When Concurrency Matters: Behaviour-Oriented Concurrency

LUKE CHEESEMAN, Imperial College London, UK
MATTHEW J. PARKINSON, Microsoft Azure Research, UK
SYLVAN CLEBSCH, Microsoft Azure Research, UK
MARIOS KOGIAS, Imperial College London, UK Microsoft Research, UK
SOPHIA DROSSOPOULOU, Imperial College London, UK
DAVID CHISNALL, Microsoft, UK
TOBIAS WRIGSTAD, Uppsala University, Sweden
PAUL LIÉTAR, Imperial College London, UK

Expressing parallelism and coordination is central for modern concurrent programming. Many mechanisms exist for expressing both parallelism and coordination. However, the design decisions for these two mechanisms are tightly intertwined. We believe that the interdependence of these two mechanisms should be recognised and achieved through a single, powerful primitive. We are not the first to realise this: the prime example is actor model programming, where parallelism arises through fine-grained decomposition of a program's state into actors that are able to execute independently in parallel. However, actor model programming has a serious pain point: updating multiple actors as a single atomic operation is a challenging task.

We address this pain point by introducing a new concurrency paradigm: Behaviour-Oriented Concurrency (BoC). In BoC, we are revisiting the fundamental concept of a behaviour to provide a more transactional concurrency model. BoC enables asynchronously creating atomic and ordered units of work with exclusive access to a collection of independent resources.

In this paper, we describe BoC informally in terms of examples, which demonstrate the advantages of exclusive access to several independent resources, as well as the need for ordering. We define it through a formal model. We demonstrate its practicality by implementing a C++ runtime. We argue its applicability through the Savina benchmark suite: benchmarks in this suite can be more compactly represented using BoC in place of Actors, and we observe comparable, if not better, performance.

CCS Concepts: • **Computing methodologies** → **Concurrent programming languages**; **Parallel programming languages**; • **Theory of computation** → **Parallel computing models**.

Additional Key Words and Phrases: actors

ACM Reference Format:
Luke Cheeseman, Matthew J. Parkinson, Sylvan Clebsch, Marios Kogias, Sophia Drossopoulou, David Chisnall, Tobias Wrigstad, and Paul Liétar. 2023. When Concurrency Matters: Behaviour-Oriented Concurrency. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 276 (October 2023), 30 pages. https://doi.org/10.1145/3622852

Authors' addresses: Luke Cheeseman, luke.cheeseman12@imperial.ac.uk, Imperial College London, UK; Matthew J. Parkinson, mattpark@microsoft.com, Microsoft Azure Research, UK; Sylvan Clebsch, sylvan.clebsch@microsoft.com, Microsoft Azure Research, UK; Marios Kogias, m.kogias@imperial.ac.uk, Imperial College London, UK and Microsoft Research, UK; Sophia Drossopoulou, s.drossopoulou@imperial.ac.uk, Imperial College London, UK; David Chisnall, David.Chisnall@cl.cam.ac.uk, Microsoft, UK; Tobias Wrigstad, tobias.wrigstad@it.uu.se, Uppsala University, Sweden; Paul Liétar, paul.lietar13@imperial.ac.uk, Imperial College London, UK.

2475-1421/2023/10-ART276
https://doi.org/10.1145/3622852

1 INTRODUCTION

In a world that demands faster and more efficient computing, the need for concurrent programming has become paramount. Concurrent programming expresses the asynchronous behaviour that arises naturally in systems (*e.g.,* process an incoming request) and harnesses the parallel compute power of modern hardware, where hardware thread counts can soar into the hundreds and beyond [Bergman et al. 2008].

Expressing parallelism and coordination is central for modern concurrent programming. Parallelism empowers us to perform multiple tasks simultaneously, while coordination provides us with the control necessary to exclude the concurrent schedules that do not meet our desired outcomes. Several mechanisms exist for expressing both parallelism and coordination.

