Efficient and Reasonable Object-Oriented Concurrency

Scott West*
Google Inc., Switzerland
scottgw@google.com

Sebastian Nanz Bertrand Meyer
Department of Computer Science
ETH Zürich, Switzerland
firstname.lastname@inf.ethz.ch

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 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 both the underlying semantics to increase the amount of concurrent execution that is possible, exclude certain classes of deadlocks, and enable greater performance. These extensions are used as the basis of an efficient runtime and optimization pass that improve performance 15× over a baseline implementation. This new implementation of SCOOP is, on average, also 2× faster than other well-known safe concurrent languages. The measurements are based on both coordination-intensive and data-manipulation-intensive benchmarks designed to offer a mixture of workloads.

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

Programming languages and libraries that help programmers write concurrent programs are the subject of intensive research. Increasingly, special attention is paid to developing approaches that provide certain execution guarantees; they support the programmer in avoiding delicate concurrency errors such as data races or deadlocks. For example, languages such as Erlang [1] and others based on the Actor model [11] avoid data races by a pure message-passing approach; languages such as Haskell [19] are based on Software Transactional Memory [22], avoiding some of the pitfalls associated with traditional locks.

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 the language while maintaining the strong execution guarantees of the model.

This work focuses on SCOOP [18], an object-oriented approach to concurrency that aims to make concurrent programming simpler by providing higher-level primitives that are more amenable to standard programming techniques, such as 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 of SCOOP programs while maintaining the core of the execution guarantees, this paper introduces a new execution model called SCOOP/Qs[1]. We first give a formulation of the SCOOP semantics which admits more concurrent behaviour than the existing formalizations [16], 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 [20].

The design and implementation choices are evaluated on a benchmark suite that includes computation-intensive and coordination-intensive workloads, showing the advantages of the SCOOP/Qs execution strategies. 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.

The remainder of this paper is structured as follows. Section 2 introduces SCOOP and formally specifies executions. Section 3 describes the implementation techniques for this model. Section 4 evaluates the effectiveness of the different optimizations. Section 5 compares SCOOP/Qs against a variety of other paradigms. An analysis of related work is performed in Section 6, and conclusions are drawn in Section 7.

*All research was done while employed at ETH Zürich; opinions in this paper do not necessarily reflect those of Google Inc.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786822

[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 [20].

2. EXECUTION MODEL

The key motivation behind SCOOP [18] is providing a concurrent programming model that allows the same kinds of reasoning techniques that sequential models enjoy. In particular, SCOOP aims to provide areas of code where pre/post-condition reasoning exists between independent threads. To do this, SCOOP allows one to mark sections of code where, although threads are operating concurrently, data races are excluded entirely.

