Non-Blocking Concurrent Imperative Programming with Session Types

Miguel Silva
Department of Computer Science
University of Porto
Porto, Portugal

Mário Florido
Department of Computer Science
University of Porto
Porto, Portugal

Frank Pfenning
Carnegie Mellon University
Pennsylvania, USA

Concurrent C0 is an imperative programming language in the C family with session-typed message-passing concurrency. The previously proposed semantics implements asynchronous (non-blocking) output; we extend it here with non-blocking input. A key idea is to postpone message reception as much as possible by interpreting receive commands as a request for a message. We implemented our ideas as a translation from a blocking intermediate language to a non-blocking language. Finally, we evaluated our techniques with several benchmark programs and show the results obtained. While the abstract measure of span always decreases (or remains unchanged), only a few of the examples reap a practical benefit.

1 Introduction

A session describes the collective conduct of the components of a concurrent system. Binary sessions focus on the interactions between two of these components, with an inherent concept of duality. A *session type* [7, 8] defines the communication between processes using this notion. Session types enforce conformance to a communication protocol, organizing a session to occur over a communication channel. Recently, session types have been linked with linear logic via a Curry-Howard interpretation of linear propositions as types, proofs as processes, and cut reduction as communication. Variations apply for both intuitionistic [2, 3] and classical [14] linear logic. The intuitionistic variant culminated in SILL, a functional language with session-typed concurrency [13].

The adaptation of SILL to other paradigms of programming gave rise to CLOO (Concurrent Linear Object-Orientation) [1], a concurrent object-oriented language that types objects and channels with session types, and Concurrent C0 [15], a session-based extension to an imperative language. In this paper, we present a non-blocking model of receiving messages in Concurrent C0. Our model will be based on the premise that we only want to halt execution to wait for a message when the data contained in that message is necessary to continue computation. We will then present a compilation function from the original version of Concurrent C0 to our non-blocking model. We also present a cost semantics for Concurrent C0 that computes the span and work [5], providing an abstract analytical measure of latent parallelism in the computation. The span always improves or remains the same under the non-blocking semantics, although the experimental results show that it is difficult to exploit these improvements in practice.

In related work, Guenot [6] has given a pure and elegant computational interpretation of classical linear logic with non-blocking input based on the solos calculus [9]. The primary notion of a process as a thread of control that pervades our work is no longer visible there and it does not immediately suggest an implementation strategy.

Section 2 briefly introduces the Concurrent C0 language, outlining its syntax and operational semantics. Section 3 exposes the model for non-blocking reception of messages. Section 4 defines a translation

Submitted to:
Linearity 2016

from the original to the non-blocking language. Section 5 provides an experimental evaluation of the implementation of the new model, and Section 6 concludes.

2 Imperative Programming with Sessions Types

This section introduces the main concepts of programming of Concurrent C0 (CC0), a session-based extension of C0 [11], itself a small safe subset of C augmented with contracts. For a more in-depth description of CC0, we refer the reader to a technical report [15].

A program in CC0 is a collection of processes exchanging messages through channels. These processes are *spawned* by functions that return channels. The process that calls these spawning functions is called the client. The new process at the other end of the channel is called the provider, who is said to *offer a session* over the channel.

CC0's session typing system is based on work by Caires and Pfenning [2], who establish a correspondence between session types for π-calculus and intuitionistic linear logic. CC0's channels have a linear semantics: there is exactly one reference to the channel besides the provider's. This captures the behavior described in the previous paragraph: the process we called the client has the unique reference to the provided channel.

Messages are sent asynchronously: processes advance in parallel without waiting for acknowledgement of the sent message being received. Programmers must specify the protocol of message exchange using session types and a linear type system enforces concordance with this protocol [12].

```
choice queue {
  <?int; ?choice queue>  Enq;      // receive 'Enq', then int, continue as 'queue'
  <!choice queue_elem>   Deq;      // receive 'Deq', continue as 'queue_elem'
  <!bool; ?choice queue> IsEmpty;  // receive 'IsEmpty', then send bool, continue as 'queue'
  < >                    Dealloc;  // receive 'Dealloc', then terminate
};
choice queue_elem {
  <?choice queue>       None;      // send 'None', continue as 'queue'
  <!int; ?choice queue> Some;      // send 'Some', send int, continue as 'queue'
};
typedef <?choice queue> queue;    // define 'queue' as external choice
```

Program 1: Protocol definition of a queue with constant time enqueue and dequeue operations from client's perspective, in CC0.