The main abstraction for the level of parallelism is a *thread* and its different instantiations, *e.g.,* kernel threads, user-level threads, coroutines, tasks, fork/join, etc. These parallelism mechanisms typically provide a minimal coordination in the form of waiting for termination (*e.g.,* join, promises). This is sufficient for problems that are easy to parallelise which are typically structured (*e.g.,* McCool et al. [2012]) with up-front knowledge of the data needed to perform a task; the key to efficient parallelism is partitioning data into isolated (ideally equi-sized) chunks to be processed individually.

Other classes of problems such as concurrent event handling or serving requests typically needs scheduling a mix of short and long-running events and *coordinating* accesses to data according to an unforseeable schedule [Kegel 2014; WhatsApp 2012]. Here the coordination mechanism are required to be more elaborate (*e.g.,* locks, transactions, condition variables).

Whilst decoupling parallelism and coordination provides flexibility, their design decisions are tightly intertwined. For an async runtime, it is wise to provide bespoke synchronisation primitives, rather than relying on standard locking which will block one of the underlying implementation threads and harm performance (*e.g.,* Tokio in Rust provides a Lock primitive). In the pursuit of performance, increasing the thread count may improve parallelism, but for codebases with coarse-grained locking, more threads may harm scalability by racing for the same resources.

Rather than decoupling parallelism and coordination, we should recognise their interdependence and achieve both through a single, powerful primitive. We are not the first to realise this, the prime example is actor model programming. In the actor model [Agha 1985], parallelism arises through fine-grained decomposition of a program's state into actors that are able to execute independently in parallel, regardless of whether they serve different requests, process different sub-problems to be joined together, or a mix. A key feature of the actor model is that each actor isolates its own state. This enables sequential reasoning inside an actor, but this is arguably also its Achilles' heel: poor support for operations that involve accessing the states of multiple actors. For this reason actor systems mix the actor model with other concurrency paradigms (giving potential for programmers to break the actor model) [Tasharofi et al. 2013], or invent complicated bespoke coordination mechanisms on-top of the underlying model.

In this paper we explore and extend the idea of coupling parallelism and coordination. We propose a programming model that we call *behaviour-oriented concurrency* (BoC). The BoC programming model relies on a decomposition of state that is akin to actors—a program's state is a set of isolated resources (that we call *cowns*). Behaviours are asynchronous units of work that explicitly state their required set of resources. Unlike messages in actor programming, behaviours are not coupled to a specific resource; crucially, BoC offers *flexible coordination*, operations that require synchronous access to multiple resources can be easily expressed.

To construct those behaviours, we introduce a new keyword, `when`, which enumerates the set of necessary resources for the said behaviour and spawns this an asynchronous unit of compute. So, a BoC program is a collection of behaviours that each acquires zero or more resources, performs

computation on them, which typically involves reading and updating them, and spawning new behaviours, before releasing the resources. When running, a behaviour has no other state than the resources it acquired. A behaviour can be run when it has acquired all of its resources, which is guaranteed to be deadlock free. Despite its apparent simplicity, this model has considerable expressive power to construct a wide range of concurrent schedules by simply nesting them and/or sequencing them in combination with the resources they require. This is important in itself as making it effortless to spawn new concurrent computation is key to writing programs that are able to scale with the parallel compute power of modern hardware, without being crippled by Amdahl's law 1967.

We introduce a formal model of the new paradigm which demonstrates how BoC can introduce concurrency to a programming language. We also lay out the expectations of a programming language, such as separation of heap, so that the language is able to fully utilise BoC.

To demonstrate that BoC is a practical approach to concurrency across different languages, we have created a C++ library that functions as a runtime for BoC and an executable model implementation in C#. Both runtimes are closely aligned with the formal model. Whilst we aim to integrate our C++ runtime as a fundamental component of a programming language, this library provides insights into the runtimes performance.