2.1 A Brief Overview

In Fig. 1 one can see two programs that are running in parallel. Supposing that `x` is the same object in each thread,

```
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

Figure 1: A simple SCOOP program

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()`. However, 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 Fig. 1, `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, which is expressed in the code 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 [11]. What distinguishes SCOOP from languages like Erlang [1] is that the threads have more control over the order in which the receiver will process the messages. When multiple processes each send multiple messages to a single receiver in Erlang, the sending processes do not know the order of processing of their messages (as they may be interleaved with messages from other processes). In SCOOP, 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 ordering gives the programmer the ability to reason about concurrent programs in a sequential way within the **`separate`** blocks. To be precise, pre/postcondition reasoning can be applied to a **`separate`** object protected by a **`separate`** block, even though the actions are being executed in parallel. A **`separate`** object is marked as such by the type system, and methods may only be called on a **`separate`** object if it is protected by a **`separate`** block. Maintaining reasoning among multiple independent **`separate`** objects is also possible, but requires all **`separate`** objects concerned be protected by the same **`separate`** block.

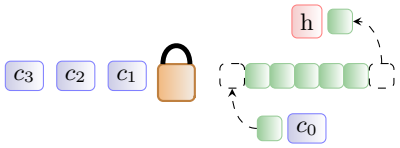


Figure 2: Normal handler implementation

The original SCOOP operational semantics [18] mandated the use of a lock to ensure that pre/postcondition reasoning could be applied by a client on its calls to a handler. One can visualize this as the client c_0 placing the calls in a queue for the handler h to dequeue and process, as in Fig. 2. 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.

2.2 Reasoning Guarantees

There are a few key reasoning guarantees that an implementation of SCOOP must provide:

1. Regular (non-**`separate`**) calls and primitive instructions (assignment, etc.) 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 effect of rule 1 is that normal sequential reasoning is applied to calls that are issued by the client, to the client. Rule 2 implies that calls that are made from the client to the handler are applied in order, thus the client can apply pre-/postcondition reasoning from one call it has made to the next.

2.3 The SCOOP/Qs Execution Model

The first SCOOP guarantee is easy to achieve, it is simply how sequential programs operate. To understand how to implement SCOOP efficiently, it is important to concentrate on the the second guarantee. This guarantee states that the requests from a particular client are processed by the handler in the order they are sent, disallowing interleaving requests from other clients. To prevent clients from interfering with one another on a particular handler can be achieved by giving each client their own private area (a queue) in which to place their requests. Each client then just shares this private queue with the handler to which it wants to send requests.

Syntax.

The following syntax of statements s is used to describe the execution model.

$$s ::= \texttt{separate}\ x\ s \mid \texttt{call}(x, f) \mid \texttt{query}(x, f) \mid \texttt{wait}\ h \mid \texttt{release}\ h \mid \texttt{end} \mid \texttt{skip}$$

Note that `separate` blocks and `call` and `query` requests model instructions of SCOOP programs, whereas the statements wait, release, end, and skip are only used to model the

$$\textsc{separate}\ \frac{}{(h, q_h, \texttt{separate}\ x\ s)\ \ ||\ (x, q_x, t) \Rightarrow (h, q_h, s; \texttt{call}(x, \mathsf{end}))\ ||\ (x, q_x + [h \mapsto []], t)}$$

$$\textsc{call}\ \frac{}{(h, q_h, \texttt{call}(x, f))\ ||\ (x, q_x, t) \Rightarrow (h, q_h, \mathsf{skip})\ ||\ (x, q_x\,[h \mapsto q_x[h] + [f]], t)}$$

$$\textsc{query}\ \frac{}{(h, q_h, \texttt{query}(x, f))\ ||\ (x, q_x, t) \Rightarrow (h, q_h, \mathsf{wait}\ x)\ ||\ (x, q_x\,[h \mapsto q_x[h] + [f, \mathsf{release}\ h]], t)}$$

$$\textsc{sync}\ \frac{}{(h, q_h, \mathsf{wait}\ x)\ ||\ (x, q_x, \mathsf{release}\ h) \Rightarrow (h, q_h, \mathsf{skip})\ ||\ (x, q_x, \mathsf{skip})}$$

$$\textsc{run}\ \frac{}{(h, [x \mapsto [s] + ss] + ys, \mathsf{skip}) \Rightarrow (h, [x \mapsto ss] + ys, s)}$$

$$\textsc{end}\ \frac{}{(h, [x \mapsto []] + ys, \mathsf{end}) \Rightarrow (h, ys, \mathsf{skip})}$$

$$\textsc{seq}\ \frac{(h, xs, s_1) \Rightarrow (h, xs, s_1')}{(h, xs, s_1; s_2) \Rightarrow (h, xs, s_1'; s_2)}$$

$$\textsc{seqSkip}\ \frac{}{(h, xs, \mathsf{skip}; s_2) \Rightarrow (h, xs, s_2)}$$

$$\textsc{parStep}\ \frac{Q \Rightarrow Q'}{P\ ||\ Q \Rightarrow P\ ||\ Q'}$$

$$\textsc{oneStep}\ \frac{P \Rightarrow Q}{P \Rightarrow^* Q}$$

$$\textsc{manyStep}\ \frac{P \Rightarrow^* P' \quad P' \Rightarrow^* Q}{P \Rightarrow^* Q}$$

Figure 3: Inference rules of SCOOP/Qs

runtime behaviour. In particular, statements wait and release describe the synchronization to wait for the result after a `query`, statement end models the end of a group of requests, and skip models no behaviour.

Operational Semantics.

In Fig. 3, an operational semantics that conforms to the SCOOP guarantees is given. It is described using inference rules for transitions of the form $P \Rightarrow Q$, where P and Q are parallel compositions of handlers. The $||$ operator is commutative and associative to facilitate appropriate reordering of handlers.

The basic representation of a handler is a triple (h, q_h, s) of its identity h, request queue q_h, and the current program it is executing, s. A request queue is a list of handler-tagged private queues, and is thus really a queue-of-queues. Private queues of a client handler c can be looked-up $q_h[c]$, and can be updated $q_h[c \mapsto l]$, where l is the new list to associate with the handler h. Both lookup and updating work on the *last* occurrence of c, which is important as this is the one that the client modifies. The queue can also be decomposed structurally, with $[x \mapsto s] + ys$ meaning that the head of the queue is from client x with private queue s, and ys is the rest of the structure (possibly empty). So although the private queues in the queue-of-queues can be accessed and modified in any order, they are inserted and removed in first-in-first-out order.

We describe the unique operations of Fig. 3: **`separate`** blocks (the rule SEPARATE), the two different kinds of requests (CALL, QUERY, SYNC rules), and how these requests are processed by the handlers (RUN and END rules). The sequential and parallel composition rules are defined in the standard way.

In the rule SEPARATE, clients insert their private queue at the end of the handler's request queue. This operation occurs at the beginning of a **`separate`** block. This registers them with the handler, who will eventually process the requests. The fact that a handler only processes one private queue at a time ensures that the reasoning guarantees are maintained. It is also a completely asynchronous operation, as the supplier's handler-triple only consists of variables, i.e., there are no restrictions on what state the supplier has to be in for this rule to apply. Additionally, the client appends a `call`(x, end) action before the end of the **`separate`** block to signal that the supplier x can take requests from other clients.

The SCOOP/Qs semantics, in contrast to the original lock-based SCOOP semantics, uses multiple queues that can all be accessed and enqueued into simultaneously by clients. This behaviour is visualized in Fig. 4, where the outer (gray)

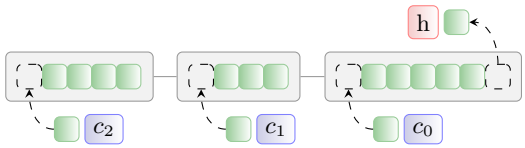


Figure 4: Handler implementation based on queue of queues

boxes are nodes in the queue of queues, and the inner (green) boxes are nodes in the private queues. This nested queueing maintains the reasoning guarantees while still allowing all clients to enqueue asynchronous calls without waiting.

In rule CALL, the `call` action is non-blocking: it asynchronously appends the requested method f to the end of the appropriate client's private queue.

Rule QUERY, requesting a query execution, however, does require blocking as it must wait for the result of the function application. This is modeled by sending the query request and introducing a pair of actions which can only step forward together: the wait/release pair. There is only one rule (SYNC) that can rewrite these into skip, and it can only do so when both processes are executing each of the pair.

Each handler processes its request queue in the following way: in rule RUN, if the handler is idle (executing skip) then it will examine the request queue. If the request queue's first entry (a private queue) is non-empty, then the first action is taken out of that private queue and placed in the program part of the handler to execute. If the request queue is empty, or it contains an empty private queue as its first entry, then the thread does nothing. In rule END, the thread finishes one private queue and switches to the next when it encounters the end statement, which was placed by the owner of the outgoing private queue when it finished executing its **`separate`** block (rule SEPARATE).

2.4 Multiple Handler Reservations

The **`separate`** block as shown so far only reserves a single handler, and this block provides race-freedom guarantees

between a single client and a single handler. However, a client may want to ensure consistency among multiple handlers or objects. To provide guarantees about multiple handlers, a multiple handler **separate** block must be used, as in Fig. 5. In this example, this has the effect that, whenever a client

