Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

Scalable Performance for Scala Message-Passing Concurrency

Andrew BATE [1]

Department of Computer Science, University of Oxford, UK

Abstract. This paper presents an embedded domain-specific language for building massively concurrent systems. In particular, we demonstrate how ultra-lightweight cooperatively-scheduled processes and message-passing concurrency can be provided for the Scala programming language on the Java Virtual Machine (JVM). We make use of a well-known continuation-passing style bytecode transformation in order to achieve performance that is several orders of magnitude higher than native JVM threads. Our library is capable of scaling to millions of processes and messages on a single virtual machine instance, and our runtime system will detect deadlock should it occur. Message-passing is over 100 times faster than Erlang, and context switching is 1000 times faster than native Java threads. In benchmarks, the performance of our library is close to compiled code.

Keywords. concurrency, Scala, message passing, lightweight threads, JVM

Introduction

Concurrent programming is often considered error-prone and difficult to scale. However, it is our belief that these problems are due to the prevalence of *shared-memory concurrency* and not due to the multiple threads of control.

Hardware is becoming increasingly parallel, with the number of cores taking the place of clock speed as the specification of first interest when describing a processor [1,2]. The massively parallel architectures of the future are unlikely to have uniform memory access [3], and thus shared memory will not be a good abstraction. Message-passing concurrency, on the other hand, models the world as a collection of independent parallel processes that share nothing and interact only by passing messages [4].

Message-passing programs can be built from millions of extremely lightweight processes that can run on a single processor, a multi-core processor, or on a network of processors [5]. Thus message-passing is appropriate for both local and distributed algorithms [6,7]. In contrast, the idioms of shared-memory concurrency are substantially different from those of distributed programming [6,8].

By exposing the potential parallelism as small well-defined and encapsulated logical tasks, existing programs built using message passing are able to scale with the increasing parallelism of the hardware on which they are run [9,10,11]. Experience has shown that this is not equally true of programs built with shared-memory concurrency.

It is also extremely difficult to prove correctness or to reason about fairness and efficiency in the presence of fine-grained locks and access to shared data structures [12]. In particular, reasoning about memory corruption and deadlock in shared memory systems is diffi-

[1]Corresponding Author: *Andrew Bate, Department of Computer Science, University of Oxford, Oxford, OX1 3QD, UK.*; E-mail: `andrew.bate@cs.ox.ac.uk`.

cult. For message passing, there exist well-developed mathematical models and robust tools for formal verification of design [13].

In this paper, we present an embedded domain-specific language (DSL) for message-passing concurrency, called CSO_2, which provides a judicious combination of the ideas from Hoare's CSP [4,13] and Milner's π-calculus [14] for the Scala programming language. In a similar way to the original CSO [15,16] and JCSP [17,18], it follows many of the principles of the occam-π [19,20] programming language, exchanging compiler-enforced safety for programmer checked rules, but allowing the model to be used in a mainstream programming language. In contrast to these previous approaches, however, our library offers low process management and communication overheads, all without any modification to the Java Virtual Machine (JVM).

In order to make message-passing concurrency and process-oriented programming a compelling alternative to shared-memory concurrency, we need fast inter-process communication and lightweight threads. The JVM, arguably the most widely deployed and targeted virtual machine, supports many languages, including Java [21], Scala [22], and Clojure [23]. However, unlike in Erlang [5] or occam-π [19], the JVM does not natively support lightweight threads or message-passing: a JVM thread corresponds to a thread provided by the operating system [24]. Thus, Java threads are too heavyweight to assign per HTTP connection in a web server or per transaction in a database.

Our library provides lightweight and fast cooperatively scheduled user-level processes. The speed of context-switching of CSO_2 processes is far superior to a mainstream concurrent programming language, and close to compiled native code. We achieve this performance by transforming the bytecode produced by the Scala compiler using a well-known continuation-passing style (CPS) transformation [25,26,27,28]. This transformation is necessary in order to achieve good performance when many more JVM threads are used than there are available processors. We also provide a lightweight runtime system, which includes support for deadlock detection, similar to that available for the Go programming language [29].

The syntax of our embedded DSL is inspired by that of CSO [15]. In our experience, its syntax is sufficiently concise and constructs are in a close correspondence to their CSP counterparts. Feedback from taught courses at the University of Oxford has reinforced to us its suitability.

Outline of the Paper

It is our intention that this paper be self-contained. In Section 1, we introduce the syntax of CSO_2 and, in Section 2, provide our rationale for implementing it as an embedded DSL. In Section 3 we describe the post-compilation CPS bytecode transformation. We describe our user-level scheduler in Section 4, report on our implementation of message-passing channels in Section 5, and provide details of our implementation of runtime deadlock detection in Section 6. In Section 7 we benchmark the performance of our concurrency library in three key aspects: (i) the ability to create many processes, (ii) speed of process creation, and (iii) speed of message passing. A survey of related work, considerations for possible future work, and concluding remarks are provided in Sections 8 through 10.

1. Introducing the Syntax of CSO_2

Owing to space constraints, we introduce only the principal constructs of CSO_2. However, we will demonstrate their use on a parallelised implementation of Quicksort. Since our syntax is similar to CSO [15], this section may be skipped should the reader already possess familiarity with this syntax.

1.1. Simple Processes

A process is constructed using the notation:

```
def ExampleProcess(x1: T1, ..., xn: Tn) = proc (name) { body }
```

where name is an optional[2] String used to identify the process, body is a sequence of statements (called the *process body*) to be executed when the process is run, and x1, ..., xn are the arguments to body with the respective types T1, ..., Tn. Any of the Ti may be ports (discussed below in Section 1.3).

If P is a process, then **run** P executes P in the current thread, whereas **fork** P executes P in parallel and allows the caller to continue.[3]

Our library is capable of multiplexing millions of these processes onto as many JVM threads as there are available hardware threads, in the manner described in Section 3. Each JVM thread is in turn mapped onto a kernel thread [24].

1.2. Parallel Composition

The parallel composition of processes is achieved using the || operator. For example, the composition of the four processes P1, P2, P3 and P4 is given as follows:

```
val comp = P1 || P2 || P3 || P4
```

A composition of parameterised processes P(0), P(1), ..., P(n) is denoted by:

```
|| (i <− 0 to n) (P(i))
```

where n may be determined at runtime.

If Q = P1 || P2 || ... || Pn, then **run** Q causes each of the processes P1, P2, ..., Pn to be run in parallel. The process Q terminates when all of the components Pi of its composition have terminated.

It is important to reaffirm the distinction between a thread and a process: a *process* is a lightweight object of type Process that is a specification of runtime behaviour, whereas a *thread* is a JVM thread running the body of some process. For example, the statement **run** (P || P) results in the execution of a parallel composition of two separate instances of the body of process P.