We evaluate both the expressiveness of BoC as a concurrency paradigm and the efficiency of our implementation. We present BoC implementations of the Savina actor benchmark suite [Imam and Sarkar 2014], first using BoC as if it were an actor language, and then better utilising BoC to demonstrate where it can be a better fit than actors.

In this paper, we make the following contributions:

- The Behaviour-Oriented Concurrency paradigm: a concurrency paradigm that achieves flexible coordination over multiple resources, and ordered execution, and scalability.
- A formal model for BoC which demonstrates how it can provide concurrency for an underlying programming language.
- An efficient proof of concept implementation of BoC in C++ that achieves almost perfect scaling for the conducted system-level experiments.

The rest of this paper is organised as follows: In Section 2 we give introduction to behaviour-oriented concurrency informally and illustrate it through examples. In Section 3 we formalise the informal definition of BoC by providing a simplified abstract execution model and show how the guarantees are met. In Section 4 we develop an implementation for BoC. In Section 5 we evaluate BoC concurrency with the Savina benchmark suite. In Sections 6 and 7 we conclude and discuss related and further work.

2 BEHAVIOUR-ORIENTED CONCURRENCY – OVERVIEW

Behaviour-oriented concurrency (or BoC) is intended as the sole concurrency feature of an underlying programming language. The underlying language is expected to provide a means to separate the heap into disjoint sets with unique entry points, for example through a type system as in [Clarke and Wrigstad 2003]. Provided that the type system satisfies the requirements outlined here and in the semantics section, its exact design is orthogonal to BoC. Several such type systems exist already [Clebsch et al. 2015; Gordon et al. 2012; Noble et al. 1998]. In [Arvidsson et al. 2023] we propose a new, more powerful type system which we are adopting for our language implementing the BoC paradigm.

2.1 BoC in a Nutshell

BoC augments the underlying language with two fundamental concepts: the *concurrent owner* or *cown*, and the *behaviour*.

Cowns a cown protects a piece of separated data, meaning it provides the only entry point to that data in the program. A cown is in one of two states: *available*, or *acquired* by a behaviour.

Behaviours are the unit of concurrent execution. They are *spawned* with a list of required cowns and a closure. We define *happens before*, as an ordering relation which strengthens the "spawned before" relation by also requiring that the sets of cowns required by the two behaviours have a non-empty intersection:

Definition 2.1. A behaviour b will *happen before* another behaviour b' iff b and b' require overlapping sets of cowns, and b is spawned before b'.

Once spawned, a behaviour can be *run* (*i.e.,* its closure starts executing) only when all its required cowns are available, and all other behaviours which *happen before* it have been run. Once available, all the cowns are acquired by the behaviour atomically, and become unavailable to other behaviours. Throughout execution of the closure the behaviour retains exclusive access to its cowns, and to their data. Moreover, the behaviour cannot acquire more cowns, nor can it prematurely release any cown it is holding. Upon termination of the closure, the behaviour terminates, and all its cowns become available.

BoC is datarace-free and deadlock-free. BoC provides the principles to ensure data-race freedom: since the state associated with each cown is isolated, and since behaviours have exclusive access to the acquired cowns' state, there is no way for two different behaviours to have concurrent mutable access to the same state. Furthermore, as behaviours acquire *all* their required cowns atomically, and because the happens before relation is acyclic, BoC is *deadlock-free* by construction.

We introduce BoC through examples expressed in an imperative, expression-oriented, strongly-typed pseudo-language with isolation. Two language features are of interest:

- The `cown[T]` type: it represents a cown with contents of type `T`, with a `create` constructor.
- The **`when`** expression: it consists of a set of cown identifiers, and a closure, and is used to spawn a behaviour that requires the specified cowns.

It is the remit of the underlying language to ensure that each object in the heap is owned by exactly one entry point, and that any object accessed within the closure of a **`when`** is owned by one of the acquired cowns.