Program 1 is an example defining several session types in CC0. We use `<...>` to enclose a session type, `?` to indicate an input and `!` to indicate and output. An empty `< >` represents the end of a session. A *choice* indicates that a label is received (external choice, prefixed by `?`) or that a label is sent (internal choice, prefixed by `!`). Choices have to be explicitly named and declared in CC0 in a syntax inspired by structs in C.

The queue's protocol exchanges messages in two directions, from the client process to the provider process and vice-versa, resulting in each direction being conveyed independently through an external (*queue*) and internal choice (*queue_element*). The session type is given from the provider's perspective and the compiler determines statically if the operations indicated in the protocol are fulfilled in the correct order with the fitting type akin to the channel's session type.

CC0 is compiled to a target language and linked with a runtime system, both written in C, responsible for implementing communication. The compiler checks if messages are being exchanged in the correct order, in agreement with the session type, and enforces linear use of channels.

The session typing of CC0 uses polarized logic, as detailed in [12], to maintain the direction of the communication. Positive polarity indicates that information is streaming from the provider and negative to the provider. A *shift* is used to swap polarities. The runtime system explicitly tracks the polarity of each channel and the compiler infers and inserts the minimal amount of *shifts* into the target language.

CC0 has the usual features of an imperative programming language, such as conditionals, loops, assignments and functions, extended by communication primitives. Below, we present the core communication syntax of the target language. We differentiate a channel from a variable by placing a \$ before the variable name, such as $\$c$.

P,Q	$::=$	$\$c = spawn(P);\ Q$	spawn
	$\mid$	$\$c = \$d$	forward
	$\mid$	$\mathsf{close}(\$c)$	send end and terminate
	$\mid$	$\mathsf{wait}(\$c);\ Q$	receive end
	$\mid$	$\mathsf{send}(\$c,\ e);\ Q$	send data (including channels)
	$\mid$	$x = \mathsf{recv}(\$c);\ Q$	receive data (including channels)
	$\mid$	$\mathsf{send}(\$c,\ \mathit{shift});\ Q$	send shift
	$\mid$	$\mathit{shift} = \mathsf{recv}(\$c);\ Q$	receive shift
	$\mid$	$\$c.lab;\ Q$	send label
	$\mid$	$\mathsf{switch}(\$c)\ \{lab_i \to P_i\}_i$	receive label

2.1 Cost Semantics

Pfenning and Griffith introduced asynchronous communication using polarized logic [12] in the operational semantics of the target language, which is expressed as a *substructural operational semantics* [10], based on *multiset rewriting* [4]. We now extend this operational semantics by assigning a cost to each operation, using the work-span model [5]. We will only count communication costs, ignoring internal computation. Although this may not lead to an entirely realistic measure for the complexity of an algorithm, it will still make for an interesting abstract one. Moreover, in many of our examples communication costs dominate performance.

In the cost semantics we maintain a span s for each executing process which represents the earliest global time (counting only communication steps) at which the process could have reached its current state. Because messages can only be received after they have been sent, each message is tagged with the time at which it is sent, and the recipient takes the maximum between its own span and the span carried by the message. Except for operations using *shifts* or *forwards*, each call to a communication function will increase the span by one unit.

The work is determined individually by each process. As with span, all operations except the ones using *shifts* or *forwards* will increase work by one unit. Although each message also carries the work of the sending process, this work is ignored unless the message is an end or *forward*. In these two cases, the receiving process will add the work carried by the message to its own, to propagate the work of the sending process, which is being terminated.