1.3. Ports and Channels

Processes communicate via channels. A channel on which values of type T may be read or written is an instance of a subclass of Chan[T] and consists of one output port of type ![T] and one input port of type ?[T]. Processes are parametrised with the ports on which they may read or write data, as described in the previous section.

Input ports and output ports are named from the perspective of the process. Therefore, an output port provides a method !(T): Unit to write values to the channel. Similarly, an input port provides a method ?(): T to read values from the channel.

Channels are either *synchronous* or *buffered*. For a synchronous channel, termination of the execution of a ! at the output port is synchronised with the termination of the execution of

[2]If name is not supplied, then the name of the process defaults to that of the method where it is defined. For example, **def** P = **proc** { ... } is equivalent to **def** P = **proc** ("P") { ... }.

[3]In the original CSO syntax, one would write P() to run P. However, forgetting to write the double parentheses was a common source of bugs, especially since the Scala type system distinguishes between a method definition with no formal parameter list and one with an empty list.

a corresponding ? at the input port. A buffered channel, however, does not synchronise any execution of ! with ?.

A OneOne[T] channel is a synchronous channel where no more than one process at a time may access its output port or its input port. Similarly defined are the channels ManyOne[T], OneMany[T], and ManyMany[T], where the naming convention is the restriction on sharing of the output port followed by that of the input port. Each of these channels have the following buffered counterparts: OneOneBuf[T](n), ManyOneBuf[T](n), OneManyBuf[T](n) and ManyManyBuf[T](n), respectively, where n is the buffer capacity.

Our library supports changing process topology by the communication of ports, as is available in occam-π [19], JCSP [30], and CSO [15]. This is equivalent to the mobility of channel names in the π-calculus [14].

Finally, a channel is closed by invoking its close method. Closing a channel enforces the contract that the channel will never be read from or written to again.

1.4. Alternation

A *guarded output event* is a construct of the form (guard &&& outport) $-!->$ { cmd }, where guard is a Boolean expression, outport is the output port of some channel, and cmd is a statement. We say that the event is *ready* when outport is ready to perform a ! at that instant and guard evaluates to true.[4] A *guarded input event* is an expression of the form (guard &&& inport) $-?->$ { cmd } whose semantics is defined analogously.

An **alt** consists of a collection of guarded events:

```
alt ( (guard1 &&& port1) --> { cmd1 }
    | ...
    | (guardn &&& portn) --> { cmdn }
    )
```

where each (guard &&& port) $-->$ { cmd } can be either a guarded input or output event.

A process executing an **alt** waits until it finds an event that is ready and then executes the corresponding cmd, after which the **alt** terminates. Note that it is the cmds that are responsible for performing the reading from or writing to a channel.

1.5. Example: Quicksort using Recursive Parallelism

We illustrate the use of the CSO_2 library with the application of the recursive parallelism pattern to Quicksort, whereby recursive method calls in the sequential code are replaced by a composition of parallel processes with the same effect [31]. In the base cases, the result is computed directly.

The base case of the Quicksort process is when the input channel contains no data, in which case we just close the output channel. Otherwise, if the input channel is non-empty, we choose the first element as the pivot, and recursively use two copies of QSort to sort the data that is less than the pivot, and greater than or equal to the pivot, respectively.

```
def QSort(in: ?[Int], out: ![Int]): Process = proc {
  attempt {
    val pivot = in?
    val toHigher, toLower, fromHigher, fromLower = OneOne[Int]

    def Partition = ... // As defined below
```

[4] We have chosen for our terminology to deviate slightly from that of Sufrin [15] for sake of clarity.

```
    // Compose the system and run it
    run ( Partition || QSort(toHigher, fromHigher) || QSort(toLower, fromLower))
  }{
    out.close // Close once all data has been received
  }
}
```

The construct **attempt** P Q executes P, and if P throws a Stop exception (for example, if a channel is closed), then it executes Q.

To perform the partitioning of elements, we define a nested process called Partition that is responsible for passing data to the correct recursive copy of the QSort process, and then reading back the sorted data from the recursive processes.

```
def Partition = proc {
  // Partition the data received around the pivot
  repeat {
    val n = in?
    if (n < pivot) toLower ! n else toHigher ! n
  }
  // Close channels to recursive processes once the final input is received
  toHigher.close; toLower.close

  // Now output the results
  Copy(fromLower, out); out ! pivot; Copy(fromHigher, out)
  out.close
}
```

where the construct **repeat** { body } will repeatedly execute the statements of body until it invokes an operation on a closed channel, and Copy is an auxiliary process that repeatedly copies values from in to out:

```
def Copy(in: ?[Int], out: ![Int]) = repeat { val n = in?; out ! n }
```

Of course, in practice, one would use limited recursive parallelism, whereby recursive processes are only spawned to a maximum depth; however, we have elided the logic necessary to achieve that for brevity.

2. Why an Embedded Domain-Specific Language?

In this section, we motivate our decision to design an embedded DSL [32] for Scala, as opposed to implementing (yet another) standalone programming language.

The historically prevalent approach has been to build standalone languages, with their own custom syntax. This has the advantage that the syntax and semantics of the language can be designed specifically for message passing. Go and occam-π are notable examples of this approach [19,29].

Conversely, implementing a new standalone language is a significant undertaking, involving a separate parser and compiler, and perhaps an interactive editor too. Considerable effort is spent implementing the common features of a general purpose programming languages, including variables, definitions, conditionals, and so on. The custom syntax for these common features also increases the learning curve of the new language.

The alternative approach to is implement message-passing concurrency as an embedded DSL within an existing host language, and thus retaining as much as possible of the existing

syntax and language features, without having to go to the trouble of defining a separate language. The DSL is then implemented as a library of definitions written in the host language, and familiarity with its syntactic conventions can be carried over to the DSL. Furthermore, this approach overcomes need to implement compiler optimisations for the sequential code in addition to the message passing code. In some benchmarks, the lack of compiler optimisations for sequential code has led to reduced performance for occam-π [6].

However, there are some downsides. It requires great care to preserve the abstract boundary between the DSL its host, and useful error messages may be more difficult to implement. In spite of this, it turns out that the Scala programming language is particularly well suited for hosting embedded DSLs, in particular because of four language features: higher-order functions, a rich type system supporting type inference, macros supporting compile-time metaprogramming, and a lightweight syntax supporting customisation.

Taken together, these features have allowed us to design an embedded DSL with syntax close to what one might provide for a corresponding standalone language. For the majority of the message-passing constructs in our DSL, we hope that the syntax bears close resemblance to its occam-π counterpart.

3. Lightweight Processes

In this section we describe our implementation of lightweight processes. We begin by giving an overview of the runtime system before presenting the bytecode transformation, with additional details of the scheduler and message-passing channels provided in later sections.

Our user-level scheduler is responsible for multiplexing possibly millions of Process objects onto as many JVM threads as there are hardware threads. In this way, we view each java.lang.Thread as a virtual processor. Each process is cooperatively scheduled.