2.2 Creating Cowns

For our examples we focus on concurrent modifications to bank account objects of type `Account` with fields `balance` and `frozen`. Listing 1 shows how a programmer can create a cown to protect an account. The programmer creates and accesses `acc1` as an `Account` object; they can then add and remove funds from `acc1` as they want (synchronously). Conversely, `acc2` is created as a cown that protects an `Account`; in other words, a cown of type **`cown`**`[Account]`. The cown prevents any direct access to the account, such as that on Line 6. The only way to gain access to the contents of a cown is by spawning a behaviour that acquires it.

List. 1. Creating cowns to protect data

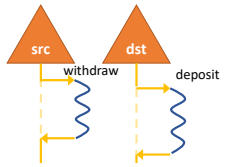
```
1 main() {
2   var acc1 = Account.create();
3   acc1.balance -= 100; // OK
4
5   var acc2 = cown.create(Account.create());
6   acc2.balance += 100 // Access is invalid
7 }
```

2.3 Spawning and Running Behaviours

Listing 2 contains a simple use-case of behaviours. The `transfer` function transfers `amount` from `src` to `dst`. It consists of two **`when`** expressions, which spawn behaviours that need mutable access to `Account` objects. Note that while a **`when`** always requires resources to be of type **`cown`**`[T]` the closure sees the resources as being of type `T`, *i.e.,* from the *inside*, an acquired **`cown`**`[T]` looks like a `T`. Each **`when`** expression spawns a behaviour that will, at a later point, execute with exclusive access to the contents of its cown; thus it will execute without data races, and logically atomically. The `transfer` function returns immediately, without waiting for the two behaviours to execute.

List. 2. Scheduling work on each account

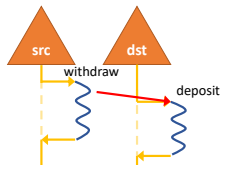
```
1 transfer(src: cown[Account], dst: cown[Account], amount: U64) {
2   when (src) { src.balance -= amount; }; // withdraw
3   when (dst) { dst.balance += amount; } // deposit
4 }
```



Note that spawning and running a behaviour are different events; moreover, **spawning is synchronous**, while **running is asynchronous**. Assuming that `src` and `dst` are not aliases, they can start running in any order: the `withdraw` behaviour might happen before, at the same time as, or after the `deposit` behaviour. In Listing 2, each behaviour is executed independently and unconditionally. The right of Listing 2 shows a possible execution timeline: each cown (triangles) transitions from available (solid lines) to acquired (dashed lines) as it is acquired by an executing behaviour (squiggly lines), and transitions back again once the behaviour terminates; as these behaviours are independent they may execute together or one at a time (and so may move up and down the timeline). That is, the withdraw behaviour might happen before, at the same time as, or after the deposit behaviour.

List. 3. Nesting spawning behaviours

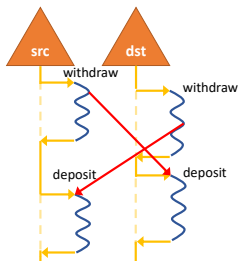
```
1 transfer(src: cown[Account], dst: cown[Account], amount: U64) {
2   when (src) { // withdraw
3     if (src.balance >= amount) {
4       src.balance -= amount;
5       when (dst) { // deposit
6         dst.balance += amount;
7 } } } }
```



But what if successful transfer is contingent on properties of the `src` cown? Consider the case where accounts may not be overdrawn, *i.e.,* the `balance` has to be positive. This can be achieved by nesting the second **`when`** expression inside a conditional path in the first, as demonstrated in Listing 3. Here, the function `transfer` merely spawns one behaviour to be executed on the `src` cown and returns immediately. When this outer behaviour executes, it will check that the `src` has sufficient funds. If it does, a new behaviour will be spawned to run on the `dst` cown. The outer behaviour terminates immediately, without waiting for the inner one to start nor finish executing. To be clear, **the deposit does *not* have access to `src`**. On the right of Listing 3, the withdrawing behaviour is shown to spawn the depositing behaviour (red arrow), the spawned behaviour can be run anytime after it was spawned (incidentally, the spawned behaviour can move along the timeline as long as the arrow does not