$$
\begin{array}{lccl}
\text{Configurations} & \Omega & ::= & \cdot \\
 & & \mid & \mathsf{queue}(\$c, q, \$d), \Omega \\
 & & \mid & \mathsf{proc}(\$c, P, s, w), \Omega \\
 & & \mid & \mathsf{cell}(\$c, x, v), \Omega
\end{array}
$$

$$
\begin{array}{lcll}
\text{Queue filled by provider} & \overleftarrow{q} & ::= & \overleftarrow{\cdot} \mid \overleftarrow{(v,s,w)\cdot q} \mid \overleftarrow{(\mathsf{end},s,w)} \mid \overleftarrow{(\mathsf{shift},s,w)} \\
\text{Queue filled by client} & \overrightarrow{q} & ::= & \overrightarrow{(\mathsf{shift},s,w)} \mid \overrightarrow{q\cdot(v,s,w)} \mid \overrightarrow{\cdot}
\end{array}
$$

Configurations describe executing processes, message queues connecting processes (one for each channel), and local storage cells. In our definition, $\mathsf{proc}(\$c,P,s,w)$ is the state of a process executing program P, offering along channel $\$c$, with span s and work w. The message queue is represented by $\mathsf{queue}(\$c,q,\$d)$, which connects processes offering along $\$d$ with a client using $\$c$. The memory cell $\mathsf{cell}(\$c,x,v)$ holds the state of variable x with value v in the process offering along channel $\$c$. For the operational semantics, we use the multiplicative conjunction ($\otimes$) to join processes, queues, and memory cells, and linear implication to express state transition ($\multimap$). Due to space constraints, we show only some representative rules.

$$
\begin{array}{lll}
\mathsf{shift\_s} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \mathsf{send}(\$d, \mathsf{shift})\ ;\ P, s, w) \\
 & & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q \cdot (shift, s, w)}, \$d) \otimes \mathsf{proc}(\$d, P, s, w)\} \\
\mathsf{data\_r} & : & \mathsf{proc}(\$e, x = \mathsf{recv}(\$c)\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(v, s_1, w_1) \cdot q}, \$d) \\
 & & \quad \multimap \{\exists x.\, \mathsf{proc}(\$e, Q, \mathsf{max}(s, s_1)+1, w+1) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{cell}(\$e, x, v)\} \\
\mathsf{close} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \mathsf{close}(\$d), s, w) \\
 & & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q \cdot (\mathsf{end}, s+1, w+1)}, \_)\} \\
\mathsf{wait} & : & \mathsf{proc}(\$e, \mathsf{wait}(\$c)\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(\mathsf{end}, s_1, w_1)}, \_) \\
 & & \quad \multimap \{\mathsf{proc}(\$e, Q, \mathsf{max}(s, s_1)+1, w+w_1+1)\} \\
\mathsf{fwd\_s} & : & \mathsf{queue}(\$c, \overleftarrow{q}, \$d) \otimes \mathsf{proc}(\$d, \$d = \$e, s, w) \\
 & & \quad \multimap \{\mathsf{queue}(\$c, \overleftarrow{q \cdot (\mathsf{fwd}, s, w)}, \$e)\} \\
\mathsf{fwd\_r} & : & \mathsf{proc}(\$d, P(\$c), s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(\mathsf{fwd}, s_1, w_1)}, \$e) \\
 & & \quad \multimap \{\mathsf{proc}(\$d, P(\$e), \mathsf{max}(s, s_1), w+w_1)\} \\
\mathsf{spawn} & : & \mathsf{proc}(\$c, \$d = P()\ ;\ Q, s, w) \\
 & & \quad \multimap \{\exists \$d.\, \mathsf{proc}(\$c, Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{\cdot}, \$d) \otimes \mathsf{proc}(\$d, P, s, 0)\}
\end{array}
$$

3 Non-Blocking Receive

In this section, we present the main contribution of our work: a new model for message reception whose goal will be to block the execution of the process to wait for a message only when the data contained in this message is necessary to continue the execution.

Receiving a message in CC0 blocks the execution of the program, a behavior matching the operational semantics. A receiving function (recv or wait) will only succeed when it is possible to retrieve the corresponding message from the queue of the associated channel.

This model for message reception may not be the optimal choice for some algorithms where an arbitrary imposed order on messages received on two different channels might prevent other computation to proceed. An extreme case is when a received value is not actually ever needed.

Our alternative for follows two principles. One, the difference should be invisible to the programmer who should not need to know exactly when a message is received. Second, the implementation will still need to adhere to the protocol defined by the session type, which forces the order of sends and receives.