```
separate x y
 do
  x.set (Red)
  y.set (Red)
 end
```

(a) Thread 1

```
separate x y
 do
  x.set (Blue)
  y.set (Blue)
 end
```

(b) Thread 2

Figure 5: Multiple reservations

reserves both `x` and `y`, the colours of each object will be the same, either both red or both blue. When written in this way and executed under either SCOOP or SCOOP/Qs, any client that comes after the execution of Thread 1 or Thread 2 (or both), and reserves `x` and `y` together will always see `x.colour = y.colour`. If using nested reservation, this may not be the case due to a possible race enqueueing the private queue into the queue-of-queues.

The modification to the SEPARATE rule to support this is straight-forward. First one defines an update function that updates a handler if it is in the set X.

$$\text{resOne}(X, h, (x, q_x, t)) = \begin{cases} (x, q_x + [h \mapsto []], t) & \text{if } x \in X \\ (x, q_x, t) & \text{if } x \notin X \end{cases}$$

Then this is applied over the parallel composition of all handlers.

$$\begin{array}{lcl} \text{resMany}(X, h, P \mid\mid Q) & = & \text{resMany}(X, h, P) \mid\mid \\ & & \text{resMany}(X, h, Q) \\ \text{resMany}(X, h, (x, q_x, t)) & = & \text{resOne}(X, h, (x, q_x, t)) \end{array}$$

Lastly, a function describes that each handler in the set (represented here by a list so it can be traversed) is sent an `end` message.

$$\begin{array}{lcl} \text{endMany}(x :: xs) & = & \texttt{call}(x, \texttt{end}); \text{endMany}(xs) \\ \text{endMany}([]) & = & \texttt{skip} \end{array}$$

These functions combine to define a generalized SEPARATE rule that can reserve multiple handlers atomically.

$$\text{SEPARATE}\frac{\begin{array}{c} P' = \text{resMany}(X, h, P) \\ \text{ends} = \text{endMany}(X) \end{array}}{\begin{array}{ll} (h, q_h, \texttt{separate } X\ s) & \mid\mid P \Rightarrow \\ (h, q_h, s; \text{ends}) & \mid\mid P' \end{array}}$$

2.5 Deadlock

Under the original handler implementation of SCOOP, the program in Fig. 6 will deadlock under some schedules. This is due to the inconsistent locking order of `x` and `y`. However, in the SCOOP/Qs execution model this example cannot deadlock because there are no longer any blocking operations: both clients can simultaneously reserve the handlers `x` and `y`, and log asynchronous calls on them. Deadlock is still possible in SCOOP/Qs, however one must also use queries (which block) to achieve the same effect. If `x.query` and `y.query` are added to the innermost **separate** blocks of Client 1 and Client 2, respectively, the program may deadlock.

```
separate x
 do
  separate y
   do
    x.foo()
    y.bar()
   end
end
```

Client 1

```
separate y
 do
  separate x
   do
    x.foo()
    y.bar()
   end
end
```

Client 2

Figure 6: Possible deadlock situation

3. IMPLEMENTATION

The semantics described in Section 2 are used to implement a compiler and runtime for SCOOP programs. The operational semantics gives rise to notable runtime performance and implementation properties. We pay particular attention to how to move the implementation from a synchronization-heavy model to one which reduces the amount of blocking.

The runtime for SCOOP/Qs is written in C, the compiler is written in Haskell and targets the LLVM framework [14] to take advantage of the lower level optimizations that are available. Using LLVM is a necessary choice for this work because it is important to compare with other more mature languages and the comparison should not focus on obvious shortcomings, such as a lack of standard optimizations. LLVM is also built to be extended; this work extends LLVM by adding a custom optimization pass. The SCOOP/Qs compiler, runtime, and benchmarks are available from GitHub [20].

The runtime is broken into 3 layers: task switching, light-weight threads, and handlers. Some of the optimizations described in this section take place at the handler layer, but there are also some that use the other two layers as well to optimize scheduling.

3.1 Request Processing

The RUN and END rules describe all of the queue management facilities that a handler has to perform. This correspondence is shown in the high-level implementation of the main handler-loop given in Fig. 7.

```
// RUN rule, when there is a private queue
//   available
while (qoq.dequeue (&private_queue))
  {
    // if dequeue returns true:
    //    RUN rule; process calls from
    //    this queue.
    // otherwise:
    //    END rule; switch to the next
    //    private queue
    while (private_queue.dequeue (&call))
      {
         execute_call (call);
      }
  }
```

Figure 7: Main handler-loop

The structure of the handler's loop directly corresponds to the data structure implementation (a queue of queues). One can see that private queues are continually taken from the outer queue, where the dequeue operation returns a Boolean result. False corresponds to no more work (indicating the

processor can shut down), not that the queue is empty as may be in a non-blocking queue implementation. For each private queue that is received, calls are repeatedly dequeued out of it and executed until false is returned from the dequeue operation, indicating that the END rule has been triggered, and the client presently does not wish to log more requests.

Note that the arrangement of clients and handlers follows a particular pattern when the queue-of-queues pattern is used. Namely, that each handler first reserves a position in the queue-of-queues: each queue-of-queues has many clients trying to gain access, but only one handler removing the private queues. This is a typical multiple-producer single-consumer arrangement, so an efficient lock-free queue specialized for this case can be used to implement the queue-of-queues. Similarly, once the private queue has been dequeued by the handler the communication is then single-producer single-consumer; the client enqueues calls, the handler dequeues and executes them. Again an efficient queue can be constructed to especially handle this case. These optimizations are important as they are involved in all communication between clients and handlers.

3.2 Client Requests

The handler-loop implementation, above, resides in the runtime library. The client-side is where the compilation and runtime system meet. In particular, the compiler emits the code allowing the client to package and enqueue requests for the handler, and handle waiting for the results of **separate** queries.

