Simplified synchronization through optimistic linearizability

Maurice Herlihy
Brown University

Eric Koskinen
University of Cambridge

Abstract

We explore a novel programming model for multicore architectures in which shared data-structures are replicated per-thread. With a suitable mechanism for broadcasting data-structure operations (which may be wait-free or lock-free) there is no longer a need for elaborate, datastructure-specific synchronization (e.g. as seen in java.util.concurrent) at each replica. Moreover, threads can lazily apply the broadcast operations on an as-needed basis.

In this position paper, we provide a formal semantics for this programming model but have not yet explored the practical viability of the approach. Nonetheless, previous work seems to indicate that replication, at least in the context of operating system internals, can yield performance improvement [1].

1. Introduction

The advent of multicore has renewed interest in the design and implementation of concurrent data structures [3]. Concurrently shared data structures present several challenges. To provide adequate concurrency, there is often a need for fine-grained synchronization in the form of multiple locks or atomic compare-and-swap operations. Because cache locality is critical for performance, it is often necessary to split data structures into components aligned and organized to fit into caches in efficient ways.

Although these challenges are often met in ingenious ways, there are inherent limitations to this approach. Such designs are necessarily type-specific. For example, synchronization structures and data layouts customized for skip lists are inappropriate for balanced trees, and vice-versa. A design optimized for one architecture may perform poorly on another, or even on future versions of the same architecture.

An alternative approach is to treat multicores more like distributed systems [1], where each core's memory is private, but shared state is replicated, and replicas are kept up-to-date by a generic distributed transaction manager.

We consider a simple transaction manager that manages replica consistency for *linearizable* objects, where each individual operation is an atomic transaction. When a thread applies an operation to an object, the transaction manager notifies each thread of the operation, and operations are delivered to each thread in the same order. Because we do not support multi-operation transactions, transactions never abort, and indeed they inherit the non-blocking properties of the underlying transaction manager.

Because the underlying transaction manager is generic, notifying each core when to install each update, there is little or no need for type-specific customization. There is no need for synchronization because each core modifies its own local replica when notified by the underlying transaction manager. There is reduced need for customized data layout because cores are not competing with one another to cache the same data. Internally, the transaction manager itself is free to exploit platform-specific shared memory and cache structures, hiding these details from its users. Such a design promises to be portable across architectures and across generations.

This design, however attractive, is still unproven, and we will not try to further justify it here. Instead, we restrict ourselves to providing a formal semantics for such objects.

Related Work. Our arguments why cores should be treated more like distributed systems are adapted from Baumann *et al.* [1]. Our formal approach is based on our earlier work on coarse-grained transactions [4].

Protocols such as Virtual Synchrony [2] and Paxos [5] also rely on a form of ordered message delivery to maintain replicated state. These protocols, however, are based on message-passing, they are substantially more complex than we need, and they provide different semantic guarantees.

Limitations. In this short position paper, we do not address the practical viability of our programming model. However, as seen previously, replication can be more effective than sharing objects in the context of operating system internals [1]. The same may be true of more general programming models.

Clearly, there are some cases for which our proposed programming model is not well-suited. When the operations themselves require significant computation (e.g. compression, encryption, etc.) then the cost of performing this computation *per-replica* may not be offset by the elimination of synchronization.

2. A model of replicated state

The model consists of replicated data-structures (or "objects") from domain $\mathcal{O}$. Threads each manipulate via methods, from domain $\mathcal{M}$, on their own local store of objects, though these replicas are treated logically as shared state.

A thread $\tau \in T$ is a tuple $\langle s, \sigma_\tau, \mathcal{A}_\tau, \ell \rangle$, which consists of program code s, the local store σ_τ, a function $\mathcal{A}_\tau : O \to \mathcal{M}$ list, mapping objects to the list of methods which have been performed, and a list of operations ℓ.

The program code of each thread is given by the following statement grammar

$$s ::= c \ ; \ s \mid o.m \ ; \ s \mid \texttt{skip}$$

A statements s can be a standard command c (such as assignment or branching) or else data-structure operations (represented as method calls $m \in \mathcal{M}$ on shared objects $o \in \mathcal{O}$). We denote each replicated store as σ_τ, and they consist of local variables as well as the replicated objects, each denoted o. The initial stores σ_τ^0 are defined as:

$$\sigma_\tau^0 \triangleq \{\forall o \in \mathcal{O}. o \mapsto \mathsf{init}(o)\}$$

where init returns the initial state of the given object. The final thread component ℓ is a list of operations which have yet to be performed on the local replica of the object currently being manipulated As we will see, this final element is typically nil but populated whenever a method is about to be invoked.

