

Simple atomic blocks

Vijay Saraswat, IBM TJ Watson Research Center

March 3, 2007

1 Language Design

Simple atomic blocks are based on the following design philosophy:

- New language design should attempt to increase productivity of concurrent programmers through higher level abstractions, without sacrificing performance.
- Common concurrency idioms (producer/consumer, dataflow, master/slave, barrier-style communication) should be expressible in new language designs in a way that the compiler can statically reason about them. (cf the design of *clocks* in X10 [5], `clocked final` types for determinate parallelism etc).
- Atomic blocks are a powerful hammer – they should be used only when other patterns are not adequate for the task at hand, e.g. when synchronization is time-dependent (work-queue management) or data-dependent (transferring money between bank accounts, B-tree updates). (Though, other constructs may be thought of as being built *semantically* on atomic blocks.)
- Atomicity and Ordering are orthogonal concepts and should be reflected with orthogonal language features.
- Atomic Blocks should express atomicity and isolation – the packaging of multiple read/writes into a single bundle communicated to the memory subsystem – without forcing the programmer to work with a particular solution (locks).
- Separate control constructs (e.g. `finish`) should be used for ordering.
- Atomic Blocks should be kept separate from heavier transactional notions (e.g. aborts, roll-backs).
- Separate efficiency from correctness. There should be a simple correct (though possibly inefficient) implementation scheme for atomic blocks. But the performance-focused programmer should be able to add extra information (via *annotations* [4]) which may improve performance, without compromising correctness.
- Aliasing and concurrency do not mix well. Design language mechanisms to permit the programmer to express locality/aliasing information (via types and annotations).

Simple Atomic Blocks are an extremely simple concurrent construct for atomicity, based on the above ideas, and implemented in X10 [5, 1]. A block `atomic S` is executed in a given store in one step, isolated from its environment. The resulting store reflects the write operations performed by `S` – even if `S` terminated abruptly. Each block is decorated with a set of *places*: the compiler guarantees that the block accesses only data items located in these places. `S` may recursively contain atomic blocks; their execution is considered part of the single step of the execution of the outer atomic block (closed nesting). I/O operations are *not* permitted inside atomic blocks.

Conditional atomic blocks are supported (`when(c) S`). However, the condition on which execution may suspend must be specified at the beginning of the block. Only top-level blocking is permitted (`when` may not occur nested inside an `atomic`). A conditional atomic block that is waiting for its condition to be satisfied is considered to be *blocking* in the programming model. If all activities are executing a conditional atomic block, and none of the conditions are satisfied, the computation is deadlocked. The implementation must report such a condition and cease any activity. That is, it is not permissible for the implementation to convert deadlock into livelock.

Fairness An implementation should deadlock only if the programming model says that it should dead-

lock. That is, any implementation level notions (e.g. locks) should not introduce a deadlock into a computation in which according to the programming model semantics there is no deadlock. (For instance if the implementation acquires locks to implement atomics, it must ensure that locks are acquired in a specific order so that no spurious deadlocks are introduced.)

Beyond this there is no guarantee of fairness. An activity wishing to execute an atomic block may be prevented from doing so indefinitely if the available computational resources are being utilized to permit other activities to progress.

Atomicity An implementation may choose to bring in reads/writes from before or after an atomic block into the atomic block, and to combine atomic blocks. Atomic blocks may not be split [6].

Ordering. We are investigating a weak memory model for atomic blocks. In such a model it is not the case that there must be one global total order of execution for all atomic blocks. Instead, we expect to use a CCCC model [3].

Simple atomic blocks vs Transactional Memory Simple atomic blocks are designed to focus the application programmer’s attention on the application’s needs for *atomicity* and *isolation* rather than whether this need should be realized using locks, and if so, which kinds of locks and how many.

Atomic blocks share the goal of high-level specification of atomicity with transactional memory. They differ in that they are focused on an efficient solution for a limited portion of the design space. There is no notion of rollback or optimistic execution or explicit abort. Unrestricted compositionality is *not* a goal for simple atomic blocks. An implementation may optimize for “small” atomic blocks. Atomic blocks are intended to be used fairly infrequently. Other schemes, such as producer consumer schemes, or clocked final schemes, or master/slave (finish / at each) schemes are useful in most contexts to divide work in parallel in such a way that there is no conflicting access to shared data. Atomic blocks are to be used in cases where these schemes do not work, e.g. in unstructured concurrent access to a shared data structure (e.g. a work queue) or on data-dependent mutual exclusion (e.g. transferring money from one bank account to another).

However, high performance programmers may add extra annotations to atomic blocks to guide the im-

plementation towards using a more efficient scheme to implement atomics (see below).

2 Implementation

In the current single-node multiprocessor implementation of X10, atomic blocks are implemented by associating a lock per place, and acquiring the lock. Thus a place defines the granularity of locking. (It is the programmers’ responsibility to partition the program’s data into places.)

We discuss some speculative ideas for a more refined implementation of simple atomic blocks.

