Actors and Hot Objects [*]

Locks and Lock-Free Programming in the Encore Programming Language

Tobias Wrigstad
Uppsala University

Thorbiörn Fritzon
Spotify

Abstract

We present Hot Objects, actors which logically support processing of multiple messages in parallel, in the Encore programming language. The internal behaviour of Hot objects is encapsulated inside an asynchronous interface. We discuss compositionality of Hot Objects and the reintroduction of locks into actor-based systems.

1. Introduction

Actor-based systems have been described as a natural way of adding concurrency to object-oriented systems. By encapsulating a single thread of control, actor systems maintain simplicity and facilitate reasoning about objects' states. By relying on asynchronous messaging across actor boundaries, object composition is greatly simplified, and deadlocks occur at higher levels of abstraction than in systems based on threads and locks, where circular dependencies of critical sections are hidden by object encapsulation.

Actor-based systems rely on saturation of a system for efficient utilisation: unless there are fewer actors than cores, an actor-based system can theoretically keep a machine busy.

Load-balancing in actor-based systems can be implemented using work-stealing, where a core that runs out of work can migrate one or more actors from other cores' run-queues to avoid idling. Again, this requires that there are busy enough actors in the system, or there will be none to steal.

The popularity of actors in a program is key to making load-balancing possible, and imbalance in popularity may destroy a system's scalability. By popularity, we mean the amount of messages an actor receives in relation to others. An actor that receives relatively few messages is *unpopular*, whereas an actor that receives relatively many is *popular*. If popular actors are on the critical path of less popular actors, the popular actors should be given higher priority in the system to avoid starvation. (For simplicity, we forego the discussion of different complexities for different messages.)

2. Example: Typical Service

Consider the following abstract description of a simple server system S for a large number of clients. Each request in S is tied to a particular user, and each user has a number of associated tokens. These tokens may correspond to privileges, resources, offers etc. Each token can be in two states—available or unavailable. An available token may correspond to a resource which a user may claim, and an unavailable token may correspond to a resource which the user has already "used up." (The example is abstracted from real-world systems used at Spotify.)

A naive actor model of this system uses one actor per client $C_1 \ldots C_n$, one actor A that maps user ids to *available* tokens, and one actor U that maps user ids to *unavailable* tokens. A and U perform some look-up operation—for simplicity imagine that all data is stored in-memory in hash tables—and sends a reply to the client (or fulfils some future value).

Since there is one connection to A and one connection to U for each of the n clients, and one connection from A and B to each client respectively, A and U are vastly more popular than each individual client. Even maximally increasing the priority of A and B, assigning them to a private core each which deals exclusively with "it's actor", a sufficiently big machine will still not be able to deal with requests to A and B fast enough as the remaining cores allow too many clients making concurrent requests.

2.1 Solution 1: multiple copies of A and B

One solution to even out the popularity scores is to multiply A and B. This has several positive consequences:

– Each actor must deal with a much lower number of requests
– Multiple requests of the same kind can be handled in parallel

In practise, this solution will not work if the hash-tables (data bases) in A and B are mutable. To preserve the single thread of control abstraction, each actor must have a copy of the *entire hash table*—and changes will cause different maps to become outdated. One possibility is to divide the data across all A actors (and similar for B), but this requires an externally known load-balancing scheme (*e.g.,* mapping certain hash-ranges to certain actors), and scaling now relies on an evenly distributed load.

2.2 Solution 2: Break Single Thread of Control

Another solution to this problem is to break the single thread of control for actors and let actors process multiple messages simultaneously. This has the added bonus of being able to encapsulate any load-balancing internally. This solution is similar to a thread-based model, except that each actor has the opportunity to *pull* several messages at the same time, and carry them out—either as several concurrent activities inside each actor, or by exploiting synergy between requests, or a combination. The key difference lies in *pulling* as opposed to being operated on, as in the case for threads.

This is the solution we take in the Encore programming language [2], which we are currently developing. Encore supports a notion of *Hot Objects*, which are actors that optionally support evaluating messages in parallel. The rest of this extended abstract deals exclusively with Hot Objects.

3. Hot Objects

The Encore language guarantees data-race freedom, which is defined as the absence of read–write races on locations without any intermediate synchronisation. Apart from reasons to avoid data-races for correctness, their absence is mandated by the ORCA

[*] Partly funded by the EU project FP7-612985 UPSCALE: From Inherent Concurrency to Massive Parallelism through Type-based Optimisations.

garbage collection protocol [5] which Encore uses under the hood. This allows Encore to issue field updates without write barriers, and empowers each actor to garbage collect at its own leisure, without waiting for anyone else in the system.

3.1 Lower Latency Through Synchronous Communication

The Encore Hot Object design decouples the implementation of a Hot Object from its interface. While the interface of a hot object is an asynchronous actor interface, communication may happen synchronously under the hood. For short requests, avoiding message passing overhead (allocating space for arguments, contention on actor mailbox, etc.) may significantly reduce latency, especially if the cross-call from the actor to the Hot Object is performed immediately rather than the caller suspending itself, waiting for a call-back. In the spirit of queue-delegation locking [7], a possible implementation can test for contention (*e.g.,* success or fail taking a lock) and only turn the call into an asynchronous one if the expected waiting time is too long (*e.g.,* more than two other actors ahead in the queue). This is similar to work by Kogan and Herlihy [8] where synchronous calls are lifted to calls returning futures.