The idea is as follows: Each process has an internal execute() method which is invoked by the user-level scheduler whenever the process is to begin running from a waiting state. Further, each process has an internal pause() method that moves the process from a running state to a paused state, and an internal resume() method that moves the process from a paused state to a waiting state; see Figure 1.

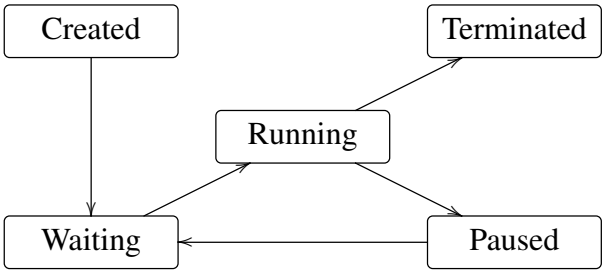


Figure 1. The various process states defined by the user-level scheduler with arrows indicating the possible transitions between states.

For example, when a process attempts to write to a buffered channel that is full, the pause() method of that process will be called. Later, when space is available, another process will invoke the corresponding resume(). Intuitively, pause() and resume() are similar to wait() and notify(), respectively, except that pausing does not block the underlying thread.

It is our user-level scheduler that is responsible for automatically managing the stacks of our processes. This cooperative scheduling is achieved through a *continuation-passing style*

(CPS) transformation and so works by rewriting the bytecode that is produced by the Scala compiler,[5] as is described in Section 3.1.

Optimisations employed in our implementation are discussed in Section 3.2. Difficulties arising in the implementation owing to restrictions placed upon us by the JVM are reported in Section 3.3, and difficulties caused by Scala's functional style are reported in Section 3.4. Performance considerations are discussed in Section 3.5, and issues relating to shared memory are detailed in Section 3.6.

3.1. Bytecode Transformation

The transformation described in this section is applied statically to the bytecode generated by the Scala compiler for the source program.

In accordance with previous work [25], we say that a method is *pausable* if it either directly or transitively (with respect to the call graph) invokes the pause() method.

The first step in the post compile-time transformation is to annotate each pausable method with the Pausable annotation. Furthermore, if a method overrides another method annotated as Pausable, then that method must also be annotated as Pausable. This is necessary because methods annotated as such will be subsequently transformed as described below, and this transformation involves the modification of method signatures. Clearly, the application of these annotations requires whole-program analysis.

Next, the resulting bytecode is further transformed using a well-known CPS transformation [25,26,27,28]. The technique we employ is a simple variant of single-shot delimited continuations [33] and largely identical to that of Srinivasan [25], and we give a brief account of that technique here.

A continuation represents the remainder of a computation from a given point in a program; hence, by using CPS, we can effectively suspend a process at the user level in order to resume it later. We use a restricted form of continuations that always transfer control back to their caller but maintain an independent stack.

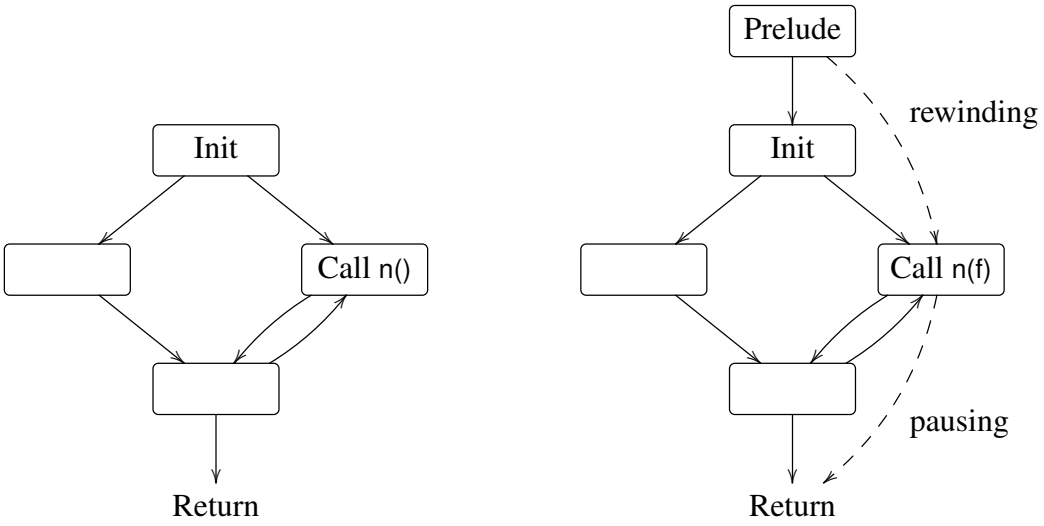
The signature of every method annotated as Pausable is transformed to include an extra parameter: the continuation object, known as a *Fiber* [34,35]. Each instance of a process has a Fiber associated with it. Whenever pause() is called from within such a method, the isPausing flag of the Fiber is set. This flag is later cleared by the scheduler when another process calls the corresponding resume().

In addition to the extra argument, each method annotated as Pausable has a so-called *prelude* block injected at the start, and each call site of a method annotated as Pausable is surrounded by both *pre-* and *post-call* sections. This is illustrated in Figure 2.

The purpose of these blocks can be understood as follows: Each pre-call block captures the current state of the activation frame, including the values of local variables and the program counter, and stores them in the Fiber passed as argument. In the corresponding post-call block, the isPausing flag of the Fiber is inspected: If unset, execution can continue unaffected. Otherwise, the process is pausing and hence the current method must return immediately (possibly with a dummy return value; see Section 3.3). This continues recursively all the way up until the internal execute() method of the process has been popped off the stack. Afterwards, the process is no longer running and is now paused.

The process will move from the paused state to the waiting state when another process calls the corresponding resume(). It then waits to be scheduled, at which point the user-level scheduler invokes the execute() method once again.

[5] We have developed a plugin for the Simple Build Tool (SBT), the de facto build tool for Scala projects, that launches the bytecode rewriter after compilation. Both the library and this plugin are available from the project website: `http://www.cso.io`.



(a) Before transformation. (b) After transformation.

Figure 2. A sample control flow graph of a pausable method that calls another pausable method, before (a) and after (b) the transformation. The transformation adds the prelude basic block (as described in the text) and two possible jumps in control flow (shown by dashed edges) for each pausable method n: one from the prelude to the call site of n that is to be followed when n is being restored onto the stack, and another from the call site of n to the end of the method that is to be followed when n is pausing.

This causes the process to be wound back onto the thread, one activation frame at a time, as follows: Each prelude section inspects the Fiber, detects that the stack is being rewound, restores the local state of the method, and jumps to the call site of the method that most recently triggered the pause. Thus the prelude section contains a sequence of conditional jumps to each call site of a method annotated as Pausable. After each frame has been restored, the process is in the running state once again and can continue execution where it left off.

