Efficient and Reasonable Object-Oriented Concurrency

Scott West   Sebastian Nanz   Bertrand Meyer*

Department of Computer Science
ETH Zürich, Switzerland
firstname.lastname@inf.ethz.ch

*Also Politecnico di Milano, Italy, and Innopolis University, Kazan, Russia

Abstract

Making threaded programs safe and easy to reason about is one of the chief difficulties in modern programming. This work provides an efficient execution model and implementation for SCOOP, a concurrency approach that provides not only data-race freedom but also pre/postcondition reasoning guarantees between threads. The extensions we propose influence the underlying semantics to increase the amount of concurrent execution that is possible, exclude certain classes of deadlocks, and enable greater performance.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors—Code generation, Optimization, Run-time environments

Keywords Concurrency, object-oriented, performance, optimization

1. Introduction

There is an increasing amount of attention on developing concurrent programming approaches that provide certain execution guarantees; they support the programmer in avoiding delicate concurrency errors such as data races or deadlocks. Providing these guarantees can, however, be at odds with attaining good performance. Pure message-passing approaches face the difficulty of how to transfer data efficiently between actors; and optimistic approaches to shared memory access, such as transactional memory, have to deal with recording, committing, and rolling back changes to memory. For this reason, execution strategies have to be developed that preserve the performance of a language while maintaining its strong execution guarantees.

This work focuses on SCOOP [3], an object-oriented approach to concurrency that aims to make concurrent programming simpler by providing higher-level primitives that are more amenable to standard pre/postcondition reasoning. To achieve this goal, SCOOP places restrictions on the way concurrent programs execute, thereby gaining more reasoning capabilities but also introducing performance bottlenecks. To improve the performance while maintaining the core of the execution guarantees, this paper introduces a new execution model called SCOOP/Qs[1]. We give a formulation of the SCOOP semantics which admits more concurrent behaviour than the existing formalizations [1], while still providing the reasoning guarantees. On this basis, lower-level implementation techniques are developed to make the scheduling and interactions between threads efficient. These techniques are applied in an advanced prototype implementation [4]. The overall performance is compared to a broad variety of other paradigms for parallel and concurrent programming – C++/TBB, Go, Haskell, and Erlang – demonstrating SCOOP's competitiveness.

A companion report [5] provides more details and, in addition, a formalization of the execution model, implementation notes, and an evaluation of the individual optimizations.

2. Execution Model

SCOOP aims to provide areas of code where pre/postcondition reasoning exists between independent threads. To do this, it allows one to mark sections of code where, although threads are operating concurrently, data races are excluded entirely. For example, consider the following two programs that are running in parallel.

```
separate x
  do
    x.foo()
    a := long_comp()
    x.bar()
  end
```

Thread 1

```
separate x
  do
    x.bar()
    b := short_comp()
    c := x.baz()
  end
```

Thread 2

Supposing that x is the same object in each thread, there are only two possible interleavings:

```
x.foo(), x.bar(), x.bar(), x.baz()   or
x.bar(), x.baz(), x.foo(), x.bar()
```

However, in contrast to `synchronized` blocks in Java, these `separate` blocks not only protect access to shared memory, but also initiate concurrent actions: for both threads, the calls on `x` are performed asynchronously, thus for Thread 1, `x.foo()` can execute in parallel with `long_comp()`. It *cannot* be executed in parallel with `x.bar()` as they have the same target, `x`. SCOOP has another basic operation, the query, that provides synchronous calls. It is so called because the sender expects an answer from the other thread; this is the case with the `c := x.baz()` operation, where Thread 2 waits for `x.baz()` to complete before storing the result in `c`.

The SCOOP model associates every object with a thread of execution, its *handler*. There can be many objects associated to a single handler, but every object has exactly one handler. In the example, `x` has a handler that takes requests from Threads 1 and 2. The threads that wish to send requests to `x` must register this desire,

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the author/owner(s).

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
ACM 978-1-4503-3205-7/15/02
http://dx.doi.org/10.1145/2688500.2688545

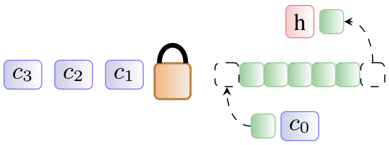
[1] Qs is pronounced "queues", as queues feature prominently in our new approach; the runtime and compiler associated with Qs is called Quicksilver, available from [4].

expressed by `separate x`. The threads are deregistered at the end of the `separate` block.

This model is similar to other message passing models, such as the Actor model. What distinguishes SCOOP is that the threads have more control over the order in which the receiver will process the messages: since each thread registers with the receiver, the messages from a single separate block to its handler will be processed in order, without any interleaving. This is summarized by the key reasoning guarantees that an implementation of SCOOP must provide:

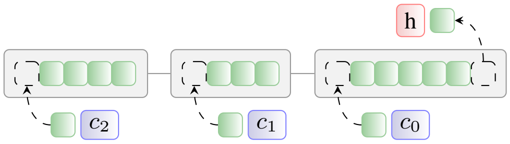
1. Regular (non-`separate`) calls and primitive instructions execute immediately and are synchronous.
2. Calls to another handler, `h`, on which object `x` resides, within the body of a `separate x` block will be executed in the order they are logged, and there will be no intervening calls logged from other clients.

