Journal of Universal Computer Science, vol. 11, no. 5 (2005), 651-660
submitted: 30/11/04, accepted: 31/1/05, appeared: 28/5/05 © J.UCS

Atomicity as a First-Class System Provision

J. Eliot B. Moss
Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
moss@cs.umass.edu

Ravi Rajwar
Intel Corporation
Hillsboro, OR 97124
ravi.rajwar@intel.com

Abstract: We argue that atomicity, i.e., atomic actions with most of the traditional "ACID" properties, namely atomicity, consistency, and isolation but perhaps not durability, should be provided as a fundamental first class resource in computer systems. This implies coherent, convenient, and well-engineered support from the hardware, through the run-time system, programming language, and libraries, to the operating system. We articulate the advantages of this approach, indicate what has already been accomplished, and outline what remains to be done to realize the vision.

Key Words: Atomicity, transactions, transactional memory, cache coherence

Category: C.0 COMPUTER SYSTEMS ORGANIZATION—General—hardware/software interfaces

1 Motivation: Why Provide Atomicity as a First Class Systems Facility?

The Dagstuhl Seminar 04181 [D04] included two discussions concerning offering transactions as a fundamental facility of computer systems. The first discussion focused on hardware, namely the ability to perform a set of operations to arbitrary memory addresses atomically in hardware, while the second discussion focused on software and the advantages of atomicity as a tool for reasoning and development of reliable and robust multithreaded programs.

The motivating developments for both communities emerged from the growing prevalence of highly-threaded software systems and the intractability of their construction and maintenance.

Explicit hardware support for multithreaded software, either in the form of shared-memory chip multiprocessors or hardware multithreaded architectures, is becoming increasingly common. As such support becomes available, application developers are expected to exploit these developments by employing multithreaded programming. Although threads may simplify a program's conceptual design, they substantially increase programming complexity. In writing shared memory multithreaded applications, programmers must ensure that threads interact correctly, and this requires care and expertise. Errors in accessing shared-data objects can cause incorrect program execution and can be extremely subtle. This is expected to become an even greater problem as we go towards heavily threaded systems where their programmability, debuggability, reliability, and performance become major issues.

The emergence of Java [G+00] as an important programming language is particularly relevant, because Java is the first widely used language that includes standardized support for multithreading as part of its ordinary platform.[1] While Java's level of abstraction of memory management is high, i.e., it provides type-safe automatic reclamation of unreachable objects (namely "garbage collection"), its support for multithreading and control of the interaction of threads is based on conventional primitives, which operate at a rather low level of abstraction. The Java memory model [LY99] was intended to articulate Java's behavior in the face of concurrent execution on multiprocessors with advanced memory coherence models (such as acquire-release synchronization [G+90], etc.), but was found to be fatally flawed. The community has developed a replacement [S04], but the overall experience is that even the experts have difficulty reasoning about all but the simplest cases.

While conventional synchronization techniques such as locks, which appear in Java as synchronized methods and blocks, have been extensively used in coordinating interactions among threads, the programmer is often left with the task of discovering and implementing the most efficient synchronization algorithm for the specific circumstance. Keeping both functionality and performance of complex software systems in perspective *simultaneously* is a daunting task, and this trade-off between programmability and performance becomes key. In addition, the software-enforced waiting introduced by locking constructs introduces undesirable behavior in the presence of context switches and thread failures–if a thread holding a lock is delayed, it can prevent the whole system from making progress. (The situation where a low priority thread holds a lock needed by a high-priority one is called *priority inversion*.)

Furthermore, locking and similar synchronization constructs are software *contracts* based purely on *convention*—protected data is to be accessed only after executing the appropriate synchronization construct. These contracts are not *enforced* in anyway: nothing prevents threads from accidentally violating a contract. For example, a programmer may mistakenly update some data structure without executing the right synchronization construct (acquiring the corresponding lock(s)). Likewise, if a collection of locks is designed for individual locks to be acquired in particular orders, so as to avoid deadlock, violating that (unwritten and unenforced) contract can lead to deadlock.