Since all necessary logic is injected at the bytecode level, our library manages the stack with no changes to the source code. Thus our library is entirely portable, and is agnostic of any particular JVM implementation. For example, our approach is equally targetable towards any of the Oracle Hotspot, Oracle JRockit, IBM J9, or Google Dalvik virtual machines.

Previous work has shown that this technique, known as *stack-ripping*, allows for faster stack switching than any implementation based on the throwing and catching of exceptions [34,25].

3.2. Implementation Details and Optimisations

The bytecode rewriter is written in Scala and makes use of the ASM 4 library for bytecode transformation [36] which is the library used by the Scala compiler for bytecode generation.

In our implementation, we follow the ideas of Srinivasan [35], and optimise the transformation described in the previous section as follows: On the return of a method marked as Pausable, the caller examines the Fiber to determine whether the isPausing flag is set. If so, then the caller will store the local state needed upon resumption in the Fiber only if this has not already been done on an earlier pause/resume cycle, and then itself return control to its caller. This is more efficient then storing the state eagerly in the pre-call section.

Hence, if a method pauses for a second time, the states of intermediate frames are already stored in the Fiber and are not stored again. This explains the need for the four cases in the post-call section of Figure 3: the caller's behaviour depends on both whether the callee is pausing or returning normally, and whether the local state of the caller has or has not already been captured on an earlier pause/resume cycle.

```
void a() throws Pausable {

  x = ...

  b(); // b is pausable

  c(x); // c is not pausable
}
```

(a) Before transformation.

```
        void a(Fiber f) {
          switch (f.pc) { // prelude
            case 0: goto START;
            case 1: goto CALL_B;
          }
 START:   x = ...
CALL_B:   f.down(); // pre-call
          b(f);
          f.up();   // post-call
          switch (f.status) {
            case NOT_PAUSING_NO_STATE:
              goto RESUME;
            case NOT_PAUSING_HAS_STATE:
              restore state;
              goto RESUME;
            case PAUSING_NO_STATE:
              capture state;
              return;
            case PAUSING_HAS_STATE:
              return;
          }
RESUME:   c(x);
        }
```

(b) After transformation.

Figure 3. Example method before (a) and after (b) the insertion of logic for suspending and resuming the stack, using the optimisations described in the text. The transformation is on bytecode, hence the presence of `goto` statements. Example adapted from [35].

Another optimisation in this technique over traditional CPS transformations is that, because only those method signatures annotated as Pausable are rewritten, only those call sites that correspond to pausable method calls (i.e. those that may involve stack switching) need be transformed and added to the prelude. This allows external libraries that do not use CSO_2 constructs (including the Scala standard library) to be used without modification (but certain restrictions apply, as detailed in Section 3.4).

We also use two important optimisations employed by Srinivasan [35]: firstly, we use live variable analysis to store in the Fiber only the values of variables which are to be used again after the restoration point, and secondly we do not store the values of constants or variable aliases in the Fiber.

Evaluation has shown that the runtime overhead of the additional code injected to implement the prelude and all pre- and post-call blocks is low, and is certainly outweighed by the increased performance in context switching (see Section 7).

3.3. Living with the JVM Verifier

The JVM does not permit a method to inspect the stack frame of its caller, thus necessitating the design described in this paper. Nonetheless, as reported in Section 7, the performance of this technique is far in excess of Java primitives.

Furthermore, the JVM verifier requires that the operand stack is always the same size and contains the same types of values, irrespective of the code path that is taken to reach any particular point [37]. Therefore it is necessary to push onto the stack as many (dummy) constants of the expected types in the correct order as is required before issuing the jump in the prelude. For example, we push `iconst_0` whenever an `int` is expected.

After transformation, it is necessary to recompute the stack map table of each modified class file. These tables were introduced with Java 6 in order to speed up the runtime verification of classes by caching type information known to the compiler. The table records the type of values of local variables and the types of each operand stack slot immediately before specific bytecode instructions [37].

The `jsr` bytecode instruction, used to jump to a local subroutine defined within the body of a method, has caused implementation difficulties for previous bytecode rewriting frameworks [35,38]. However, the Scala compiler and all modern Java compilers no longer emit the `jsr` instruction (i.e. those that target bytecode version 51.0 and higher [37]), and thus our bytecode rewriter does not include logic for handling `jsr`.

3.4. Interaction with Functional Expressions

The Scala programming language offers both imperative and functional language features [22]. As of the current version of Scala (2.10.x), functional expressions, including for comprehensions, are implemented as standard library method calls. For example,

```
for (i <− 0 until n; j <− i until n) println(i)
```

is compiled to:

```
intWrapper(0).until(n).foreach(i: Int => intWrapper(i).until(n).foreach(j: Int => println(i)))
```

A difficulty arises when pausable methods are called from within anonymous functions passed to such standard library methods or external libraries. This would require those external methods to already have the Pausable annotation and the CPS transformation applied.

Our (partial) solution is to rewrite commonly used functional expressions, such as for comprehensions, as the equivalent imperative code at compile time using Scala macros. Since macros define abstract syntax tree transformations, they operate at the abstract source level rather than at the object code level, and so should be preferred wherever possible over bytecode transformation. For example, if we change the above expression to:

```
for (i <− 0 until n optimized; j <− i until n optimized) println(i)
```

our code is now compiled to:

```
var i: Int = 0
while (i < n) {
  var j: Int = i
  while (j < n) { println(i); j += 1 }
  i += 1
}
```

where `i` and `j` are fresh variable names. As a side benefit, the performance of the source program is optimised.

Nevertheless, there will always be pathological cases where such ad-hoc program transformations are unable to avoid delegation to library routines. In these cases, the transformation attempt will fail. Accounts of real user experience are needed to determine whether such rewritings alone are sufficient in practice. An alternative design, which would lift all aforementioned restrictions, would employ bytecode rewriting at runtime using a custom class loader. However, this could impact on performance.

3.5. Performance Considerations

Support for tail-call optimisations in the Scala compiler, which are unavailable in all Java compilers of which we are aware, are of great benefit in this setting: the storing and restoring of call stacks will obviously degrade performance when recursion is deep or call chains are large; tail-call optimisations can transform some recursive functions into equivalent iterative code, thus eliminating repeated passing of continuations.

Moreover, in process-oriented programming, systems are often built by the composition of processes, each of which have a specific well-defined task [39]. Thus, the lengths of call chains, and hence the depths of activation frames that are to be unwound and rewound, are expected to be shorter if process-oriented patterns are employed in users' programs. Therefore we expect the techniques outlined above to perform well in practice. Of course, however, benchmarks and case studies of real systems are needed.

3.6. Support for Shared Memory

Although we are primarily concerned with message-passing concurrency, for the reasons outlined in the introduction, our library also allows for shared-memory concurrency. This requires some additional changes to the bytecode in the post-compilation stage: If a pausable method calls another pausable method from within a synchronized block, then all usages of that object's monitor must be replaced with a newly introduced explicit lock which sets the thread affinity of the process to the current thread, and any call to wait(), notify() or notifyAll() must be replaced with the equivalent to calls to pause() and resume(). These changes are necessary to ensure that, after a call to resume(), a process is rewound back onto the same thread when it holds a lock.