The original SCOOP operational semantics [3] mandated the use of a lock to ensure this behaviour. One can visualize this as the client c_0 placing the calls in a queue for the handler h to dequeue and process:



The other clients (c_1, c_2, c_3) that may want to access the handler's queue must wait until the current client is finished.

The SCOOP/Qs semantics, in contrast, gives each client their own private queue in which to place their requests, thus preventing clients from interfering with one another. Each client then just shares this private queue with the handler to which it wants to send requests:



This nested queueing maintains the reasoning guarantees while still allowing all clients to enqueue asynchronous calls without waiting.

3. Implementation

To evaluate the change in semantics, and explore lower-level optimizations, a compiler and run-time were constructed. The compiler uses the LLVM framework to generate optimized code, and the accompanying C run-time is also very efficient.

An important aspect of the new semantics is that each of the queues (the outer queue-of-queues and the private queue) can be implemented by specialized concurrent queue structures. The queue-of-queues is a multiple-producer-single-consumer (MPSC) pattern because only a single handler will dequeue from it; there is an efficient, specialized, implementation of the MPSC queue. The private queues however are single-producer-single-consumer (SPSC), which can be made even more efficient than the multiple-producer case. These queues are lock-free in the common case, only blocking when the queues are empty.

Another key run-time optimization involves, on shared-memory systems, the execution of synchronous calls. There are two optimizations in this area:

1. Have the client, instead of the handler, execute the call.
2. Group synchronizations so that multiple back-to-back synchronous calls to the same handler synchronize only once.

Having the client execute the call means that the call is known statically and can be optimized (i.e., inlined) as normal; this isn't possible when the call is sent to the handler as a function pointer. To maintain the reasoning guarantees, the client must first send a synchronizing message to the handler (sync) to make sure that it is not executing any other asynchronous calls.

Naïvely, each sync call requires a round trip to the handler. A run-time optimization remembers the sync-state of the handler and the client turns subsequent sync operations into no-ops, until an asynchronous call is made; this can also be done statically through our custom LLVM optimization pass.

When a client is copying large amounts of memory, as in many parallel data processing operations, these optimizations make the code about as efficient as `memcpy`. On data-intensive parallel benchmarks this increases performance by 100× on average.

4. Language Comparison

We compared SCOOP/Qs with four well-established languages: C++/TBB (Threading Building Blocks), Erlang, Go, and Haskell. To rigorously evaluate performance, two types of benchmarks are used: the *parallel* problems [6] focus on numerical processing and working over large arrays and matrices; the *concurrent* problems focus on coordination between different threads or handlers. External review by language experts [2] increased confidence in the overall quality of the benchmark set.

The parallel benchmark times (above the dashed line) on 32 cores, and the concurrent task times (below the dashed line), are given here in seconds:

Task	C++	Erlang	Go	Haskell	SCOOP/Qs
randmat	0.08	4.14	0.08	1.03	0.24
thresh	0.11	11.96	0.17	0.50	2.30
winnow	0.15	23.95	0.28	0.52	2.59
outer	0.14	8.05	0.67	0.36	0.91
product	0.12	11.33	0.13	0.15	1.36
chain	0.32	16.01	2.60	2.94	0.60
chameneos	0.32	8.67	2.40	61.97	4.19
condition	15.92	2.15	5.95	26.05	1.46
mutex	0.14	6.13	0.17	0.86	0.14
prodcons	0.40	8.78	0.66	2.99	0.88
threadring	34.13	3.30	13.98	57.44	5.08
geom. mean	0.46	7.41	0.75	2.68	1.10

Note that neither Go nor C++/TBB offers any of the guarantees of SCOOP/Qs, and SCOOP/Qs offers more guarantees than Erlang. This places SCOOP/Qs as the best performing of the data-race-free languages (Haskell, Erlang).

Acknowledgements

This work was partially supported by ERC grant CME #291389.

References

[1] B. Morandi, M. Schill, S. Nanz, and B. Meyer. Prototyping a concurrency model. In *Proc. ACSD'13*, pages 170–179. IEEE, 2013.

[2] S. Nanz, S. West, K. Soares da Silveira, and B. Meyer. Benchmarking usability and performance of multicore languages. In *Proc. ESEM'13*, pages 183–192. IEEE, 2013.

[3] P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zurich, 2007.

[4] Quicksilver, an implementation of the SCOOP/Qs model. `https://github.com/scottgw/quicksilver`, Sept. 2014.

[5] S. West, S. Nanz, and B. Meyer. Efficient and reasonable object-oriented concurrency. `http://arxiv.org/abs/1405.7153`, 2014.

[6] G. V. Wilson and R. B. Irvin. Assessing and comparing the usability of parallel programming systems. Technical Report CSRI-321, University of Toronto, 1995.