Flanagan and Qadeer [FQ03] have also noted the general problem of improper synchronization in multithreaded Java code, and their tool for checking proper atomicity of Java programs found errors in basic language libraries. Their strategy is to develop type systems and checking tools. We advocate a different approach: include support for

[1] It is true that threads are available with many other languages, but Java has really brought them into main stream programming, and thus introduced a potentially huge problem. Java is also relevant because it has tried to specify semantics of concurrency *in the language*, something not done well in previous widely used languages.

atomicity in the hardware, run-time system, and programming language, so that it is easy to insure atomicity in programs.

1.1 The Challenge of Engineering Multithreaded Software

The challenge in engineering multithreaded software can be summarized in the desire to obtain three properties *simultaneously*:

1. **Reliability:** Data should be accessed only under the appropriate contract, thus guaranteeing intended atomicity of access and update, and avoiding deadlocks. This has to do with *correctness* of the system.
2. **Performance:** The running system should use resources, particularly available processor cycles, effectively. Not only should we avoid priority inversions (which may also be a functionality or correctness issue), we should not unduly delay threads waiting to access data. The desire for performance tends to lead to systems with more, finer-grained, locks and more complex contracts concerning them.
3. **Programmability:** Constructing the system originally, and modifying and maintaining it afterwards, should not be overly expensive. System complexity and subtlety, such as caused by having many locks and complex unenforced contracts, dramatically reduces programmability. Most programmers today do not have the knowledge and skills to build and maintain multithreaded software effectively.

The present state of affairs is that we cannot achieve all three of these properties simultaneously, because a reliable *and* high performing design will tend to be complex. We argue that providing suitable, i.e., well designed and good performing, support for atomicity in the hardware, programming language, and operating system will enable future designs to achieve all three properties, essentially by making it easier to program multithreaded systems to be reliable and high performing.

2 Our Proposed Strategy: Atomic Actions, not Locks

The essence of the strategy we propose is to raise the level of abstraction of support for concurrent (multithreaded) programming. Rather than having the hardware and run-time system offer a mechanistic primitive, namely locks, we envision direct support for *atomic actions*. Since we wish a well-engineered result, achieving all three of the goals of reliability, performance, and programmability, we can offer at the outset only a rough and provisional definition of exactly what constitutes an atomic action; the exact definition needs to emerge from the process of designing language and hardware features that achieve the goals. Here is a provisional definition of "atomic action", which we offer both from the software (programmer's) and hardware viewpoints:

Software View: An atomic action is a (temporally) contiguous sequence of primitive actions in one thread that appears to execute as a whole (or not at all), and *as if* no other thread executes during the action. This is similar to the traditional notion of transaction from database systems [G+75], though we are not necessarily insisting that the effects of an atomic action be durable across system failures of at least some kinds.

Hardware View: An atomic action is a (temporally) contiguous sequence of memory reads and writes in one thread that appears to execute as a whole (or not at all), and *as if* no other thread executes during the action. This is similar to transactional memory [HM93] and related proposals, and ultimately relies on processor and cache rollback mechanisms and cache coherency protocols.

We make two primary claims here. First, that the hardware view can be offered by implementing relatively small enhancements to existing processor designs, caches, and cache coherency protocols. Thus, if there is sufficient motivation from the software side, hardware manufacturers might be persuaded to begin to offer this support. It is a good value added for incremental design effort.

Second, we claim that, given the hardware support, the software (programmer's) view can be achieved through incremental language and run-time system enhancements, and that the whole will support atomic actions with excellent performance and minimal additional cost.

3 The Software Role: *Using* Atomicity

Using an atomicity construct with simple and clear semantics is attractive for reasoning about, and for writing, multithreaded programs because one may assume, and guarantee, stronger invariants. For example, consider a linked list data structure and two operations upon the list: insertion and deletion. Today, *the programmer* would have to ensure that the appropriate lock is acquired by any thread operating upon the linked list. However, an attractive approach would be to declare all operations upon the linked list as "atomic". How the atomicity is provided is abstracted out for the programmer, and the underlying system (hardware, software, or a combination) guarantees the contract of atomicity. The underlying system would also worry about the performance implications of such a guarantee. Decoupling performance from correctness, and abstracting out the machinery for achieving inter-thread coordination, by using an **atomic** construct provides more powerful tools for programmers. For one, they do not have to worry about the details of *providing* atomicity and can focus on ensuring application algorithm correctness.

