

# Compiler Analysis and Optimization Challenges for Atomic Sections

Vivek Sarkar

The idea of providing a parallel programming abstraction for mutual exclusion that is simpler than lock-based synchronization dates back to the concept of *critical sections* from four decades ago [8]. We use the term, *atomic section*, to denote a variant of a critical section in which further restrictions are placed on the subset of data for which atomicity is guaranteed. In the X10 language [5], the restriction imposed is that all shared mutable data locations accessed within an atomic section must belong to the same *place* as the activity (thread) executing the atomic section. In the *analyzable atomic sections* [18] model, a further restriction is imposed that the addresses of all shared mutable locations be computable on entry to the atomic section. This position statement outlines three key areas for compiler challenges and research opportunities related to atomic sections:

**Program Analysis.** As with other synchronization operations, the entry and exit of an atomic section represent program points at which there may be data flow interactions between the current activity and other activities. It is important for these interactions to be modeled as precisely as possible so as not to limit analyses and optimizations that can be performed within and outside atomic sections. The use of high-level semantic information (*e.g.*, immutable [14], activity-local, non-escaping [6], single-owner) to partition data into different storage classes can greatly help the compiler. Past work on performing these analyses for synchronized blocks in the Java language [4] and other explicitly parallel programming models provide a convenient starting point for addressing these problems. Two optimizations that serve as good examples for testing data flow and data dependence analysis in the presence of atomic sections are register allocation (scalar replacement) of heap accesses [9] and instruction scheduling [4]. It is also important to enable automatic *concurrency analysis* [1] and *data race* detection [7] in the presence of atomic sections. Note that atomic sections automatically guarantee absence of *deadlock*, even in the presence of other synchronization constructs [17]. Further, atomic sections are not only useful as a programming abstraction but also as the output of an *automatically parallelizing* compiler that is capable of recognizing *reductions* [20, 15] (such as histogram updates) and other *commutative* operations [16] (such as insertions into a shared data structure). Finally, it is important to keep in mind that data flow and dependence analyses for atomic sections can be further complicated by the presence of exceptions [11].

**Compiler Optimizations and Runtime Mechanisms.** There are multiple runtime mechanisms that can be used to implement atomic sections, all of which can benefit from compiler optimizations. One of them is *transactional memory* [12] — for the purpose of this discussion, transactional memory with explicit retry and rollback (whether implemented in hardware, software, or some combination thereof) is viewed as a system capability that is used for implementing atomic sections rather than as a parallel programming abstraction that is made directly available to the programmer. Another is *automatic lock assignment* by the compiler [21]. Yet another technique is to build on *non-preemptive scheduling* techniques (as in Jikes RVM [2]) for atomic sections that are guaranteed to contain no *yieldpoints*. Finally, in certain cases, an atomic section can be implemented by nonblocking instructions available in the target hardware [13]. Partitioning of data into non-overlapping storage classes is a necessary prerequisite to the use of multiple mechanisms within a single program. Selection of appropriate mechanisms for different atomic sections represents a major compiler challenge, as does finer-grained optimization of individual mechanisms *e.g.*, optimization of acquire/refresh and release/writeback operations [10] within a single transaction. For example, compiler-assigned locks will be a better choice than transactions in the (infrequent) case that an atomic section modifies so much state that it may overflow the available transactional memory buffer, especially when the overhead of locks is low as in modern virtual machines [3]. Another related challenge is optimized integration of the mechanisms selected for atomic sections with thread scheduling policies, such as eager vs. lazy execution of atomic sections.

**Atomic Sections as a replacement for Global Collectives.** The dominant programming model used in scientific applications for high-end parallel machines is a *bulk-synchronous* Single Program Multiple Data (SPMD) model with collective operations such as reductions that piggyback on coarse-grained barrier operations (*e.g.*, as in MPI [19]). This model was well suited to past distributed-memory parallel machines with uniprocessor nodes. However, as the industry moves to *multi-core SMP nodes* and systems with support for a *global shared memory*, it becomes increasingly important to “strength reduce” coarse-grained collective operations into multiple fine-grained invocations of atomic sections. In this way, we can think of atomic sections as “distributed reductions”. The distributed reduction capability is also important for irregular applications, such as applications that work with unstructured and adaptive meshes.

## References

- [1] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna Shyamasundar. May-happen-in-parallel analysis of x10 programs. *Proceedings of Symposium on Principles and Practice of Parallel Programming (PPOPP'97)*, 1997. (To appear).
- [2] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. Flynn Hummel, D. Lieber, V. Litvinov, T. Ngo, M. Mergen, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalepeno virtual machine. *IBM Systems Journal special issue on Java performance*, 39(1), 2000. (see also <http://www.research.ibm.com/jalapeno>).
- [3] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 258–268, June 1998.
- [4] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, and Harini Srinivasan. Dependence analysis for Java. In *12th International Workshop on Languages and Compilers for Parallel Computing*, August 1999.
- [5] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005 Onward! Track*, 2005.
- [6] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1999.
- [7] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, New York, NY, USA, 2002. ACM Press.
- [8] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.
- [9] Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 155–174, London, UK, 2000. Springer-Verlag.
- [10] Guang R. Gao and Vivek Sarkar. Location consistency—a new memory model and cache consistency protocol. *IEEE Trans. Comput.*, 49(8):798–813, 2000.
- [11] Manish Gupta, Jong-Deok Choi, and Michael Hind. Optimizing java programs in the presence of exceptions. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 422–446, London, UK, 2000. Springer-Verlag.
- [12] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.
- [13] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM Press, 1996.
- [14] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):639–662, 2005.
- [15] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.
- [16] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: a new analysis framework for parallelizing compilers. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 54–67, New York, NY, USA, 1996. ACM Press.
- [17] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'05)*, August 2005.
- [18] V. Sarkar and G. R. Gao. Analyzable atomic sections: Integrating fine-grained synchronization and weak consistency models for scalable parallelism. Technical report, CAPSL Technical Memo 52, February 2004.
- [19] Anthony Skjellum, Ewing Lusk, and William Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1999.
- [20] Michael Wolfe. Beyond induction variables. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 162–174, New York, NY, USA, 1992. ACM Press.
- [21] Yuan Zhang, Vugranam C. Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang R. Gao. Optimized lock assignment and allocation for productivity: A method for exploiting concurrency among critical sections. Technical report, CAPSL Technical Memo 65, May 2006.