Cooperative Concurrency for a Multicore World (Extended Abstract)

Jaeheon Yi[1] Caitlin Sadowski[1] Stephen N. Freund[2] Cormac Flanagan[1]

[1] University of California at Santa Cruz [2] Williams College

Developing reliable multithreaded software is notoriously difficult, due to the potential for unexpected interference between concurrent threads. Even a familiar construct such as "`x++`" has unfamiliar semantics in a multithreaded setting, where it must in general be considered a non-atomic read-modify-write sequence, rather than a simple atomic increment. Understanding where thread interference may occur is a critical first step in understanding or validating a multithreaded software system.

Much prior work has addressed this problem, mostly focused on verifying the correctness properties of race-freedom and atomicity (see, for example, [6, 13, 10, 3, 4, 1, 9, 11, 7, 14, 15, 8, 5]). Race-freedom guarantees that software running on relaxed memory hardware behaves as if running on sequentially consistent hardware [2]. Atomicity guarantees that a program behaves as if each atomic block executes serially, without interleaved steps of concurrent threads. Unfortunately, neither approach is entirely sufficient for ensuring the absence of unintended thread interference.

We propose an alternative approach whereby all thread interference must be specified with explicit yield annotations. For example, if multiple threads intentionally access a shared variable `x` concurrently, then the above increment operation would need to rewritten as "`int t=x; yield; x=t+1`" to explicate the potential interference.

These yield annotations enable us to decompose the hard problem of reasoning about multithreaded program correctness into two simpler subproblems:

- **Cooperative correctness**: Is the program correct when run under a *cooperative scheduler* that context switches only at yield annotations?
- **Cooperative-preemptive equivalence**: Does the program exhibit the same behavior under a cooperative scheduler as it would under a traditional preemptive scheduler that can context switch at any program point?

A key benefit of this decomposition is that cooperative-preemptive equivalence can be mechanically verified, for example, via a static type and effect system that reasons about synchronization, locking, and commuting operations [17, 16]. Alternatively, cooperative-preemptive equivalence can be verified dynamically by showing that the transactional happens-before relation for each observed trace is acyclic (where a transaction is the code between two successive yield annotations) [18].

The remaining subproblem of cooperative correctness is significantly more tractable than the original problem of preemptive correctness. In particular, cooperative scheduling provides an appealing concurrency semantics with the following desirable properties:

- Sequential reasoning is correct by default (in the absence of yield annotations), and so for example "`x++`" is always an atomic increment operation.
- Thread interference is always highlighted with yields, which remind the programmer to allow for the effects of interleaved concurrent threads.

Experimental results on a standard benchmark suite show that surprisingly few yield annotations are required—only 13 yields per thousand lines of code [16]. In addition, a preliminary user study showed that the presence of these yield annotations produced a statistically significant improvement in the ability of programmers to identify concurrent defects during code reviews [12]. These experimental results suggest that cooperative concurrency is a promising foundation for the development of reliable multithreaded software.

Acknowledgements. This work was supported by NSF grants CNS-0905650, CCF-1116883 and CCF-1116825.

References

1. M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.
2. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
3. M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *Conference on Programming Language Design and Implementation (PLDI)*, 255–268, 2010.
4. J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Operating Systems Design and Implementation (OSDI)*, 1–16, 2010.
5. A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Computer Aided Verification (CAV)*, 52–65, 2008.
6. C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, 2010.
7. C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 30(4):1–53, 2008.
8. C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 293–303, 2008.
9. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 308–319, 2006.

10. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 167–178, 2003.
11. P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive correlation analysis for detecting races. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 320–331, 2006.
12. C. Sadowski and J. Yi. Applying usability studies to correctness conditions: A case study of cooperability. In *Onward! Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2:1–2:6, 2010.
13. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.
14. C. von Praun and T. R. Gross. Static detection of atomicity violations in object-oriented programs. In *Journal of Object Technology*, 103–122, 2003.
15. L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, Feb. 2006.
16. J. Yi, T. Disney, S. N. Freund, and C. Flanagan. Types for precise thread interference. Technical Report UCSC-SOE-11-22, The University of California at Santa Cruz, 2011.
17. J. Yi and C. Flanagan. Effects for cooperable and serializable threads. In *Workshop on Types in Language Design and Implementation (TLDI)*, 3–14, 2010.
18. J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 147–156, 2011.