The essence is that the programmer specifies the *what*, i.e., what must be atomic, and not the *how*, i.e., the implementation details.

4 The Hardware Role: *Providing* Atomicity

The hardware notion of atomicity involves performing a sequence of memory operations atomically. The identification of the sequence is of course best left to the programmer/software layer. However, the provision and guarantee of atomicity comes primarily from the hardware. The core algorithm of atomically performing a sequence of memory operations involves obtaining the ownership of appropriate memory locations in hardware, performing temporary updates to the locations, and then instantaneously releasing these locations and making the updates permanent (meaning, committing them to memory, not necessarily durable to system failures). In the event of action failure, such as induced by some kinds of conflicts, any temporary updates are discarded, thus leaving all critical state consistent. (We can use the same strategy for transient hardware failures as well.) Hardware has become exceedingly proficient in executing operations optimistically and speculatively, performing updates temporarily, and then making them permanent instantaneously as appropriate.

4.1 What Such Hardware Might Look Like

The notion of speculatively updating memory and subsequently committing updates has been developed in the context of speculatively parallelizing sequential programs [K86, S+95], and in the context of explicitly parallel programs [S+93, HM93, RG01, RG02].

Transactional Memory [HM93] and The Oklahoma Update [S+93] were the initial proposals for employing such hardware support for developing lock-free programs where applications did not suffer from the drawbacks of locking, and were generalizations of the load-linked/store-conditional proposals [J+87].[2] The Transactional Memory and Oklahoma Update proposals advocated a new programming model replacing locks. Recently Speculative Lock Elision [RG01] and also Transactional Lock Removal [RG02] have been proposed, where the hardware can dynamically identify and elide synchronization operations, and transparently execute lock-based critical sections as lock-free optimistic transactions while still providing the correct semantics. The hardware identifies, at run time, lock-protected critical sections in the program and executes these sections without acquiring the lock. The hardware mechanism maintains correct semantics of the program in the absence of locks by executing and committing all operations in the now lock-free critical section “atomically”. Any updates performed during the critical section execution are locally buffered in processor caches. They are made visible to other threads instantaneously at the end of the critical section. By not acquiring locks, the hardware can extract inherent parallelism in the program independent of locking granularity.

[2] Of course these were preceded by a number of read-modify-write primitives, such as test-and-set and compare-and-swap [IBM]. Load-linked/store-conditional has more the flavor of atomic actions, however, since it allows an arbitrary computation on the loaded value and separates the read and write parts of the atomic action.

While the mechanism sounds complex, much of the hardware required to implement it is already present in systems today. The ability to recover to an earlier point in an execution and re-execute is used in modern processors and can be performed very quickly. Caches retain local copies of memory blocks for fast access and thus can be used to buffer local updates. Cache coherence protocols allow threads to obtain cache blocks containing data in either shared state for reading or exclusive state for writing. They also have the ability to upgrade the cache block from a shared state to an exclusive state if the thread intends to write the block. The protocol also ensures all shared copies of a block are kept consistent. A write on a block by any processor is broadcast to other processors with cached copies of the block. Similarly, a processor with an exclusive copy of the block responds to any future requests from other processors for the block. The coherence protocols serve as a distributed conflict detection and resolution mechanism and can be viewed as a giant distributed conflict manager. Coherence protocols also provide the ability for processors to retain exclusive ownership of cache blocks for some time until the critical section completes. A deadlock avoidance protocol in hardware prevents various threads from deadlocking while accessing these various cache blocks.

The model just described assumes that a “processor” (a cache, actually) has only one thread of control executing at a time. Many modern processors support multiple simultaneous thread *in hardware*. In such a case, reads and writes buffered in a cache need to be associated with hardware threads, perhaps necessitating a thread id with each cache block. Similar mechanisms may be required to some extent anyway in order to implement atomic read-modify-write operations correctly, etc. The principles are similar, though one needs extra bits and a little extra hardware than without multithreading.

Thus it appears that the stage is well set for hardware to offer with low cost at least a simple high-performance atomic action construct.