4. Scheduling

The user-level scheduler is provided by our small runtime system. The scheduler manages the states of processes as described in the introduction of Section 3 and illustrated in Figure 1. We have experimented with two different scheduler algorithms, as reported below. The relative performance of these two schedulers is considered in Section 7.

Empirical evaluation has shown us that the best scheduling performance is achieved when one scheduler is responsible for each JVM thread (as opposed to one scheduler multiplexing processes across many threads). Thus, each thread has its own scheduler in an effort to improve cache affinity. Furthermore, previous work shows us that throughput is increased when the number of JVM threads matches the number of hardware threads on the system: more threads will likely result in a high context switching overhead, and fewer threads could result in higher cache misses due to the increased possibility of a thread being rescheduled on a different core [40].

4.1. Naïve First In, First Out Scheduler

The implementation of the FIFO scheduler is quite simple: the running process occupies the JVM thread; the processes that are paused have called pause() but no corresponding resume() has yet been invoked; and the processes that are waiting are placed in a queue. Processes enter the queue when a resume() is called, and leave once the JVM thread is unoccupied. In this scheme, processes do not migrate between threads when a resume() is invoked.

The scheduler is fair in that processes are scheduled in first in, first out order. One could experiment with different scheduling disciplines by specifying a process priority function and using a priority queue.

```
1: function WRITETOBUFFEREDCHANNEL(msg)
2:     // The ENQUEUE operation atomically stores a message in the ring buffer,
3:     // returning FALSE iff the buffer was full.
4:     while not ENQUEUE(msg) do
5:         // The PAUSE operation causes the current stack to unwind.
6:         PAUSE
7:     r := GETPAUSEDREADER
8:     RESUME(r)
```

Algorithm 1. Writing to a buffered channel.

4.2. Multi-Core Batch Scheduler

This scheduler is a simplified implementation of that provided by the occam-π runtime. It is intended that in a future release, we will provide a scheduler that implements both the work-stealing algorithm and all of the runtime heuristics detailed in [6].

The scheduler instance maintains a *run queue*, which is a linked list of *batches*. Each batch is a linked list of processes. The scheduler executes each batch by moving the processes it contains to the *active queue*. The *dispatch count* is calculated as a constant multiple of the batch size, bounded by a fixed constant. The dispatch count is decremented each time a process is taken from the active queue and executed. When the dispatch count reaches zero (or the active queue is empty), the processes in the active queue form a new batch which is appended to the end of the run queue (if non-empty), and the head of the run queue is moved to the active queue for execution. Since these data structures are manipulated only by the current thread, they can be maintained without locking.

When a process is resumed, and the dispatch count is non-zero, it is added to the end of active queue of the rescheduling thread, otherwise it is added to a new batch which is appended to the end of the run queue of the rescheduling thread. In this way, a process can migrate between threads during execution (unless prohibited by an explicit affinity setting, as detailed in Section 3.6).

The use of batches addresses the issue of cache thrashing that can occur with process-oriented designs [6]. By partitioning the run queue, we reduce the size of the working set for each thread, which should fit within the processor cache. Performance is improved by repeated execution of batches within the cache.

5. Channel Communication

Our library, CSO_2 provides all of the message-passing capabilities of the original CSO library [15], including synchronous and buffered channels, extended rendezvous, barriers, and port type variance. In particular, it provides a CSP-style generalized **alt**, borrowing the provably-correct implementation of Lowe [16]. Future work will consider how the performance of this primitive can be optimised in our setting of cooperative yielding.

Intuitively, the implementations of channels are as usual, except with calls to wait() and notify() replaced with calls to pause() and resume() on the appropriate processes. Owing to space constraints, we have illustrated this for the case of buffered channel communication only, in Algorithms 1 and 2.

For a single-core machine, since there will be only one JVM thread onto which all processes are cooperatively scheduled, it is safe to eliminate the locking used in the ENQUEUE and DEQUEUE operations because there will be no preemption.[6]

[6]We did experiment with several lock-free algorithms for buffered channel communication, but were unable

```
1:  function READFROMBUFFEREDCHANNEL
2:      // The DEQUEUE operation atomically fetches a message from the ring buffer,
3:      // returning NIL if the buffer was empty.
4:      msg := DEQUEUE
5:      while msg = NIL do
6:          // The PAUSE operation causes the current stack to unwind.
7:          PAUSE
8:          msg := DEQUEUE
9:      r := GETPAUSEDWRITER
10:     RESUME(r)
11:     return msg
```

Algorithm 2. Reading from a buffered channel.

Our library makes use of the specialization of primitives on the parametric polymorphic types of channels. In Scala, type specialization allows the compiler to generate primitive-specific versions of classes that have type parameters [41]. This feature is similar in spirit, although not identical, to that of C++ templates. Therefore the performance of communicating primitives is optimised in CSO_2 compared to CSO. For example, this feature is of particular use when building a concurrent solution to numeral integration.

One last feature of interest is the possibility of optimising *extended rendezvous* in our framework. Given a channel c that can communicate values of type T and a function f : T ⇒ U, an extended rendezvous is of the form c?f, and is evaluated by pausing the caller until a value v can be read from c and then computing f(v) before allowing the writer process to continue. However, this is existentially equivalent to f(c?) provided f does not perform any channel operations nor accesses shared memory. Since channel operations are tagged in the bytecode, we are able to optimise the former into the latter. The user needs to be aware however that with these optimisations switched on, should f perform any I/O, then this may occur in a different order.

6. Runtime Deadlock Detection

Our runtime system also includes deadlock detection for global deadlocks, similar to that available for the Go programming language [29]. Informal benchmarking showed that this feature adds little to no overhead to the runtime performance since our scheduler records those lightweight processes currently paused.