3.2 Lower Latency Through Full Concurrency

On the flip side of keeping the single thread of control but improving the speed of message sends by making them synchronous is using Encore's lock-free type system [3] that supports implementation of lock-free data structures with guarantees of data race-freedom. Implementing lock-free Hot Objects allows all calls to be synchronous with a guarantee that all processing of messages inside an actor is written in such a way that conflicts are handled internally. Lock-free data structures are inherently scalable, but are difficult to program. While Encore's type system will help a programmer avoid easy mistakes such as duplication of ownership, it will not solve coming up with a lock-free protocol.

It is entirely possible to implement Hot Objects using Software Transactional Memory, which is likely much simpler, but less performant than lock-free data structures based on CAS.

3.3 Re-Introduction of Locks

In the middle of the above extremes sit the re-introduction of locks in actor-based programs. While the encapsulation of locking behaviour destroys compositionality of objects in traditional object-oriented programming, encapsulation of locks inside actor boundaries work to preserve compositionality of Hot Objects: as long as locks only govern access to objects *inside* the Hot Object, the only methods which are affected by the re-introduction of locks are the methods of the Hot Object itself. Thus, deadlocks can only happen due to a faulty implementation of a set of methods belonging to the same object, whose composition is known by design.

Due to Encore's type system, which is based on the Kappa system of reference capabilities [4], statically determining what data inside an operation belongs to the current actor is trivial, and declaring locks on unencapsulated objects (which could for example be shared with another actor) is not allowed.

Encore locks support nesting similar to the lock-ordering in SafeJava [1]. This allows *e.g.,* protecting access to an entire hash-table with a readers–writer lock, which can subsequently be acquired by many, and thus allows several threads of control to navigate through this data structure to places where a second, nested, lock can be grabbed, *e.g.,* a write-lock to update a single bucket in a hash-table, or several threads sharing a bucket through a read-lock for looking up available tokens for some user id.

4. Related Work, Discussion and Wrapping Up

Hot Objects in Encore are a form of actors useful for implementing designs where a few actors are vastly more popular than others, or where low latency is important, but not possible or feasible to achieve by replicating data across lots of actors. Actors that process several messages at a time is not new. For example, Henrio *et al.* [6] define a notion of multi-threaded active objects, where unchecked programmer assertions are used to determine if messages can run in parallel. This is not expressive enough for Encore, since Encore relies on guaranteed absence of data-races. In contrast, Östlund's Joelle [9] supports parallel message processing checked for non-interference through an ownership-based effect system. This system however cannot express the example above, as it cannot be expressed as static non-interference.

While technically, Hot Objects implemented using locks reintroduce all the pitfalls of thread-based programming in the actor model, there is a fundamental conceptual difference: locks are added inside a single capsule, which cannot be composed with the surrounding system in a way that introduces new dependencies between operations. It is of course possible to implement a system inside a single parallel actor, but even then—the entry point of each "thread" is known and inside just a single name space.

Hot Objects' interfaces are identical to actors—they support asynchronous communication which return future values. The underlying implementation can be changed without changing the interface: synchronous serial communication, concurrent data structures, software transactional memory, or locks. This allows late-optimising the implementation of a system's performance critical places, without breaking compositionality.

In Encore, Hot Objects require different garbage collection than the rest of the system. For example, a lock-free implementation of a Hot Object will impose write barriers for fields, and a more costly collection of objects because of the possible contention. By isolating the places where multiple threads of control may manipulate the same objects, we can keep the original design of the ORCA garbage collector, and only pay overhead for contention in the places where it is needed.

References

[1] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. Ph.D., MIT, Feb. 2004.

[2] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I. Pun, S. L. T. Tarifa, T. Wrigstad, and A. M. Yang. *Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore*, pages 1–56. Springer International Publishing, Cham, 2015.

[3] E. Castegren and T. Wrigstad. LOLCAT: Relaxed linear references for lock-free programming. Technical Report 2016-013, Department of Information Technology, Uppsala University, July 2016.

[4] E. Castegren and T. Wrigstad. Reference Capabilities for Concurrency Control. In *ECOOP 2016*, volume 56 of *LIPIcs*, pages 5:1–5:26, Dagstuhl, Germany, 2016.

[5] S. Clebsch, S. Blessing, J. Franco, and S. Drossopoulou. Implementation, compilation, optimization of object-oriented languages, programs and systems workshop. In *Ownership and Reference Counting based Garbage Collection in the Actor World*. ACM, 2015.

[6] L. Henrio, F. Huet, and Z. István. *Multi-threaded Active Objects*, pages 90–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[7] D. Klaftenegger, K. Sagonas, and K. Winblad. Delegation locking libraries for improved performance of multithreaded programs. In *European Conference on Parallel Processing*, pages 572–583. Springer, 2014.

[8] A. Kogan and M. Herlihy. The future(s) of shared data structures. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 30–39, New York, NY, USA, 2014. ACM.

[9] J. Östlund. *Language Constructs for Safe Parallel Programming on Multi-Cores*. PhD thesis, Uppsala UniversityUppsala University, Computing Science, Division of Computing Science, 2016.