```
private_queue* h_p = client.queue_for (h);

// SEPARATE rule, adding an empty queue
// to the queue of queues
h.qoq.enqueue (h_p);

<compiled body>

// SEPARATE rule, compiler adds the
// code to enqueue the END marker
h_p.enqueue (END);
```

Figure 8: A compiled **separate** block

When a client reserves a handler with **separate** h **do** <body> **end**, this corresponds to the code shown in Fig. 8. The client receives a private queue h_p for the desired handler h, represented in the SEPARATE rule by the private queue appearing on the handler's queue-of-queues. This private queue can either be freshly created or taken from a cache of queues, to improve execution speed. The client then enqueues this new private queue on the queue-of-queues for the handler, which means the private queue is now ready to log calls in the body. Finally, corresponding to the end of the **separate** block, the constant denoting the end of requests is placed in the private queue, allowing the handler to move on to the next client.

There will typically be calls to the handler in the body of a **separate** block. The asynchronous calls are packaged using the libffi library [13], which abstracts away the details of various calling conventions. This packaged call is then put into the proper private queue for the desired handler. This can be seen in Fig. 9, the enqueue operation relating directly to the CALL rule. Packaging the call entails setting up the call interface (cif) with the appropriate argument and

```
arg_types[0] = &ffi_type_pointer;
arg_values[0] = &arg;
ffi_prep_cif(ffi_call, FFI_DEFAULT_ABI, 1,
             &ffi_type_void, arg_types);

// CALL rule, showing the setup via libffi.
h_p.enqueue(call_new(ffi_call, 1, arg_values));
```

Figure 9: Enqueueing an asynchronous call

return types with ffi_prep_cif, and then storing the actual arguments for later application by the handler. Note that the allocation of arguments and argument types for the call cannot be done on the client's stack because the call may be processed by the handler after the client's stack frame has been popped.

For efficiency reasons, a different strategy is used for synchronous calls (queries). This is because packaging a call involves allocating memory, populating structures, and the handler must later unpack it. In short: this takes longer than a regular function call. In the asynchronous case these steps are unavoidable because the execution of the call must be done in parallel with the client's operations. However, for synchronous calls this is not the case: the client will be waiting for a reply from the supplier when the supplier finishes executing the query. To make use of this optimization opportunity, for shared-memory systems, we can change the QUERY rule to the following:

$$\begin{array}{ll} (h, q_h, \mathsf{query}(x, f)) & || \ (x, q_x, t) \Rightarrow \\ (h, q_h, \mathsf{wait}\ x; f) & || \ (x, q_x\,[h \mapsto q_x[h] + [\mathsf{release}\ h]]\,, t) \end{array}$$

Note that the execution of the call f is shifted to the client, after the synchronization with the handler has occurred. This does not change the execution behaviour because, as in the original rule, all calls on the handler are processed before the query and the client does not proceed to log more calls until the query has finished executing. As can be seen from Fig. 10, the old rule first generates the call, sends it to the handler,

```
<packing same as async>
ffi_call(&ffi_call, f,
         &result, 0);
// QUERY rule
h_p.enqueue(ffi_call);
// SYNC rule
h_p.sync();
```

(a) Generated code for initial SYNC rule.

```
// New QUERY rule
h_p.enqueue(SYNC);
// SYNC rule
h_p.sync();
// New QUERY rule
result = f();
```

(b) Generated code for modified SYNC rule.

Figure 10: Executing a query f

and then synchronizes (Fig. 10a), these actions come from the combination of the QUERY and SYNC rule. The new rule just performs the call after synchronization occurs (Fig. 10b). This approach offers three main benefits:

- there is no memory allocation required,
- no encoding/decoding of the call is required, and
- which call is being made is known statically.

The last item is important, as now the underlying optimizer knows which call is being made, statically. This allows optimizations such as inlining.

One last optimization uses the knowledge that when the handler finishes synchronizing with a client, it will have no more work to do. Therefore, it control passes directly from the handler to the client, using the scheduling layer of the lightweight threads to avoid global scheduler. This optimization is safe, because the handler will otherwise just be idle, and avoids unnecessary context switching.

3.3 Multi-reservation Separate Blocks

The code generation for the multi-reservation separate block differs slightly from the single-reservation case which is optimized due to it being a simpler operation. One can see in Fig. 11 that some of the complexity is pushed into the client

```
client.new_reservations ();
client.add_handler (h1);
client.add_handler (h2);
client.reserve_handlers();

private_queue* h1_p = client.queue_for (h1);
private_queue* h2_p = client.queue_for (h2);

<compiled body>

h1_p.enqueue (END);
h2_p.enqueue (END);
```

Figure 11: A compiled 2-reservation **separate** block

run-time library. The run-time maintains structures that allow the multiple handlers to be stored. The interface between the compiled code and run-time consists of marking the start of a new set of reservations with `new_reservations`, adding a handler with `add_handler`, and finally safely reserving all handlers with `reserve_handlers`. The client can now retrieve the private queues that were just reserved; they do not need to be inserted into the handler's queue-of-queues because the reservation mechanism has already done that. Signalling the end of the private queue is done as before. Currently, the multiple reservation implementation uses one spinlock for every handler to maintain the ordering guarantees. However, since the number of memory accesses to enqueue in the queue-of-queues is quite small, a more sophisticated implementation could use transactional memory to implement the same behaviour. These spinlocks were not found to decrease performance.

3.4 Removing Redundant Synchronization

The SCOOP model prevents data races by mandating that one must access (read and write) separate areas of memory through their respective handlers. Due to this separation of memory spaces, it is common to copy data from one handler to another and then to work on the data. To do this, many synchronous calls must be issued to fetch data from one place to another.

Therefore, it is important to make synchronous calls efficient. This was partially addressed by using sync operations and executing the call on the client. There is a further enhancement that can be made to this approach, which is removing unnecessary sync calls altogether. A sync call is only performed to ensure that there is no more work on the handler, and the client can safely execute the function. Thus, a sync call is unnecessary if the previous call to the same handler was also a sync call.

3.4.1 Dynamic Avoidance

A dynamic method of eliding sync calls keeps the synced status in the private queue structure. When a sync call is made on a private queue, nothing happens if the queue is already synchronized (the private queue is empty and the handler is idle); the call merely returns and the synced status is unaffected. If the queue is not currently synced, the sync message is sent to the handler as usual and when it returns the synced flag is set in the handler reflecting that the handler is processing this private queue, but the queue is empty.

3.4.2 Static Removal

The static analysis starts by traversing the control flow graph (CFG). It annotates every basic block, basic blocks being sequences of basic instructions, with a set of handlers that are synchronized by the end of the block. This set of handlers is called the *sync-set*. The traversal of a function's basic blocks can be seen in Fig. 12. Each block acts as a