Once all processes are paused, we run the deadlock detection algorithm shown in Algorithm 3, which is due to [42]. In the algorithm, we say that process u is waiting for process v when u is attempting to communicate with v across some channel. The runtime reports the cycle of ungranted requests that caused the deadlock. For example, the output provided for a deadlocking run of the dining philosophers problem is as follows:

```
Deadlock detected! The cycle of ungranted requests is:
Philosopher0 -!-> Fork4          Fork0 -?-> Philosopher0
Philosopher1 -!-> Fork0          Fork1 -?-> Philosopher1
Philosopher2 -!-> Fork1          Fork2 -?-> Philosopher2
Philosopher3 -!-> Fork2          Fork3 -?-> Philosopher3
Philosopher4 -!-> Fork3          Fork4 -?-> Philosopher4
```

to find an algorithm that outperformed the use of locks. We conjecture that this is a result of instantiating only as many JVM threads as there are hardware threads, and thus the contention for locking is low. This is a topic for future work, especially as the number cores on commodity hardware increases.

```
1: // Input: contention network G = (V, E) where
2: //   V is the set of nodes representing processes, and
3: //   E contains all edges (u, v) such that u is waiting for v.
4: // Returns: TRUE if some sub-network is deadlocked.
5: function CHECKDEADLOCK(nodes,edges)
6:     interiorNodes := {u | (u, v) ∈ edges}
7:     // The leaves are the processes not waiting for any other process
8:     leaves := nodes \ interiorNodes
9:     // If a non-empty graph contains no leaf nodes then all nodes are waiting
10:    if (nodes ≠ ∅ ∧ leaves = ∅) then return true
11:    else
12:        // Construct the adjacency list
13:        Adj := {u ↦ {v | (u, v) ∈ edges} | u ∈ nodes}
14:        // Maintain set of visited nodes (processes known not to be deadlocked)
15:        visited := leaves
16:        // Starting from the leaves, perform a backwards mark and sweep
17:        q := MAKEQUEUE(leaves)
18:        while q ≠ ∅ do
19:            next := DEQUEUE(q)
20:            for all n ∈ Adj [next] do
21:                if n ∉ visited then
22:                    visited := visited ∪ {n}   // Mark n as deadlock free
23:                    ENQUEUE(q,n)
24:        // Graph is acyclic iff we have visited all nodes
25:        return nodes = visited
```

Algorithm 3. Deadlock Detection [42].

where the left-hand column shows the ungranted write requests and the right-hand column shows the ungranted read requests.

This is a level of diagnostics not available without great effort when using native Java threads and shared memory, and is of particular importance given that such software errors are inherently difficult to reproduce.

7. Performance Evaluation

In this section we evaluate the performance of our library against competing solutions. The source codes for the two benchmarks reported here and for each language/library tested are available from `http://www.cso.io`.

Erlang is a concurrent message-passing functional language [5], and Go is a compiled language that natively supports message-passing concurrency [29]. JCSP is a library which provides CSP-style concurrency primitives for Java [17], and occam-π is a concurrent programming language [19]. All are therefore suitable candidates for benchmarking our library against. The Scala Actors library is the language designers' recommendation for concurrency in Scala [22], and is thus also a suitable candidate.

The first performance benchmark measures the total time for n processes to each communicate a single message to every other process, for several values of n. Thus each test involved $O(n^2)$ messages in total.

The benchmarks were performed on a server with two 3.06 GHz Intel Xeon X5667 processors (each with four cores) and 96GB RAM running Windows Server 2008 R2 Enterprise x64, with JVM 1.7 update 21, Scala 2.10.1, JCSP 1.1-rc4, Erlang R16B and Go 1.0.3. For the

benchmarking of occam-π we installed CentOS 6.4 and KRoC 1.6.0 on the same hardware. Fifty times were recorded for each experiment, after allowing virtual machines to warm up. Some Erlang experiments reported very high times, which we believe, having inspected the memory graph in the task manager, were due to the garbage collector. These times were excluded; the variances of the remaining results in all experiments were small enough to be ignored. The times reported in Figure 4 are the mean wall times in milliseconds and include the time for context switching and process start-up.

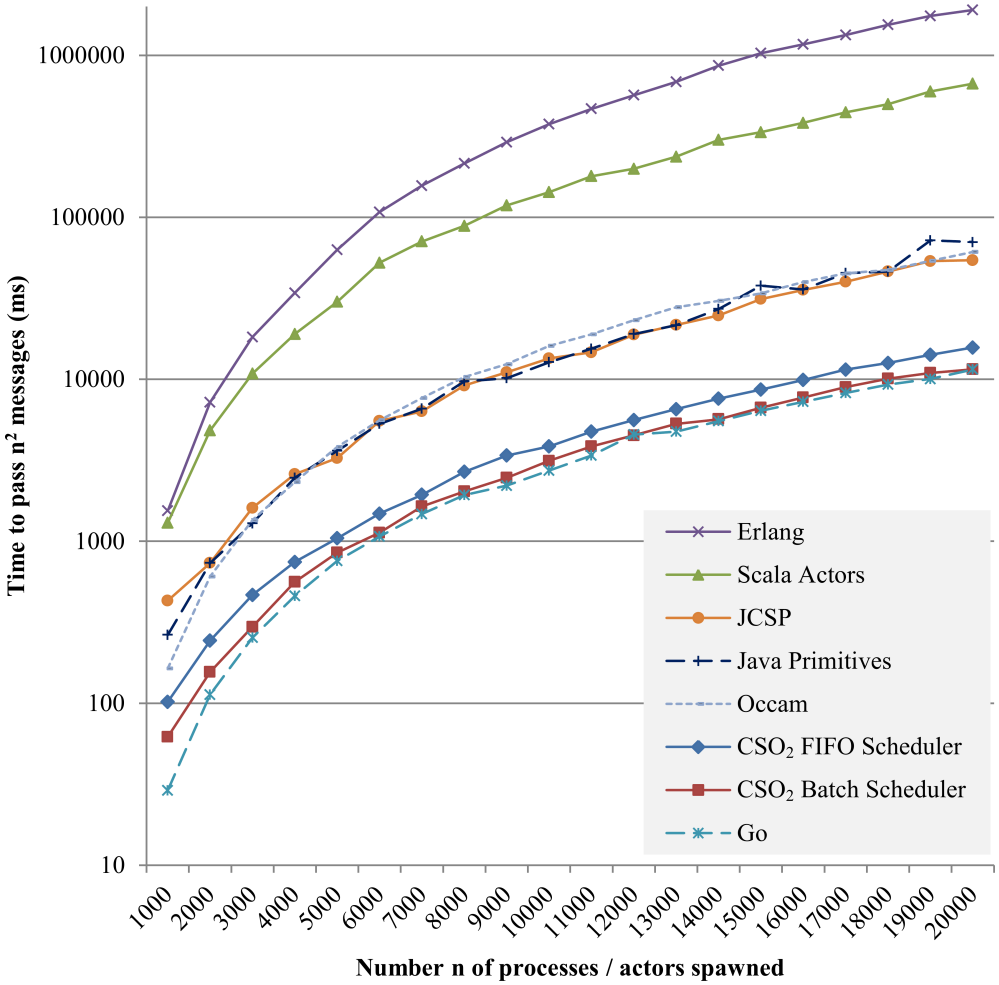


Figure 4. Benchmarking the performance of our CSO_2 library, using both the FIFO and batch scheduler, against several alternatives for speed of message passing. Time is the mean wall time in milliseconds for fifty runs on a logarithmic scale (lower is better).

The results show that our library and Go scales linearly in n^2 when communicating n^2 messages amongst n processes, and that our library achieves the best performance when using the batch scheduler.

From the graph, we can read off that the performance of the internal mechanism of the library, whereby processes are unwound from the stack whenever they need to pause and rewound back onto the stack when they are later able to continue, is almost as good as compiled Go code and well in excess of 100 times faster than Erlang.