Atomic blocks in the source program may be translated into two kinds of atomic blocks – *immediate* blocks and *locked* blocks. An immediate block can be executed immediately using hardware read/modify/write operations, e.g. CAS.

2.1 Locked Blocks

The semantics of a program may be understood as a partially ordered multiset of steps or blocks, with some steps marked as `atomic` [6].

After Lamport [2] say that two blocks are ordered if one must complete before the other. For instance, in `finish A; B`, A is ordered before B . We write $A\#B$ for the relation “ A and B are unordered.”

Two blocks A and B are said to *conflict* on a place p if $A\#B$ and $p \in places(A) \cap places(B)$. A place p is said to be *hot* for A if there is some block B such that A and B are in conflict on p . The set of places hot for A is $hot(A)$.

The basic idea behind the implementation of an atomic block is to define a partial mapping which associates each place p with at most one lock $X(p)$.¹ Places are considered to be totally ordered. An atomic block A is implemented by acquiring $X(p)$ for each place $p \in places(A)$ (in the order specified by places), executing the body of the block and releasing the locks.

Idea 1: Acquire $X(p)$ when executing A only if $p \in hot(A)$.

This requires examining the ordering relation at compile-time/runtime. In fact it is possible that this could result in no lock being acquired at runtime.

Even with this idea, a block may acquire many locks at runtime. It is desirable to minimize the

¹Below, for a set of places Q , we shall use the notation $X(Q)$ to signify the set $\{X(q) \mid q \in Q\}$.

number of locks acquired. Acquiring fewer locks may mean less concurrency (e.g. acquiring a single global lock common to all places will work but will sharply reduce concurrency), but better serial execution.

It seems reasonable, therefore, to consider constraints on X of the form $X(p) = X(q)$ or $X(p) \neq X(q)$, i.e. constraints that force multiple places to share (or not share) locks. Note that *any* set of (satisfiable) constraints guarantees *correctness*; but different sets may affect performance. Therefore we will permit the programmer to manually add assertions $X(p) = X(q)$ and $X(p) \neq X(q)$ as a way of managing performance.

Therefore the design problem for the implementation of X10 atomics is to devise ways in which these constraints can be generated and evaluated.

Idea 2: Assign the same lock to two places p and q if there is some block for which they are both hot. i.e. $p, q \in \text{hot}(B)$ implies $X(p) = X(q)$.

This ensures that each block requires the acquisition of at most one lock at runtime. However, this may force *false sharing*. Two blocks A and B are said to be falsely sharing if $A \# B$, $\text{places}(A)$ and $\text{places}(B)$ are disjoint, but $X(\text{places}(A))$ and $X(\text{places}(B))$ are not. For instance, consider unordered blocks A , B and C with $\text{places}(A) = \{p, q\}$, $\text{places}(B) = \{p\}$ and $\text{places}(C) = \{q\}$, and $X(p) = X(q)$. Now there is false sharing between B and C . They are forced to contend for the same lock even though they work on disjoint places.

Idea 3: Assign the same lock to two places only if it does not induce false sharing: $X(p) = X(q)$ implies for all A, B : ($A \# B$ and $\text{places}(A)$ and $\text{places}(B)$ are disjoint) implies $X(\text{places}(A))$ and $X(\text{places}(B))$ are disjoint.

In the above example, this would force $X(p)$ and $X(q)$ to be distinct.

We are considering several such schemes and evaluating them for implementation.

A drawback of the simple mapping approach described above is that the equality relation is transitive. Therefore if a block A forces $X(p) = X(q)$ and block B forces $X(q) = X(r)$, block C will see $X(p) = X(r)$, thereby inducing false sharing.

A richer setting is one in which locks are associated with *subsets* of places. Sets of place are ordered by inclusion. This induces an order on the set of locks: $k < l$ if $\text{places}(k)$ is contained in $\text{places}(l)$. We are developing a framework for *partially ordered locks* which

ensures that while a lock l is being held, no lock $k < l$ can be held.

Acknowledgements. The ideas discussed here are being developed in collaboration with Radha Jagadeesan, Christoph von Praun, Maged Michael and Nate Nystrom.

References

- [1] Philippe Charles, Christopher Donawa, Christian Grothoff, Kemal Ebcioglu, Allen Kielstra, Vijay Saraswat, Vivek Sarkar, and Christoph von Praun. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.
- [2] Leslie Lamport. A New Approach to Proving the Correctness of Multiprocess Programs. *ACM TOPLAS*, 1(1):84–97, July 1979.
- [3] Doug Lea and Vijay Saraswat. Cache coherent causal consistency. 2007.
- [4] N. Nystrom and V. Saraswat. An annotation and compiler plugin system for x10. Technical report, IBM TJ Watson Research Center, 2007.
- [5] Vijay Saraswat and Radha Jagadeesan. Concurrent Clustered Programming. In *Concur*, pages 353–367, 2005.
- [6] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *To appear in Proceedings of ACM symposium on Principles and Practice of Parallel Programming*, 2007.