```
while changed ≠ ∅
    b ∈ changed, changed := changed − {b}
    common := ⋂ b.predecessors.sync_set

    if b.sync_set ≠ UpdateSync (b, common)
        b.sync_set := UpdateSync (b, common)
        changed := changed ∪ b.successors
```

Figure 12: Sync-set calculation for a function

sync-set transformer, adding and removing handlers from the set. As an initial input, the intersection of the sync-sets of all the block's predecessors is used. The traversal continues until every basic block's sync-set has stopped changing.

Of course this only says how the blocks are traversed, not how a given block's sync set is calculated given the instruction in that block. This is described in the UpdateSync function, shown in Fig. 13. Each type of instruction is han-

```
UpdateSync (b, synced):
    for inst ∈ b
        h := HandlerOf(inst)
        synced := synced ∪ {h}   if inst is sync.
                  synced − {h}   if inst is async.
                  ∅              if inst has side effects
                  synced          otherwise
    return synced
```

Figure 13: Sync-set calculation for a block

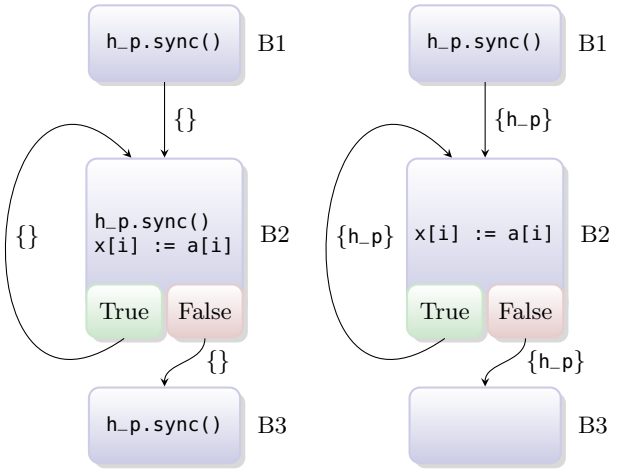
dled differently: synchronization calls add the target handler to the sync-set, asynchronous calls remove those handlers (and anything they may be aliased to), and arbitrary calls clear the sync-set entirely. Obviously this final case is quite severe, as it has to be, because a call could subsequently issue asynchronous calls on all the handlers currently in the sync-set. This can be mitigated by not emptying the sync-set for functions which are marked with the `readonly` and `readnone` flags. LLVM will automatically add these flags when it can determine that they hold.

The static analysis operates on LLVM bitcode, and is implemented as a standard LLVM pass (outside of the base compiler). Keeping the pass outside of the base SCOOP/Qs compiler has the advantage that it separates the generation

of code from the analysis and transformation of the generated control flow graph.

3.4.3 Example

The effect of the sync coalescing pass can be seen in Fig. 14. This program has three blocks, with sync operations in each



(a) A simple loop before the sync-coalescing pass.

(b) After sync-coalescing sync-sets label edges.

Figure 14: Sync-coalescing pass

one. Before the sync-coalescing pass, in Fig. 14a, the client is reading values out of a handler's array, for which a naïve code generator will produce a sync before every array read. Fig. 14b shows the results of the sync-coalescing pass in such a situation. The sync-sets are shown explicitly on the edges out of each block. In this case there are no calls that may invalidate a sync-set, so the handler `h_p` appears on all edges. The result of this is that the sync calls in blocks B2 and B3 can be removed. Removing sync calls in the body of a loop can greatly increase performance. Note that even though the sync call in the body of B2 was removed, `h_p` still appears on B2's outgoing edges because B2 doesn't invalidate the synchronization on `h_p` by issuing an asynchronous call.

The static analysis is important as it goes further towards getting SCOOP-specific operations out of the way of the optimization passes. In the end our implementation uses both the static and dynamic approaches. The static analysis is used when it can be, but it is necessarily conservative. For the cases where the static analysis fails to eliminate a sync call, the dynamic check will step in and eliminate the round-trip to the handler.

4. OPTIMIZATION EVALUATION

Here we examine the impact of the following optimizations (also outlined in Section 3) of applying no optimizations (**None**), dynamically and statically coalescing sync operations (**Dynamic**, **Static**), using the queue-of-queues structure (**QoQ**), and finally applying all optimizations (**All**). Note that the **None** variant is already quite efficient, lacking only the SCOOP-specific optimizations.

To rigorously evaluate performance, two types of benchmarks are used: *parallel* and *concurrent*. The *parallel* benchmarks come from the Cowichan problem set [24] and focus on numerical processing and working over large arrays and matrices.

The programs include:

- randmat: randomly generate a matrix of size n_r.
- thresh: pick the top $p\%$ of a matrix of size n_r and construct a mask.
- winnow: apply a mask to a matrix of size n_r, sorting the elements that passed the mask based on their value and position, and taking only n_w from that sorted list.
- outer: constructing a matrix and vector based off a list of points.
- product: matrix-vector product.

These benchmarks can be sequentially composed together, the output of one becoming the input to the next, to form a chain. This chain is more complex and sizable than the individual and gives a more diverse picture of a language's parallel performance.

The *concurrent* problems focus on coordination between different threads or handlers. For this purpose, we have created three benchmarks that represent different interaction patterns:

- mutex: n threads all compete for access to a single resource, the threads do not depend on each other.
- prodcons: n producers and n consumers each operate on a shared queue; the queue has no upper limit so producers do not depend on consumers, but consumers must wait until the queue is non-empty to make progress.
- condition: n "odd" and n "even" workers increment a variable from an odd (even) to an even (odd) number. Each group depends on the other to make progress.

All of the above are repeated for m iterations. Finally to these we add two concurrency benchmarks from the Computer Language Benchmarks Game [6]:

- threadring: threads pass a token around a ring in sequence until the token has been passed n_t times.
- chameneos: colour changing "chameneos" mate and change their colours depending on who they mate with. This is done n_c times.

The combination of these parallel and concurrent benchmarks gives us a balanced view of the performance characteristics of the approach.