3.1 Runtime Implementation

We will now focus on the runtime system, describing how the concepts previously introduced are implemented and how we solved the issues that arose from our model. Our baseline CC0 runtime is implemented in C, using the *pthread* library in a straightforward way to obtain parallelism in multi-core machines. Message passing is implemented using the queues in shared memory.

When a process in CC0 is spawned, it provides a channel to its parent process, i.e., the process that called the spawn function. This channel is represented internally as a structure that aggregates crucial information, namely, a queue of messages.

Our model involves postponing reception as much as possible, by interpreting receives as a request for a message. The request is saved on the channel until a synchronization is necessary. A synchronization is triggered when some data contained on any of the requests is required to continue execution. For example, if a process requested a shift from a channel, a synchronization is required to correct the polarity of the channel, that is, drain the message queue in order to change the direction of communication using the same queue.

The requests are handled in the order they were made, which guarantees that the session type is still being respected. Furthermore, all these changes only occur in an intermediate language, allowing CC0 to keep the same source-level syntax.

From a low level implementation point of view, this change required adding a queue data structure to the channel to hold the requests, creating synchronization functions and modifying the existing receiving functions. It was also necessary to define a translation function between the original, blocking, intermediate language to the new, non-blocking, one. This translation is introduced in the next section. Here are the new constructs introduced by the translation.

$$\begin{array}{llll}
P,Q & ::= & \dots & \\
 & | & \mathsf{async\_wait}(\$c);\ Q & \text{request an end} \\
 & | & x = \mathsf{async\_recv}(\$c);\ Q & \text{request data} \\
 & | & \mathit{shift} = \mathsf{async\_recv}(\$c);\ Q & \text{request a shift} \\
 & | & \mathsf{sync}(\$c,x);\ Q & \text{synchronize variable} \\
 & | & \mathsf{sync}(\$c,\mathsf{shift});\ Q & \text{synchronize shift} \\
 & | & \mathsf{sync}(\$c,\mathsf{end});\ Q & \text{synchronize end}
\end{array}$$

Program 2 shows a possible implementation for the queue example from the previous section. It is presented using an abbreviated version of the target language, using the non-blocking model. Note that receiving a label requires branching between different cases and therefore continues to block in order to avoid speculative execution.

3.2 Cost Semantics

We define a receive request to increase both span and work by one unit. Upon synchronising any request, the span must also be synchronized with the value carried by the message. As before, during the synchronization, any end or fwd message will require the addition of the work contained in the message to the work of the receiving process.

There is also the need to change the definition of Configuration, modifying the queue predicate, which will need to keep a new queue of requests, r, defined below.

$$\begin{array}{llll}
\text{Configurations} & \Omega & ::= & \cdot \\
 & & | & \mathsf{queue}(\$c,q,\$d,r),\Omega \\
 & & | & \mathsf{proc}(\$c,P,s,w),\Omega \\
 & & | & \mathsf{cell}(\$c,x,v),\Omega
\end{array}$$

$$\text{Request queue} \quad r \quad ::= \quad \cdot \mid x \cdot r \mid \mathsf{end} \mid \mathsf{shift}$$

The span of a non-blocking program either decreases or is the same as the corresponding blocking one. For terminating programs, work remains the same between the two models. We have a proof sketch for these two intuitive properties, but a more rigorous proof is still in development. Here, we only show three representative rules.

```
queue $q elem (int x, queue $r) {
  switch ($q) {
    case Enq:
      int y = async_recv($q);
      $r.Enq;
      sync($q, y); send($r, y);
      $q = elem(x, $r);
    case Deq:
      shift = async_recv($q);
      sync($q, shift);
      $q.Some; send($q, x); send($q, shift);
      $q = $r;    // forward request
    case IsEmpty:
      shift = async_recv($q);
      sync($q, shift);
      send($q, false); send($q, shift);
      $q = elem(x, $r);
    case Dealloc:
      shift = async_recv($q);
      $r.Dealloc; send($r, shift);
      async_wait($r);
      sync($r, end); sync($q, shift);
      close($q);
 } }
```

```
queue $q empty () {
  switch ($q) {
    case Enq:
      int y = async_recv($q);
      queue $e = empty();
      sync($q, y);
      $q = elem(y, $e);
    case Deq:
      shift = async_recv($q);
      sync($q, shift);
      $q.None; send($q, shift);
      $q = empty();
    case IsEmpty:
      shift = async_recv($q);
      sync($q, shift);
      send($q, true); send($q, shift);
      $q = empty();
    case Dealloc:
      shift = async_recv($q);
      sync($q, shift);
      close($q);
  }
}
```