4.2 Benefits of the Hardware Atomic Action Construct

The main benefits of this hardware atomic action construct include:

– It eliminates the possibility of deadlock. (It pushes deadlock down to the hardware level, where we can use a priority scheme to guarantee success of at least one of a conflicting set of transactions.)

– It eliminates problems of lock inversion, where a high priority task blocks on a lock held by a de-scheduled low priority task, etc.

– It synchronizes directly on the conflicting data, whereas with locks one must know which locks to acquire in order to manipulate each shared datum.

– It eliminates the overhead of the actual manipulation of the locks. Previous work has shown that in high-traffic, low-conflict situations this overhead is significant.

- It is as fine-grained as need be, boosting performance over the case of coarse-grained locks, and improving reliability compared with designing and using necessarily more complex arrangements of fine-grained locks.
- It allows for more efficient communication of access/update and synchronization information between processors at the hardware level.

In sum, there are good reasons to pursue hardware support for atomicity on grounds not only of improved software engineering but also better performance.

5 What a Software Atomic Action Construct Might Look Like

Recently Harris and Fraser [HF03] proposed a simple yet powerful language construct, offering the semantics of conditional critical regions.[3] Its form (in Java) is as follows:

$$\textsf{atomic}\ (p)\ \{\ S\ \}$$

where p is an optional predicate (expression of type boolean) and S is a statement. It meaning is: execute S if p evaluates to true; the evaluation of both p and S is done as a single atomic unit with respect to other threads. If p evaluates to false, the executing thread needs to wait for some other thread to change some variable on which p depends (a fact which we can detect in hardware by noticing a conflicting access; we would then abort the evaluation of p that led to false and retry the whole construct).

This construct is slightly more sophisticated and flexible than a more basic one with the condition p, which we might write:

$$\textsf{atomic}\ \{\ S\ \}$$

We might be able to express the Harris and Fraser construct in terms of the more basic one if we are also given a primitive such as abort or retry, though mapping onto hardware detection of changes that might affect the value of p would be more difficult.

6 Open Questions

Crucial work remains–both in hardware and software systems. The classic chicken-and-egg problem persists. On the one hand, existing software-only implementations of atomicity and transactions for general use suffer from poor performance; on the other hand, no hardware systems today provide the notion of generalized atomic transactions. A major hurdle for hardware transactions remains in their specification. Importantly, what hardware transaction abstraction should be provided to the software? How is the limitation of finite hardware resources for temporarily buffering transactions handled? One strategy is to attempt to provide transactions of essentially unbounded size (though

[3] See also Lomet's earlier work [L77].

large transactions might have large performance impacts). This would be nicest from the standpoint of software design, but still ultimately demands some way to structure software to provide good performance. Multi-level (open nested) transactions seem a fruitful direction. A tension will always exist between "power" users, who would prefer all possible flexibility available from the hardware, and users who would prefer a simple hardware abstraction where they do not worry about underlying implementations. These are some of the questions that must be addressed. However, much of the core mechanisms in hardware required for atomic transactions are well understood and have been proposed for numerous other reasons.

Likewise, the software area requires significant work. Harris and Fraser's construct may provide a good starting point, but one immediate issue is how to specify the semantics with enough adequate rigor (but in a way that programmers and language implementors can use effectively). At least from the formal methods community perspective, specifying a concise formal description of the above constructs as a semantic inference rule in the operational semantics style affords one approach—though the semantics would benefit consideration from a variety of formal perspectives.

A first pass at an operational semantics definition is:

$$s[p(s', \mathit{true}) \wedge s'[S]s'' \vdash s[\textbf{atomic}\ (p)\ \{\ S\ \}]s''$$

In English: If we start in state s and the guard predicate p evaluates to true, then we make the atomic state transition that evaluates p followed by S. No other other process will be able to observe or affect the intermediate state s' or any other intermediate state. Note that no transition occurs if p evaluates to false

Looking forward, we suggest language designs will need to go beyond such simple constructs. Some of the issues designs might want to handle include:

– Connecting with durability somehow, perhaps through providing special durable memory regions.