All benchmarks were performed 20 times on a Intel Xeon Processor E7-4830 server (4×2.13 GHz, each with 8 cores; 32 physical cores total) with 256 GB of RAM, running Red Hat Enterprise Linux Server release 6.3. Language and compiler versions used were: gcc-4.8.1, go-1.1.2, ghc-7.6.3, erlang-R16B01. For the parallel benchmarks, the problem sizes used are $n_r = 10{,}000$, $p = 1$ and $n_w = 10{,}000$; for the concurrent benchmarks $n = 32$, $m = 20{,}000$, $n_t = 600{,}000$, and $n_c = 5{,}000{,}000$.

Results.

The results for the parallel and concurrent benchmarks can be seen in Table 1 and Table 2, respectively. They show that each optimization either improves or doesn't change the running time of each benchmark, however the magnitude of the effect differs depending on the work load.

Table 1: Times (in seconds) with optimizations applied on parallel benchmarks

Task	None	Dyn.	Static	QoQ	All	All/TC
randmat	56.73	0.65	0.22	47.07	0.22	0.24
thresh	123.14	3.37	2.51	94.54	2.40	2.30
winnow	116.33	3.55	2.72	95.14	2.58	2.59
outer	57.55	1.27	0.94	46.76	0.88	0.91
product	57.81	1.81	1.34	47.10	1.35	1.36
chain	8.47	0.68	0.63	6.67	0.59	0.60

Table 2: Times (in seconds) for optimizations applied on concurrent benchmarks

Task	None	Dyn.	Static	QoQ	All	All/TC
chameneos	21.93	10.99	10.80	14.08	4.76	4.19
condition	3.13	3.04	3.02	1.52	1.54	1.46
mutex	1.10	1.12	1.08	0.69	0.68	0.14
prodcons	2.76	2.45	2.41	1.97	1.59	0.88
threadring	21.74	16.17	16.09	14.54	5.24	5.08

In particular, **Dynamic** and **Static** sync coalescing dramatically improve the running time of SCOOP on the parallel benchmarks, due to the copying of large arrays between handlers. This improvement is between 12× and 250×, which puts solving data-heavy problems within reach of SCOOP, which it was *not* previously. The effect of the **QoQ** optimization is still significant (around a 20% improvement), but not as dramatic.

The benefit of **QoQ** in the concurrent benchmarks, where the improvements move between 1.5× and 2× for individual optimizations, is more comparable to the sync coalescing optimizations. This is not entirely surprising as **QoQ** increases the amount of inherent concurrency in the execution. The concurrent benchmarks also benefit from sync coalescing, though mostly on chameneos and threadring. chameneos and threadring are also interesting because applying all optimizations results in a faster execution than applying any single optimization by itself (as it tended to be in other benchmarks). Therefore, the sync coalescing and **QoQ** can also be quite effective in combination. prodcons and mutex are, in particular, improved by TCMalloc due to the heavy use of asynchronous calls (thus many small allocations by many threads) by all workers in mutex, and the producers in prodcons.

Over all benchmarks, the geometric mean for All/TC (using all optimizations and the TCMalloc implementation) is about 12× faster than no optimizations. This shows that the optimization techniques provide a new level of performance for SCOOP programs.

5. LANGUAGE COMPARISON

This section presents a comparison of SCOOP/Qs with four well-established languages. We have chosen the following languages: C++/TBB (Threading Building Blocks) [12], Erlang [1], Go [8], and Haskell [19]. To make the diversity of the languages clear, we present an overview of their characteristics in Table 3. The Memory column refers to how memory is shared between threads. Erlang copies data (messages) between processes, so there is no sharing. In SCOOP/Qs the programmer is only able to access shared memory through a handler. Repa is a Haskell library to parallelize certain pure functions over arrays.

To increase confidence in the overall quality of the benchmark set, experts reviewed the parallel benchmarks for C++ and Go [17]; Erlang's parallel benchmarks also received external review. For C++, Go, Haskell, and Erlang, the chameneos and threading implementations are taken from the Language Benchmark Game.

Table 3: Language characteristics

Language	Races	Threads	Type	Memory	Approach
C++/TBB	yes	OS	Imper.	Shared	Traditional
Go	yes	light	Imper.	Shared	Goroutines
Haskell	no	light	Func.	STM	STM/Repa
Erlang	no	light	Func.	Unshared	Actors
SCOOP	no	light	O-O	Unshared	Active Objects

5.1 Parallel Benchmarks

The parallel benchmarks are meant to measure how well a language can handle taking a particular program and scaling it given more computational resources (cores). Note that it is common in the Erlang and SCOOP/Qs implementations of the Cowichan problems that a significant amount of time is spent sharing results among the threads. To more clearly see the effect of communication, we distinguish the time spent computing versus the time spent communicating the results.

Execution Time.

We can see the graph of performance given 32 cores in Fig. 15. The general trend seen in this figure is also present with 1, 2, 4, 8, and 16 cores (limited space allows to present the precise timing data, Table 4, only for 32 cores).

Table 4: Parallel benchmark times on 32 cores (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

As with SCOOP/Qs, to give a clearer picture of the performance characteristics of Erlang, we also distinguish in Fig. 15 the computation time from the communication time. We can see that SCOOP/Qs and Erlang both spend a majority of their time in communication, with the exception of the chain problem, which has much less communication between the workers. For less fine-grained tasks, like the chain problem, the communication burden is much lighter. Erlang has unfavorable absolute performance results compared to the other languages due to it having no compact array structure available.

Besides Erlang, the other languages are more closely grouped. The geometric means for total time are, in increasing order: C++/TBB (0.32s), Go (0.57s), Haskell (0.89s), SCOOP/Qs (1.32s), and Erlang (18.07s).

Scalability.

The other aspect that we investigated was the speedup of the benchmarks across 32 cores. In Fig. 16 we can see the performance of the various languages on the different problems. We can see that on chain, most languages manage to achieve a speedup of at least 5x. Go is the exception to this, and performance decreases past 8 cores. Erlang also sees a performance degradation, though only past 16 cores.