Program 2: Implementation of a queue with constant span enqueue and dequeue operations from client's perspective, in CC0's intermediate language using non-blocking input.

$$
\begin{array}{lll}
\mathsf{data\_async\_r} & : & \mathsf{proc}(\$e, x = \mathsf{async\_recv}(\$c)\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r) \\
& & \quad \multimap \{\exists x.\, \mathsf{proc}(\$e, Q, s+1, w+1) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r \cdot x)\} \\
\mathsf{sync\_wait\ 1} & : & \mathsf{proc}(\$e, \mathsf{sync}(\$c,\ \mathsf{end})\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(v, s_1, w_1) \cdot q}, \$d, y \cdot r) \\
& & \quad \multimap \{\mathsf{proc}(\$e, \mathsf{sync}(\$c,\ \mathsf{end})\ ;\ Q, max(s, s_1), w) \otimes \mathsf{queue}(\$c, \overleftarrow{q}, \$d, r) \otimes \mathsf{cell}(\$e, y, v)\} \\
\mathsf{sync\_wait\ 2} & : & \mathsf{proc}(\$e, \mathsf{sync}(\$c,\ \mathsf{end})\ ;\ Q, s, w) \otimes \mathsf{queue}(\$c, \overleftarrow{(\mathsf{end}, s_1, w_1)}, \$d, \mathsf{end}) \\
& & \quad \multimap \{\mathsf{proc}(\$e, Q, max(s, s_1), w + w_1)\}
\end{array}
$$

4 Translation

We represent the translation from a blocking program to a non-blocking one by $[\![\ ]\!]$. It requires an auxiliary table σ of requests, which is used to determine where to include the synchronization functions. This table is represented by a set of pairs, $(\$c, x)$, where $\$c$ is a channel and x is either a variable, an end or a shift. Each process has its own local table. The $[\![\ ]\!]$ function takes a pair $(\mathsf{Instruction}, \sigma)$ and returns another pair carrying the translation of the instruction and a new table. Table 1 presents some examples of the translation.

To translate a receive, there is no synchronization needed, which is a consequence of defining the rule to bind a new variable. The function async_recv produces a request so we have to include a pair in the table of requests, indicating both the variable and the channel tied to the request.

The translations for close and send an expression are similar. To close a channel we need to synchronize it and all its clients, so sync_all generates a list with a number of pairs equal to the number of requests in σ. The auxiliary function generate_sync takes this list and returns the minimum sequence

$$\cdot\ [\![(x = \mathsf{recv}(\$d),\ \sigma)]\!] \quad = \quad (x = \mathsf{async\_recv}(\$d),\ \sigma \cup \{(x, \$d)\})$$

$$\begin{array}{lll}
\cdot\ [\![(\mathsf{close}(\$d),\ \sigma)]\!] & = & (\mathit{sync\_instructions};\ \mathsf{close}(\$d),\ \{\}) \\
 & & \quad \text{where} \quad l = \mathsf{sync\_all}\ \sigma \\
 & & \qquad\qquad (\mathit{sync\_instructions}, \sigma') = \mathsf{generate\_sync}\ l\ \sigma \\
\cdot\ [\![(\mathsf{send}(\$d, e),\ \sigma)]\!] & = & (\mathit{sync\_instructions};\ \mathsf{send}(\$d, e),\ \sigma') \\
 & & \quad \text{where} \quad l_1 = \mathsf{check\_shift}\ \sigma\ \$d \\
 & & \qquad\qquad l_2 = \mathsf{check\_exp}\ \sigma\ e \\
 & & \qquad\qquad (\mathit{sync\_instructions}, \sigma') = \mathsf{generate\_sync}\ (l_1 \cup l_2)\ \sigma
\end{array}$$

Table 1: Translation scheme of recv-value, close and send-expression operations.

of instructions needed to synchronize the channel. If there is no need to perform any synchronization, generate_sync produces a no operation instruction.