List. 4. Deadlock-free transfers

```
1 transfer(src, dst, 1);
2 transfer(dst, src, 2);
```



point upwards), the depositing behaviour may in fact terminate before the withdrawing behaviour terminates. So, whilst the depositing behaviour is textually nested within the withdrawing behaviour, nested **when**s have the same semantics as any other **when**s. Contrast this with transactions where nested transactions have special semantics [Ni et al. 2007].

We extend the example from above to two calls of `transfer`: one from `src` to `dst`, and one from `dst` to `src`, illustrated in Listing 4. Readers accustomed to lock-based programming or nested transactions might think that the example would deadlock, but this is not so: the two `withdraw` behaviours may run in parallel (since they require different cowns), and during their execution they will spawn the `deposit` behaviours. Each `deposit` behaviour will be able to run once its cown is no longer held by the `withdraw` behaviour. Moreover, the two withdraws can happen in either order as can the two deposits. We will revisit this example in Section 3.

2.4 Behaviours Requiring Multiple Cowns

Listing 3 avoids transfers with insufficient funds, but what if successful transfer is contingent on properties of *both* cowns? For instance, a bank account may be marked as "frozen", meaning any transfer to or from the account should be forbidden.

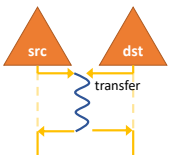
We could try to solve the problem by checking each account in turn, withdraw the money from the `src` and finally deposit the money on the `dst`, requiring further nesting of behaviours. However, by doing so we would lose atomicity guarantees: between checking the account and depositing the money, the `dst` could be frozen by another behaviour spawned by a different part of the program.

Instead, we will employ a key feature of our paradigm which allows behaviours to acquire multiple cowns at once, thus giving *simultaneous* exclusive mutable access to several cowns.

In Listing 5, the function `transfer` avoids the earlier atomicity problems. It consists of a **when** block that requires access to `src` as well as `dst`. The **when** checks that the accounts are not frozen and that there are sufficient funds, and if so, proceeds with transferring the funds.

List. 5. Spawn a behaviour that requires both accounts

```
1 transfer(src: cown[Account], dst: cown[Account], amount: U64) {
2   when (src, dst) { // withdraw and deposit
3     if (src.balance >= amount && !src.frozen && !dst.frozen) {
4       src.balance -= amount;
5       dst.balance += amount;
6 } } }
```



This example demonstrates the power of behaviour-oriented concurrency: It allows the programmer to coordinate access to multiple shared resources. A rendezvous of cowns is now a simple task: a single **when** expression creates the synchronisation between multiple cowns and allows a behaviour to read/write the state of all involved cowns. This rendezvous is demonstrated in to the right of Listing 5 by a behaviour that acquires both `src` and `dst` at once.

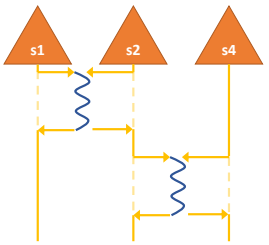
2.5 Order Matters

So far, we assumed that `src` and `dst` do not alias. For behaviours requiring non-overlapping cowns, the order of execution does not matter. Namely, the effects of behaviour execution demonstrate themselves through modification of the state under their cowns; therefore, cowns cannot observe the effects of execution of behaviours that took place on separate cowns.

However, when behaviours execute on overlapping cowns, the order matters, and BoC has rules for such cases. For illustration, consider the following scenario: we want to transfer money from one account to another, and then use the money from the second account to pay a third party. This is done in Listing 6. The first transfer will be spawned before the second transfer, but what we have discussed so far does not preclude the second example from running first. Here the order of execution of the behaviours matters: e.g., if the second behaviour runs first, it could deplete the funds in `s2`, and incur overdraft fees.