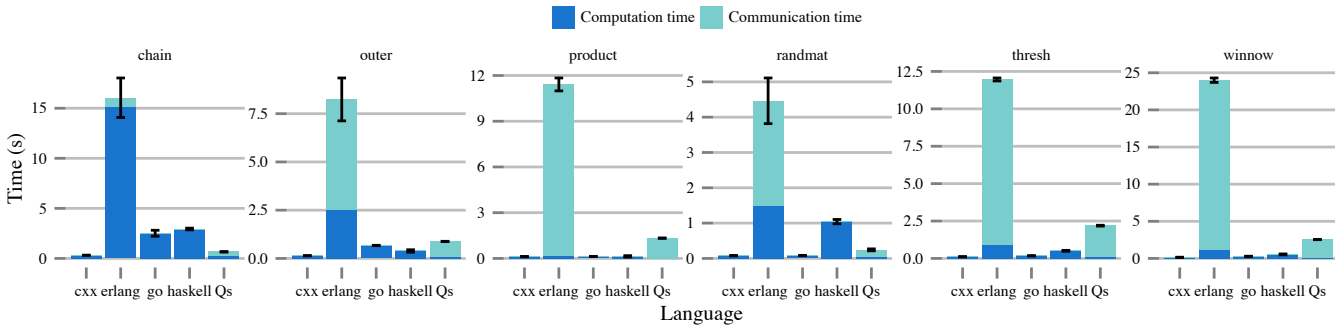


Figure 15: Execution times of parallel tasks on different languages, executed on 32 cores

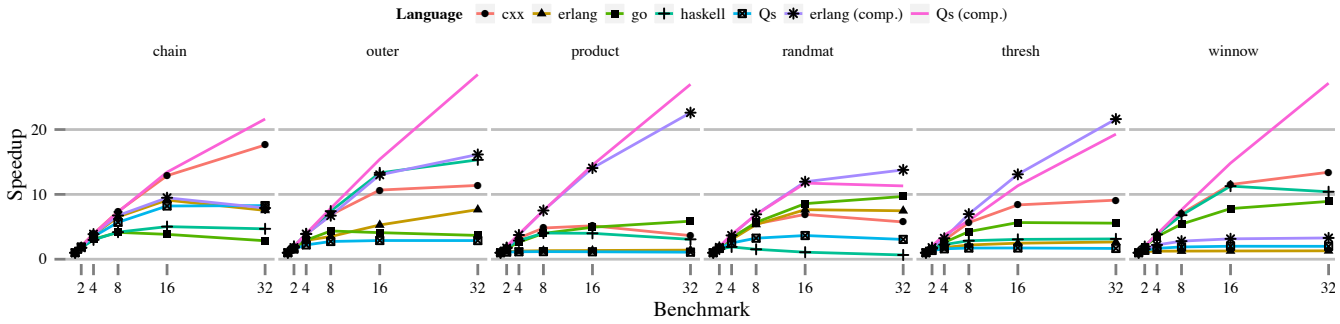


Figure 16: Speedup over single-core performance, up to 32 cores

5.2 Concurrent Benchmarks

The concurrent task times are given in Table 5.

Table 5: Concurrent benchmark times (in seconds)

Task	C++	Erlang	Go	Haskell	SCOOP/Qs
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

Haskell tends to perform the worst, which is likely due to the use of STM, which incurs an extra level of bookkeeping on every operation. Erlang performs better, but in general lags behind the other approaches. C++/TBB tends to be the fastest, except in the condition and threadring benchmarks, which are both essentially single-threaded; they are designed to test context switching overhead in various forms. Go does quite well uniformly, never the fastest, but never the slowest. Lastly, SCOOP/Qs is the fastest overall, the fastest in condition, tying for fastest with C++ on mutex, and performing reasonably well in all other benchmarks. The times in increasing order of geometric means: SCOOP/Qs (1.31s), C++/TBB (1.57s), Go (1.82s), Erlang (5.01s), and Haskell (12.20s).

5.3 Summary

This evaluation presents a wide variety of approaches to concurrency and situates SCOOP/Qs among them. In particular, we can see that SCOOP/Qs is generally quite comparable on coordination or concurrency tasks, falling in the middle of the pack after Go and C++/TBB, but faring better than Erlang and Haskell. Note, however, that neither Go nor C++/TBB offers any of the guarantees of SCOOP/Qs, and SCOOP/Qs offers more guarantees than Erlang.

For all problems, concurrent and parallel, the geometric means are: C++/TBB (0.71s), Go (1.02s), SCOOP/Qs (1.31s), Haskell (3.30s), and Erlang (9.51s). This places SCOOP/Qs as the best performing of the safe languages.

6. RELATED WORK

Finding runtime and compiler optimizations is a vital research goal when developing programming approaches for concurrency and parallelism. While approaches in this area are based on a broad variety of concepts, and in this respect each require different solutions, this work profited from insights and discussions of a number of related works.

Cilk [2] is an approach to multi-threaded parallel programming based on a runtime system that provides load balancing using dynamic scheduling through work stealing. Work stealing [3] assumes the scheduling forms a directed acyclic graph. In contrast, we tolerate some cyclic schedules through the use of queues. Since we use queues, handler A can log work on handler B while handler B logs work on A, as long as they do not issue queries on one another (forcing a join edge). We are not strict: edges go into handlers from the outside, other than at spawn; this is actually the normal case when logging calls. Although Cilk has been extended into Cilk++ [7], this does not indicate a significant uptake of object-oriented concepts to ensure correctness properties such as race freedom.

X10 [5] is an object-oriented language for high performance computing based on the partitioned global address space model, which aims to combine distributed memory programming techniques with the data referencing advantages in shared-memory systems. Although there is a mechanism to ensure local atomicity through the keyword `atomic`, it

is opt-in, and as such admits programs with data races by default. The `async` blocks allow computations to run on different address spaces, but there is no way for the caller to ensure consistency between `async` blocks directed to the same address space. The help-first stealing discipline [10] in X10 offers that the spawned task is left to be stolen, while the worker first executes the continuation; this is in contrast to Cilk's work-first strategy where the spawned task is executed first. The help-first strategy has benefits as it avoids the necessity of the thieves synchronizing. This only applies because the thefts in a finish block in X10 are serialized in work-first, whereas they are not for help-first. This technique would not be directly applicable to our work because SCOOP/Qs waits only on the result of a single handler.

Aida [15] is an execution model that, like SCOOP, associates threads of control with portions of the heap. The technique is implemented on top of Habanero-Java [4], an extension of the X10 implementation for Java. When there is contention for a particular heap location, the "loser" rolls back its heap modifications, suspends, and appends itself (delegates) to the run queue of the winner, effectively turning two concurrent tasks into a single one. This is fundamentally different from the SCOOP model, which also has isolated heaps, but allows interaction between threads of control, and even provides reasoning guarantees on this interaction. Therefore the underlying mechanisms are fundamentally different, where Aida requires efficient heap ownership and conflict resolution via a parallel union-find algorithm, SCOOP/Qs requires efficient communication which is attained via novel and nested uses of specialized queue structures. Otello [25] extends the isolation found in Aida to include support for nested tasks.

Another object-oriented approach which, like SCOOP, associates threads of execution with areas of the heap is JCoBox [21]. It also makes the distinction (similar to **`separate`**) between references that are local and those that are remote, although this can only be applied per-class, not per-object as in SCOOP. Each CoBox contains a queue for incoming asynchronous calls, though the reasoning guarantees are weaker for JCoBox, so this structure can be simple. The synchronous calls in JCoBox are also executed locally, but no dynamic or static method to reduce communication, ensuring data race freedom, is performed.