When we inspected the raw data we found that our library, and also Go, provides exceptionally consistent timings even under heavy loads. In the case of our library, we attribute this both to the uniform way in which methods that perform concurrency operations are transformed in our framework and to the reliability of the JVM, particularly the engineering effort to reduce pauses during garbage collection [43].

We also timed the performance of both synchronous and buffered channel communications between exactly two processes, both with and without the use of an **alt** at either end of the channel. We found that the number of communications per second achieved by our library was equal to that of the original CSO library. Therefore we conclude that the chief utility of CSO_2 is when the programmer requires many more processes then there are available processors.

In the second benchmark we measured the mean wall time in milliseconds for n processes, whose communication graph formed a ring, to communicate a single message around the ring 300 times. As before, fifty times were recorded for each experiment, after allowing the JVM to warn up. The results are shown in Figure 5. It can be seen that in this benchmark the performance of our library is far in excess of Java primitives. However of greatest interest is the influence of the scheduler on the runtime performance: It demonstrates that grouping processes into cache affine batches and allowing processes to migrate between worker threads can result in greater throughput.

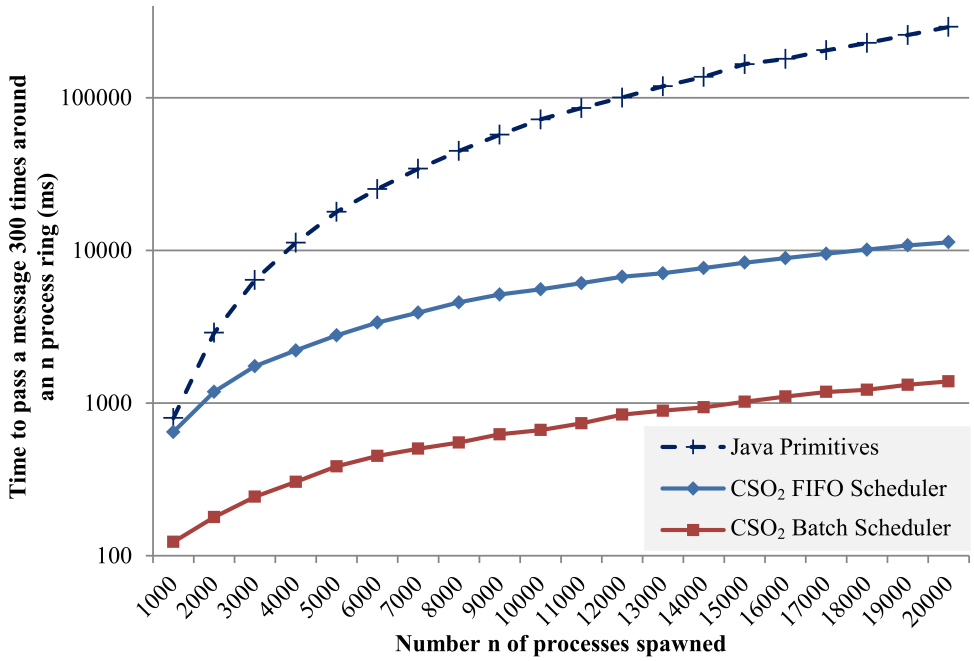


Figure 5. Benchmarking the performance of the batch scheduler against both the FIFO scheduler and Java primitives. Time is the mean wall time in milliseconds for fifty runs on a logarithmic scale (lower is better).

One should not place too much reliance on micro-benchmarks; however, this evaluation demonstrates that our library is capable of achieving excellent performance. Whilst we seek real experience with CPU-intensive applications, we feel that it is a safe conjecture that our library provides better performance than any other competing message-passing solution on the JVM.

8. Related Work

The work presented here builds on the ideas of [25,44,35,26,27,28]. A good survey of related techniques appears in [35].

Srinivasan implemented a similar continuation-passing style transformation for the Java Kilim actors library [35]. One significant difference is that Srinivasan requires that pausable methods be explicitly annotated in the user's source code. Our view is that these additional annotations clutter the source code of methods that need to perform concurrency operations regardless, and in Scala where lambda functions are supported as first-class citizens, this is

not always possible. We do acknowledge, however, that the use of explicit annotations does alert the programmer to the potential cost of message passing and stack switching [25].

Thus our approach is similar to Apache Javaflow [27], which transforms a method if it can reach an operation that causes the owner thread to suspend. However, in contrast to our approach, Javaflow unnecessarily transforms all methods reachable from that method, irrespective of whether or not those methods perform thread suspending operations. This can lead to a substantial increase in the size of the bytecode.

There have been a number of projects to develop lightweight threads at a lower level than we have investigated here. One example is the Capriccio project [45], a modified version of POSIX threads that is capable of scaling to hundreds of thousands of preemptively-scheduled user-level threads. The Native POSIX Thread Library also offers lightweight threads with greatly improved performance over the original LinuxThreads implementation [46].

Pettyjohn et al. [47] demonstrate how continuations can be supported in environments that allow neither stack inspection nor manipulation, such as the JVM or the CLR. However, their approach relies on tail-call optimisations, which are unavailable on the JVM (as of version 1.7, but are available in the Scala compiler). They also use exceptions to capture state and their transformations result in loops being split into several method invocations, which is likely to be expensive on the JVM [34]. Furthermore, since their transformation changes access modifiers to increase visibility of private methods, the transformed code is unsuitable for linking against by other developers.

Of particular importance is occam-π and the Kent Retargetable occam Compiler [19]. Occam natively supports lightweight threads and combines many of the features of both CSP and the π-calculus. The runtime overheads are low enough to support millions of concurrent processes. The occam-π scheduler groups processes into cache affine batches of communicating processes using runtime heuristics and process migration. It has been demonstrated that these techniques can lead to superior communication times [6].

9. Future Work

One interesting future direction for our library would be to allow the developer to define a user-level scheduler. These have proven to be of particular use in the performance tuning of applications that deal with database transactions, through the use of an earliest-deadline-first scheduler [25].

A formal verification of design for our library, as has been done for core components of CSO [16] and JCSP [48], is needed, and this is a consideration of future work.

In the long term, support for lightweight threads on the JVM at the execution engine layer is needed. The JVM does not permit any form of stack manipulation, and thus we have been forced to follow the design described in this paper. Whilst our approach is close in performance to a compiled concurrent programming language, it is clear that direct access to memory would allow for more efficient support of continuations. A lower-level implementation of these techniques would also allow for more efficient jumps to call sites within methods, as arbitrary jumps are disallowed by the bytecode verifier (as explained in Section 3.3). Support for lightweight threads in the JVM could improve context switching by a further two orders of magnitude [25].

An important area of work is to support distributed programming. Such an extended library would provide a uniform interface for both local concurrent and distributed programming, allowing systems to scale from one machine to many, with no architectural changes to the user's code.