List. 6. Statically Ordered

```
1 transfer(s1, s2, 10);
2 transfer(s2, s4, 20);
```

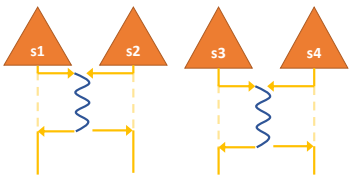


Thus, we would like the first transfer to be completed before the second transfer starts running. We could transform the program to achieve the required ordering, for example by using nested behaviours, but there is no need: BoC guarantees the desired order of behaviours. Namely, a behaviour will only be run once all behaviours that must *happen before* it have been run (Definition 2.1).

In Listing 6, the first transfer requires cowns `s1` and `s2`, the second transfer requires cowns `s2` and `s4`, and {`s1`, `s2`} overlaps with {`s2`, `s4`}. As we said earlier, the first transfer is spawned before the second transfer. Therefore, the first transfer will happen before the second transfer.

List. 7. Dynamically Ordered

```
1 transfer(s1, s2, 10);
2 transfer(s3, s4, 20);
```



Behaviour ordering is determined at runtime. Consider the slightly modified example from Listing 7, where we have four different cown identifiers, `s1`, `s2`, `s3`, `s4`. If `s2` and `s3` are aliases, then the situation is as in Listing 6, and the first transfer will complete before the second starts running. But if the four identifiers point to four different cowns, then execution of the first and second transfer may take place in any order, and may overlap. Below Listing 7 we show diagrammatically a possible execution, when `s1`, `s2`, `s3`, `s4` differ.

2.6 Putting It All Together

Revisiting our example from section Section 2.3, in Listing 2, if `src` and `dst` are aliases, then withdraw will complete before deposit. In Listing 8, green arrows represent happens before relations.

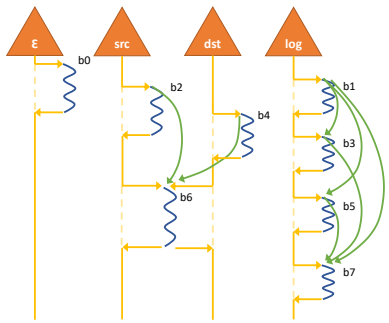
This ordering is a deep property that can be used to order nested spawned behaviours. Consider a scenario where, in addition to account operations, we also wanted to create a log of what happened; we augment each behaviour to output strings to an `OutStream` as in Listing 8. Notice that $b2$ and $b6$ spawn further behaviours, $b3$ and $b7$, both of which require `log`. Again, we expect the start message, on Line 3, to be logged before the deposit message, on Line 5, and the deposit message before the transfer message on Line 11.

The `"begin"` message will be logged before the `"deposit"` message: $b1$ and $b3$ require `log` and $b1$ will be spawned before $b3$ (we know this as `b1` will be spawned before $b2$, and $b2$ spawns $b3$); thus, we know $b1$ will happen before $b3$ from Definition 2.1.

The `"deposit"` message will be logged before the `"transfer"` message: $b3$ and $b7$ require `log`; we know $b2$ spawns $b3$ and $b6$ spawns $b7$, and we know $b2$ happens before $b6$ from previous discussion; therefore, we know $b3$ will be spawned before $b7$ and so $b3$ will happen before $b7$.

