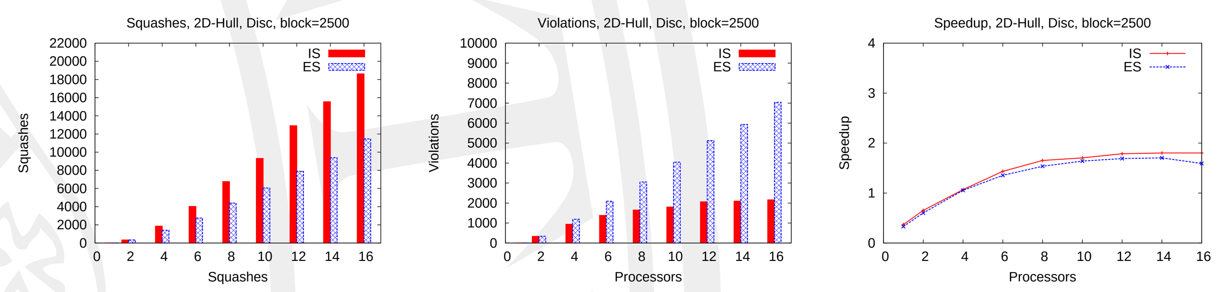
A Thread W speculatively loads element D3 from the speculative structure.
B Thread 2 loads the same element D3, forwarding it from the reference value.
C Thread 2 speculatively writes element D1 to the speculative structure; dependence violations are not found.
D Thread 3 speculatively loads element D1. Since thread 2 has the value, thread 3 writes in consumer_list[3][2] to mark that it will consume a value from thread 2.
E Thread 3 forwards datum D1 from thread 2.
F Thread 1 speculatively writes element D3.
G A squash operation takes place. Threads that have incorrectly consumed the value D3 are squashed.
H Consumer_list is checked in search for threads that have consumed any datum from squashed threads. In our example, thread 3 is also squashed, and its consumer_list column is also checked.
Note that most speculative pointer is not modified and bubbles are generated.

Results

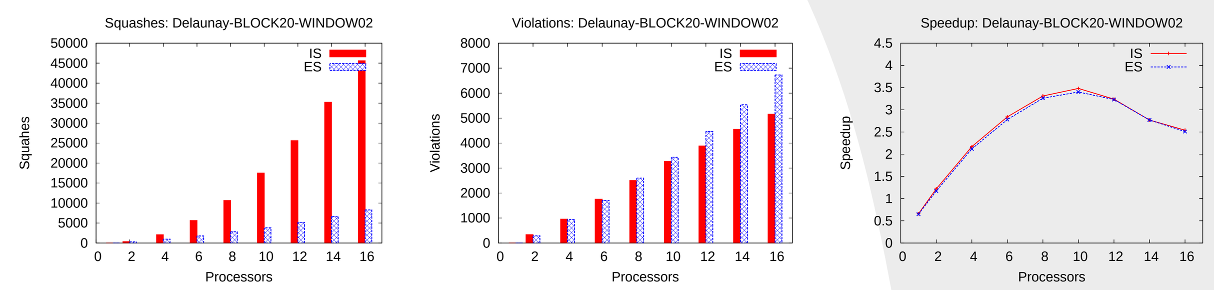
Applications without dependences



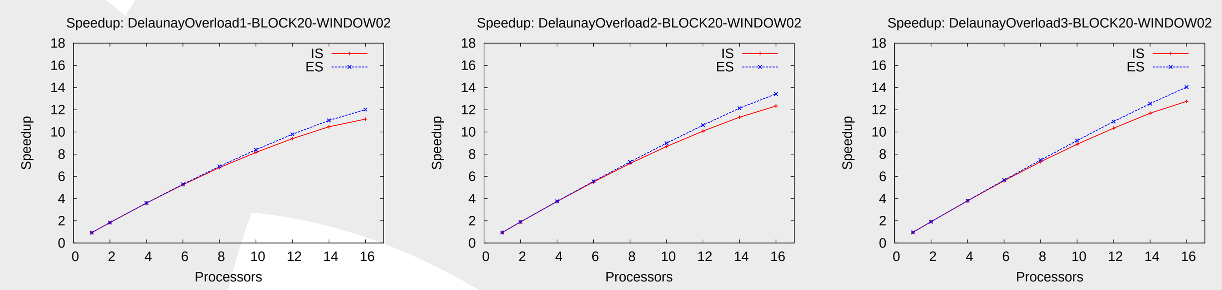
2D Convex Hull



2D Delaunay Triangulation (2DT)



2DT with overloaded computation



Conclusions

- Exclusive squashing reduces number of squashes from 10% for 4 threads, to 85% for 16 threads.
- Usefulness in terms of speedup heavily depends on the cost associated to discard potentially valid work for each application.
- Computational load is not high enough for the two applications with dependences considered: Adding an artificial load to 2DT improves the speedup in comparison to inclusive squashing policy.

This research is partly supported by the Ministerio de Educación y Ciencia, Spain (TIN2007-62302) and Junta de Castilla y León, Spain (VA094A08).

References

[1] Rauchwerger, L., and Padua, D. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming Language Design and Implementation* (La Jolla, California, United States, 1995), ACM, pp. 218–232.
[2] Dang, F., Yu, H., and Rauchwerger, L. The R-LRPD test: Speculative parallelization of partially parallel loops. In *Parallel and Distributed Processing Symposium., Proc. Intl. Par. and Distr. Processing Symposium* (2002), IEEE, pp. 20–29.
[3] Cintra, M., and Llanos, D. R. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA, 2003), ACM, pp. 13–24.