Another area of future work will be in an improved runtime, supported with the use of a debugger. This will largely be a port of our existing work described in [42]. We believe that

the performance and usefulness of this debugging tool will be even greater in this setting, owing to the cooperative yielding of user processes. In particular, we would expect a reduction in the reordering of atomic actions caused by the overhead of the instrumentation of the debugger.

Finally, it should be relatively straightforward, if not development intensive, to bring many of the features of Erlang [5], such as process-linking and supervisor processes to our library, with good runtime performance.

10. Conclusions

We have presented an embedded domain-specific language for building massively concurrent systems using Scala on the JVM. The technique we have described here offers performance that is over 100 times faster than Erlang, and context switching that is 1000 times faster than competing libraries and native JVM threads. The performance of our library is also close to that of compiled Go code.

The traditional threading facilities available on the JVM are tied to kernel resources, which limits their scalability and the efficiency of context switching. Using the techniques described in this paper, we are able to map millions of processes onto only as many threads as there are hardware threads. We achieve this by a CPS transformation of bytecode produced by the Scala compiler and through the use of a user-level scheduler that supports process migration. The approach described in this paper is entirely portable and can target any JVM. Our runtime system also supports deadlock detection, reporting it to the user should it occur.

We have demonstrated that message-passing concurrency can be realised in both a portable and a pragmatic way. We have done so by leveraging an existing body of work, and extending it to full message-passing concurrency and functional programming while maintaining excellent performance.

Acknowledgements

I would like to thank Gavin Lowe for his valuable feedback on an early draft of this paper. I would also like to acknowledge Sriram Srinivasan for his work on high performance Java Actors whose ideas we have followed, and Bernard Sufrin whose work on CSO lead us to choose Scala as the host language.

References

[1] Matt Gillespie. Preparing for the Second Stage of Multi-Core Hardware: Asymmetric (Heterogeneous) Cores. White paper, Intel Corporation, 2008.
[2] R. M. Ramanathan. Intel® Multi-Core Processors: Making the Move to Quad-Core and Beyond. White paper, Intel Corporation, 2006.
[3] Intel® QuickPath Architecture. White paper, Intel Corporation, 2008.
[4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.
[5] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, December 2003.
[6] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, 11th International Conference, COORDINATION 2009*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009.
[7] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
[8] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[9] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.
[10] András Vajda. *Programming Many-Core Chips*. Springer, 1st edition, 2011.
[11] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 1st edition, July 2007.
[12] Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.
[13] A.W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag, New York, NY, USA, 1st edition, 2010.
[14] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, New York, NY, USA, 1999.
[15] Bernard Sufrin. Communicating Scala Objects. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 35–54, September 2008.
[16] Gavin Lowe. Implementing Generalised Alt. In Peter H. Welch, Adam T. Sampson, Jan Baekgaard Pedersen, Jon Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2011*, pages 1–34, June 2011.
[17] Peter H. Welch, Neil C.C. Brown, James Moores, Kevin Chalmers, and Bernhard H.C. Sputh. Integrating and Extending JCSP. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 349–369, July 2007.
[18] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Alting Barriers: Synchronisation with Choice in Java using JCSP. *Concurrency and Computation: Practice and Experience*, 22(8):1049–1062, June 2010.
[19] Peter H. Welch and Frederick R. M. Barnes. Communicating Mobile Processes: Introducing occam-pi. In Ali Abdallah, Cliff Jones, and Jeff Sanders, editors, *Communicating Sequential Processes. The First 25 Years*, volume 3525 of *Lecture Notes in Computer Science*, pages 712–713. Springer Berlin, April 2005.
[20] Frederick R.M. Barnes, Peter H. Welch, and Adam T. Sampson. Barrier Synchronisation for occam-pi. In Hamid R. Arabnia, editor, *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05)*, pages 173–179, Las Vegas, Nevada, USA, June 2005. CSREA Press.
[21] J. Gosling, B. Joy, G.L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley, 1st edition, 2013.
[22] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima Incorporation, USA, 2nd edition, 2011.
[23] C. Emerick, B. Carper, and C. Grand. *Clojure Programming*. O'Reilly Media, 1st edition, April 2012.
[24] JDK 1.1 for Solaris Developer's Guide. Technical report, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303-4900, U.S.A., February 2000.
[25] Sriram Srinivasan. Kilim: A server framework with lightweight actors, isolation types and zero-copy messaging. Technical report, University of Cambridge, Computer Laboratory, February 2010.
[26] Stefan Fünfrocken. Transparent Migration of Java-based Mobile Agents. In Kurt Rothermel and Fritz Hohl, editors, *Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 26–37. Springer Berlin Heidelberg, 1998.
[27] JavaFlow: Apache Commons project for Java comunications. `http://commons.apache.org/sandbox/commons-javaflow`.
[28] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In *Proceedings of the Third International Conference on Coordination Languages and Models*, COORDINATION '99, pages 211–226, London, UK, 1999. Springer-Verlag.
[29] The Go Programming Language Specification. `http://golang.org/ref/spec`.
[30] Kevin Chalmers, Jon Kerridge, and Imed Romdhani. Mobility in JCSP: New Mobile Channel and Mobile Process Models. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 163–182, July 2007.
[31] G.R. Andrews. *Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 1st edition, 2000.
[32] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.
[33] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing Control in the Presence of One-Shot Continuations. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming Language Design and Implementation*, volume 31 of *PLDI '96*, New York, NY, USA, May 1996. ACM.

[34] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[35] Sriram Srinivasan. A Thread of One's Own. In *Workshop on New Horizons in Compilers*, 2006.

[36] Eric Bruneton. *ASM 4.0: A Java bytecode engineering library*. OW2 Consortium, September 2011.

[37] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley, 1st edition, 2013.

[38] Xavier Leroy. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, August 2003.

[39] Adam T. Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, University of Kent, October 2010.

[40] Kuo-Yi Chen, J.M. Chang, and Ting-Wei Hou. Multithreading in Java: Performance and Scalability on Multicore Systems. *IEEE Transactions on Computers*, 60(11):1521–1534, 2011.

[41] Iulian Dragos and Martin Odersky. Compiling Generics Through User-Directed Type Specialization. In *Proceedings of the Fourth Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 42–47, New York, NY, USA, 2009. ACM.

[42] Andrew Bate and Gavin Lowe. A Debugger for Communicating Scala Objects. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan Baekgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2012*, pages 135–154, August 2012.

[43] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-First Garbage Collection. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[44] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In Jan Vitek, editor, *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, Berlin, Heidelberg, July 2008. Springer-Verlag.

[45] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 268–281, New York, NY, USA, October 2003. ACM.

[46] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. Technical report, RedHat, Inc, 2003.

[47] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from Generalized Stack Inspection. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 216–227, New York, NY, USA, 2005. ACM.

[48] Peter H. Welch and Jeremy M. R. Martin. Formal Analysis of Concurrent Java Systems. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 275–301, September 2000.