– Expressing relative ordering constraints (or lack thereof) for transactions issued conceptually concurrently (e.g., iterations of counted loops, as typical of scientific programs operating on numerical arrays).

– Supporting (closed) nesting (in the style of Moss's nested transactions [M85]) and the bounded rollback that it implies on failure.

– Supporting *open* nesting [WS92] (also called *multi-level transactions*), where commitment of a nested transaction releases basic resources (e.g., cache lines) but implies retention of semantic locks and building a list of undo routines to invoke if the higher level transaction fails. This process of undoing a committed nested open transaction is often called *compensation*.

– Providing for lists of actions to perform only if the top-level enclosing transaction commits.

- Supporting the leap-frogging style of locking along a path one is accessing in a data structure. (In a singly-linked list, one holds locks on elements i and $i + 1$. To move down the list, one acquires a lock on $i + 2$ and releases the lock on i.) Essentially this requires means to release specific previously acquired items.
- More general split-join transaction semantics.
- More general save-points for rolling back.

Some of these features may require more functionality from the hardware. For example, it would appear that leap-frog style locking requires managing more than one set of accessed cache lines, so the hardware may need to include the notion of a *transaction id* and the ability to commit/abort a transaction, given its id, independently of other ongoing transactions, etc.

7 Atomicity as a First Class System Provision

The important challenges of language and software systems support for such transactions and their interactions with the underlying atomic hardware transactions requires a coordinated inter-disciplinary research effort. We believe future software systems should use transactions for improving their reliability and programmability, and hardware mechanisms such as atomic transactions should provide the common-case performance for such software systems. The uncommon case might be handled using a slower software interface, thus guaranteeing transaction properties in all cases. This decoupling of performance and programmability holds the key to future reliable high-performance systems [RG03].

References

[D04] Dagstuhl Seminar 04181. Atomicity in System Design and Execution. organized by C. Jones, D. Lomet, A. Romanovsky, G. Weikum, http://www.dagstuhl.de/04181/

[FQ03] Cormac Flanagan and Shaz Qadeer. A Type and Effect System for Atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003),* pages 338–349, San Diego, CA, May 2003. ACM SIGPLAN, ACM Press.

[G+90] K. Gharachorloo, et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[G+00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java language specification, second edition. Addison-Wesley, 2000.

[G+75] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of Locks in a Large Shared Data Base. In *Proceedings of the International Conference on Very Large Data Bases*, pages 428–452, 1975.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 2003 ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003),* Anaheim, CA, October 2003. ACM SIGPLAN, ACM Press.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, May 1993.

[IBM] IBM Corporation. IBM System/370 Principles of Operation. IBM Systems library, order number GA22-7000.

[J+87] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.

[K86] Thomas F. Knight. An architecture for mostly functional languages. In *Proceedings of ACM Lisp and Functional Programming Conference*, pages 105–112, August 1986. *An initial proposal for speculatively parallelizing sequential code using hardware support.*

[LY99] Tim Lindholm and Frank Yellin. The Java virtual machine specification, second edition. Addison-Wesley, 1999.

[L77] David B. Lomet. Process Structuring, Synchronization, and Recovery Using Atomic Actions. In *Proceedings of an ACM Conference on Language Design for Reliable Software (LDRS)*, pages 128–137, 1977.

[M85] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, 1985.

[RG01] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, pages 294–305, Austin, TX, December 2001. IEEE.

[RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 5–17, San Jose, CA, May 2002. ACM, ACM Press.

[RG03] Ravi Rajwar and James R. Goodman. Transactional Execution: Toward Reliable, High-Performance Multithreading. In *IEEE Micro*, pages 117–125, Volume 23 Number 6 November/December 2003.

[S+95] Gurindar S. Sohi, Scott E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 414–425, 1995.

[S+93] Janice M. Stone, Harold S. Stone, Phil Heidelberger, and John Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, November 1993.

[S04] Sun Microsystems. JSR-000133 Java memory model and thread specification revision. Available from http://www.cs.umd.edu/˜pugh/java/memoryModel, February 2004.

[WS92] Gerhard Weikum and Hans-Jörg Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. Database Transaction Models for Advanced Applications, (A. Elmagarmid, ed.), Morgan Kaufmann, 1992, pages 515–553.