To send an expression, we need to perform two different checks: if the channel has the correct polarity and if there is any variable in the expression that needs to be synchronized. The auxiliary functions check_shift and check_exp handle these two verifications, respectively. They return a list of pairs containing the variables and channels that require synchronisation, lists that are passed on to generate_sync, which produces the synchronisation instructions, as mentioned previously. This function also removes all the synchronized pairs from the table of requests.

5 Experimental Evaluation

All benchmarks were run on a 2015 Macbook Pro, with a 2.7 GHz Intel Core I5 (2 cores) processor and 8 GB RAM. The benchmarking suite[1] can be found in table 2, adapted from [15].

bitstring1	bitstrings with external choice	parfib	parallel naive Fibonacci, simulating fork/join
bitstring3	bitstrings with internal choice	primes	prime sieve (sequential)
bst	binary search trees, tree sort	queue-notail	queues without tail calls
insert-sort	insertion sort using a queue	queue	queues written naturally
mergesort1	mergesort with Fibonacci trees	reduce	reduce and scan on parallel sequences
mergesort3	mergesort with binary trees	seg	list segments
mergesort4	mergesort with binary trees, sequential merge	sieve-eager	eager prime sieve merge
odd-even-sort1	odd/even sort, v1	sieve-lazy	lazy prime sieve
odd-even-sort4	odd/even sort, v4	stack	a simple stack
odd-even-sort6	odd/even sort, v6		

Table 2: CC0 benchmarking suite

[1]available at `http://www.cs.cmu.edu/~fp/misc/cc0-bench.tgz`

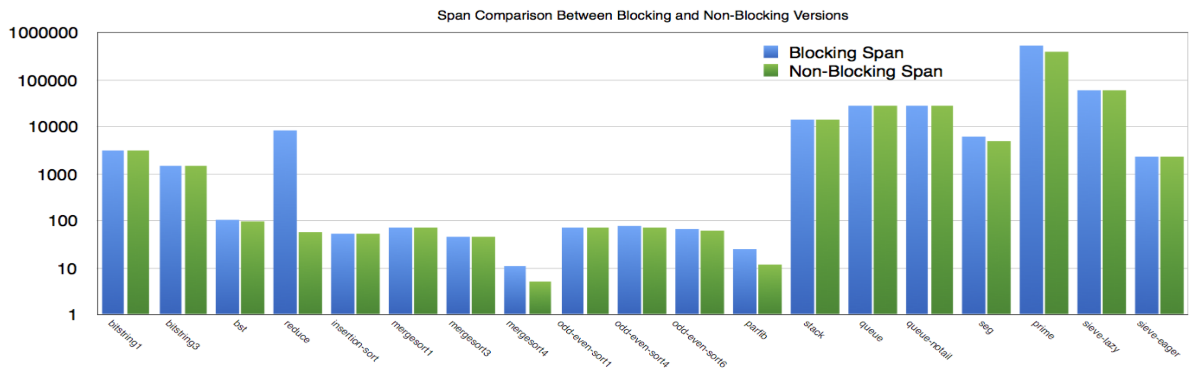

Figure 1: Blocking and Non-Blocking span benchmarks on a log scale.

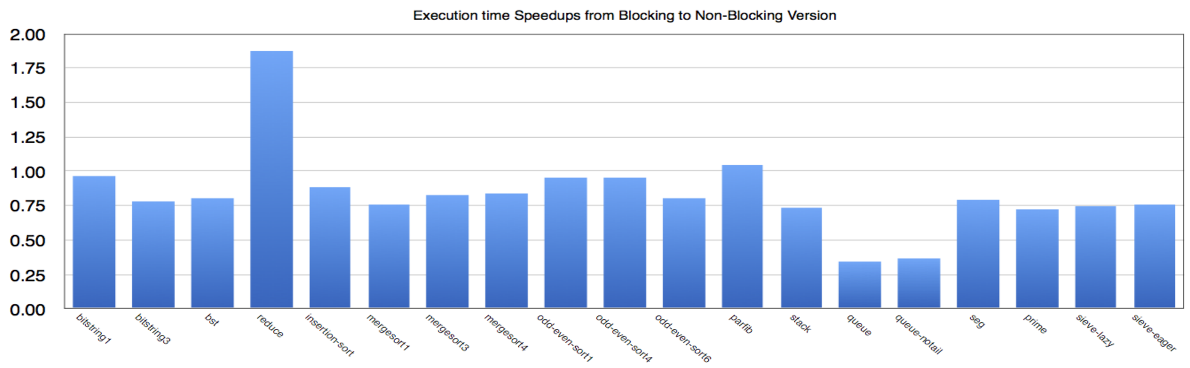

Figure 2: Execution time speedup from Blocking to Non-Blocking Version.