Kilim [23] is a framework that supports the implementation of Actor-based approaches in Java. It improves message-passing performance by treating messages differently from other Java objects, in that they are free of internal aliases and owned by at most one Actor at a time. The messages arrive via explicitly declared mailboxes in the objects, which also do not provide the reasoning guarantees between messages that the SCOOP model provides. The Kilim mailboxes have, therefore, a more simplistic behaviour compared to the queue-of-queues approach in SCOOP/Qs. Kilim also sets new standards in creating lightweight threads, which are not tied to kernel resources, thereby providing scalability and low context switching costs. SCOOP implementations have previously been based on operating system threads, and using lightweight threads in SCOOP/Qs we can report similar improvements in scalability as observed by Kilim. Kilim is extended with ownership-based memory isolation [9] for messages to reduce the amount of unnecessary copying. Although not strictly a message-based model, SCOOP/Qs may be able to apply this technique to so-called expanded classes, which are more like standard C structures, and are presently copied when used as arguments to separate calls.

We summarize the above approaches by stating whether they offer *guards* (protection against races) and *delegation* (ability for one entity to give work to another).

- *No guarding, no delegation* – Cilk/Cilk++.
- *Partial guarding, delegation* – X10 allows delegation as the only way for one place to modify another. However, a place can asynchronously modify itself using the same mechanism, thus there may be races within a place.
- *Guarding, protective delegation* – Aida and Otello extend X10 with the ability to resolve races by rolling back changes and reducing the amount of concurrent execution.
- *Guarding, delegation* – JCoBox and Kilim both have different approaches to the actor/active object model. This implies strict guarding and delegation of actions.
- *Guarding, enhanced delegation* – SCOOP follows the actor approach, but also offers enhanced delegation by allowing clients to maintain pre/postcondition reasoning with the handlers that they are delegating to.

7. CONCLUSION

We have presented SCOOP/Qs, an efficient execution model and implementation for the SCOOP concurrency model. As many other programming models that ensure strong safety guarantees, SCOOP introduces restrictions on program executions, which can become performance bottlenecks when implemented naively, standing in the way of practicality and more widespread adoption. The key to our approach was a reformulation of the SCOOP guarantees in abstract form, allowing one to explore a larger design space for runtime and compiler optimizations than previous operational descriptions. In particular, it enabled us to remove much of the need for synchronization between threads, thereby providing more opportunities for parallelism. In the evaluation of our approach, we traced the impact of the key optimizations, and compared SCOOP/Qs with a number of well-known and varied approaches to concurrency and parallelism: C++/TBB, Go, Haskell, and Erlang. This confirmed that, on a broad benchmark including both coordination-intensive and computation-intensive tasks, SCOOP/Qs can compete with and often outperform its competitors.

SCOOP offers a method of controlling access to other actors which is more exclusive than typical Actor-like languages. In SCOOP, messages can be bundled together to provide better pre/postcondition reasoning between messages (calls). The underlying techniques used in SCOOP/Qs are an efficient way to offer temporary control of one active object, or actor, over another. As such the technique could also be used in approaches like JCoBox [21] or Kilim [23] to provide more structured interactions between entities.

In the future, we plan to further explore the utility of the private queue design, in particular the usage of sockets as the underlying implementation. To further investigate and advance the efficiency of the runtime, a SCOOP-specific instrumentation for the runtime, providing detailed measurements for the internal components, will be essential.

8. ACKNOWLEDGMENTS

This work was supported by ERC grant CME #291389.

9. REFERENCES

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in Erlang*. Prentice Hall, 2nd edition, 1996.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[4] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[6] Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`, 2013.

[7] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[8] Go programming language. `http://golang.org/`, 2013.

[9] O. Gruber and F. Boyer. Ownership-based isolation for concurrent actors on multi-core machines. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 281–301, Berlin, Heidelberg, 2013. Springer-Verlag.

[10] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[11] C. Hewitt, P. Bishop, and R. Steiger. A universal modular Actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[12] W. Kim and M. Voss. Multicore desktop programming with Intel Threading Building Blocks. *IEEE Software*, 28(1):23–31, 2011.

[13] libffi. `http://sourcware.org/libffi/`, Mar. 2014.

[14] LLVM. `http://www.llvm.org`, Mar. 2014.

[15] R. Lublinerman, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Delegated isolation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 885–902, New York, NY, USA, 2011. ACM.

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

[17] 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.

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

[19] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM.

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

[21] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[23] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[25] J. Zhao, R. Lublinerman, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Isolation for nested task parallelism. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 571–588, New York, NY, USA, 2013. ACM.