$$\frac{}{\langle \texttt{skip}; s, \sigma_\tau, \mathcal{A}_\tau, \mathsf{nil}\rangle, T, \mathcal{A} \;\rightsquigarrow\; \langle s, \sigma_\tau, \mathcal{A}_\tau, \mathsf{nil}\rangle, T, \mathcal{A}}\; SKIP$$

$$\frac{\sigma'_\tau \in [\![c]\!]\sigma_\tau}{\langle c; s, \sigma_\tau, \mathcal{A}_\tau, \mathsf{nil}\rangle, T, \mathcal{A} \;\rightsquigarrow\; \langle s, \sigma'_\tau, \mathcal{A}_\tau, \mathsf{nil}\rangle, T, \mathcal{A}}\; CMD$$

$$\frac{}{\langle o.m; s, \sigma_\tau, \mathcal{A}_\tau, \mathsf{nil}\rangle, T, \mathcal{A} \;\rightsquigarrow\; \langle o.m; s, \sigma_\tau, \mathcal{A}_\tau, \mathsf{diff}(\mathcal{A}(o), \mathcal{A}_\tau(o))@[o.m]\rangle, T, \mathcal{A}[o \mapsto \mathcal{A}(o) :: m]}\; OI$$

$$\frac{\sigma'_\tau \in \mathsf{apply}(\ell, \sigma_\tau)}{\langle o.m; s, \sigma_\tau, \mathcal{A}_\tau, \ell\rangle, T, \mathcal{A} \;\rightsquigarrow\; \langle s, \sigma'_\tau, \mathcal{A}_\tau[o \mapsto \mathcal{A}_\tau(o)@\ell], \mathsf{nil}\rangle, T, \mathcal{A}}\; OC$$

$$\begin{array}{lcl}
\mathsf{diff}(m :: \ell_1, m :: \ell_2) & = & \mathsf{diff}(\ell_1, \ell_2) \\
\mathsf{diff}(\mathsf{nil}, m :: \ell_2) & = & [m] :: \mathsf{diff}(\mathsf{nil}, \ell_2) \\
\mathsf{diff}(\mathsf{nil}, \mathsf{nil}) & = & [] \\
\mathsf{apply}(m :: \ell, \sigma) & = & \mathsf{apply}(\ell, [\![m]\!]\sigma) \\
\mathsf{apply}(\mathsf{nil}, \sigma) & = & \sigma
\end{array}$$

Figure 1: Programming model semantics

Semantics A configuration $\mathcal{C} \in T \times \mathcal{A}$ consists of the running threads, and a (global, authoritative) mapping $\mathcal{A}$ from objects to lists of operations. For simplicity we assume a parallel composition rather than an explicit treatment of `fork`.

The semantics is given in Figure 1. A single thread is nondeterministically chosen from the set of threads and takes a step. The $SKIP$ rule consists of discarding the `skip` and preserving all other components of the configuration. In the CMD rule, the command c is applied to the relevant thread's local state.

The OI and OC rules correspond to the invocation and completion of an object method $o.m$. The OI rule atomically copies the list of method operations which have been performed on o, calculates which operations are new (via diff) and inserts the list in the final component along with m. Moreover, in this rule the new method m is enqueued in the global mapping $\mathcal{A}$ (this is the linearization point). The OC rule atomically applies all of the ℓ operations to the local replica of o (via apply) which includes the new operation m. Note that while our semantics tracks replica currency in terms of lists of methods, in practice this can simply be implemented as local pointers into the global list $\mathcal{A}$.

Remark: Operations must be deterministic. Since operations are replayed at each replica, their outcomes must agree (see Figure 1). The operations may have nondeterministic *specifications*, so long as the replicas interpret them deterministically.

Remark: Operations must be total. Because operations are applied sequentially, they must be *total:* any operation applied in any state must return a value instead of blocking. For example, a *dequeue* operation that finds an empty queue must throw an exception instead of blocking. We believe this requirement can be dropped, but we do not do so here.

Theorem 2.1. *(Linearizability) Any replicated data-structure o is linearizable.*

Proof: (Sketch) The linearization point is in the atomic rule OI where the new intended operation is enqueued to $\mathcal{A}$. When any other thread accesses the object in OC, it will construct an equivalent copy by applying all outstanding operations (all of which must complete because operations are deterministic and total).

3. Conclusion

We have outlined a formal model for optimistic linearizability, a novel programming model for multicore architectures. This formal model is only a first step: much remains to be done to establish the viability of the approach.

References

[1] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 29–44.

[2] BIRMAN, K. P. The process group approach to reliable distributed computing. *Commun. ACM 36*, 12 (1993), 37–53.

[3] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[4] KOSKINEN, E., PARKINSON, M., AND HERLIHY, M. Coarse-grained transactions. *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010) 45*, 1 (2010), 19–30.

[5] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998), 133–169.