Note that the span is often the same across our benchmark suite, where reduce, mergesort4, and parfib see noticeable improvements. In the case of reduce, some of the difference could be recovered by fine-tuning the CC0 source program; it is interesting that our generic technique can make up for a performance bug (when considered under the blocking semantics) introduced by the programmer. This shows that, at the very least, our implementation can help identify some performance issues in the given code. As figure 2 shows, the overhead of maintaining the request queue is considerable in some examples. Only in reduce and parfib do we realize an actual performance improvement.

6 Future Work

Since recipient behavior on input is opaque to the sender, we may be able to craft an optimisation which avoids the slowdown in the common case where no improvement in the span is available. For this purpose we would likely combine our fully dynamic technique with static dependency analysis to use non-blocking input only where promising. Our cost semantics could also become a tool in performance debugging which may be helpful in particular to novice programmers with little experience in concurrency.

References

[1] Stephanie Balzer & Frank Pfenning (2015): *Objects As Session-typed Processes*. In: *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, ACM, New York, NY, USA, pp. 13–24, doi:10/bd3b.

[2] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In: *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, Springer LNCS 6269, Paris, France, pp. 222–236.

[3] Luís Caires, Frank Pfenning & Bernardo Toninho (2016): *Linear logic propositions as session types*. *Mathematical Structures in Computer Science* 26, pp. 367–423, doi:10/bd27.

[4] Iliano Cervesato & Andre Scedrov (2009): *Relating state-based and process-based concurrency through linear logic (full-version)*. *Information and Computation* 207(10), pp. 1044 – 1077, doi:10/bgr46k. Special issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006).

[5] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest & Charles E. Leiserson (2001): *Introduction to Algorithms*, 2nd edition, chapter 27. McGraw-Hill Higher Education.

[6] Nicolas Guenot (2014): *Session Types, Solos, and the Computational Contents of the Sequent Calculus*. Talk at the Types Meeting.

[7] Kohei Honda (1993): *Types for Dyadic Interaction*. In: *4th International Conference on Concurrency Theory*, CONCUR'93, Springer LNCS 715, pp. 509–523.

[8] Kohei. Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *7th European Symposium on Programming Languages and Systems*, ESOP'98, Springer LNCS 1381, pp. 122–138.

[9] Cosimo Laneve & Björn Victor (2003): *Solos in Concert*. *Mathematical Structures in Computer Science* 13(5), pp. 657–683.

[10] F. Pfenning & R. J. Simmons (2009): *Substructural Operational Semantics as Ordered Logic Programming*. In: *Logic In Computer Science, 2009. LICS '09. 24th Annual IEEE Symposium on*, pp. 101–110, doi:10.1109/LICS.2009.8.

[11] Frank Pfenning & Rob Arnold: *C0 Language*. Available at `http://c0.typesafety.net`.

[12] Frank Pfenning & Dennis Griffith (2015): *Polarized Substructural Session Types*. In A. Pitts, editor: *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, Springer LNCS 9034, London, England, pp. 3–22. Invited talk.

[13] Bernardo Toninho, Luis Caires & Frank Pfenning (2013): *Higher-Order Processes, Functions, and Sessions: A Monadic Integration*. In: *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, Springer-Verlag, Berlin, Heidelberg, pp. 350–369, doi:10/bd29.

[14] Philip Wadler (2015): *Propositions As Types*. *Commun. ACM* 58(12), pp. 75–84, doi:10/bd28.

[15] Max Willsey, Rokhini Prabhu & Frank Pfenning: *Design and Implementation of Concurrent C0*. Available at `http://maxwillsey.com/assets/cc0-paper.pdf`.