List. 8. Creating an accurate log

```
1  main(src: cown[Account], dst: cown[Account],
2       log: cown[OutStream]) { /* b0 */
3    when(log) { /* b1 */ log.log("begin") }
4    when(src) { /* b2 */ ...
5      when(log) { /* b3 */ log.log("deposit") }
6    }
7    when(dst) { /* b4 */ ...
8      when(log) { /* b5 */ log.log("freeze") }
9    }
10   when(src, dst) { /* b6 */ ...
11     when(log) { /* b7 */ log.log("transfer") }
12 } }
```



These constraints are presented in the execution timeline in the right of Listing 8: this depicts which behaviours must happen before which others (green arrows), we can see for example that $b2$ and $b4$ can happen in either order, but both must happen before $b6$, and thus $b3$ and $b5$ happen before $b7$. Yet, there is no happens before order between $b3$ and $b5$ and so these behaviours can run in either order.

2.7 Cost of Order

We showed how a desired order can be obtained by construction of a program, but what about when order is not desired? Consider a very simplified version of the Dining Philosophers with 4 forks and 4 philosophers each trying to eat once. If we scheduled them in sequential order, then we would fully sequentialise the program. The overlapping cown sets in Listing 9 forces all the operations into a single linear order, whereas Listing 10 enforces that $b1$ and $b3$ must occur before $b2$ and $b4$, but no other constraints. Thus, the second program can execute two things in parallel, whereas first can only execute one. We can view the Philosophers problem as a generalisation of this pattern.

List. 9. Sequential scheduling

```
1 when (f1, f2) { /* b1 */ }
2 when (f2, f3) { /* b2 */ }
3 when (f3, f4) { /* b3 */ }
4 when (f4, f1) { /* b4 */ }
```

List. 10. Alternating scheduling

```
1 when (f1, f2) { /* b1 */ }
2 when (f3, f4) { /* b3 */ }
3 when (f2, f3) { /* b2 */ }
4 when (f4, f1) { /* b4 */ }
```

It is important to note that this degenerate case affects *performance* but not *correctness*. In contrast, the failure modes of a similar error in a lock-based implementation are either data races or deadlock. Implementations using transactional memory would have similar problems, with adjacent philosophers attempting to mutate the same state in a transaction and then rolling back and hitting a slow-path with guaranteed ordering. BoC provides an advantage in that the error causing the performance problem is both observable and fixable in the source language. The happens-before ordering is part of the source-level semantics and so can be broken with source-level constructs, such as the interleaved ordering presented above. We believe that this combination of properties—that failure modes affect performance rather than correctness and that the programmer can reason about performance problems at the source-language level—are key benefits to the BoC model.

2.8 How Does BoC Measure Up?

Returning to our desired research direction from Section 1: BoC provides a single powerful abstraction for parallelism and *flexible* coordination through the concept of **when**. Like Actors, a BoC

program can saturate a system with behaviours which can be run in parallel, but, unlike Actors, BoC programs can flexibly coordinate access such that behaviours access multiple cowns. Moreover, BoC guarantees an *ordering on the execution* of behaviours which enables a runtime to *implicitly parallelise* behaviours as long as the order is respected.

3 SEMANTICS

Having introduced BoC informally, and illustrated it through examples, we move on to give a formal model to refine the understanding of how a BoC program executes. We will build on this model in Section 4 to demonstrate how BoC can be implemented.

BoC can be built on top of any programming language with the necessary building blocks. Thus, our model is parametric in an underlying programming language. In Definition 3.1 we define the building blocks upon which BoC can be constructed. Note, few constraints are placed on *Context* and *Heap* to allow for many implementations of an underlying language (which is expected to instantiate their structure).

Definition 3.1 (Simple underlying programming language). A tuple $(Context, Heap, \hookrightarrow, finished)$ is an *underlying programming language* if all of the points that follow hold. We use the identifiers $E, E', \ldots$ to range over elements of *Context*, and $h, h' \ldots$ for *Heap*.

(1) The evaluation relation $\hookrightarrow$ has signature $\hookrightarrow \subseteq (Context \times Heap) \to (Context \times Heap)$.

(2) The set $finished \subseteq Context$ describes terminal contexts.

In Definition 3.2 we show how we can extend any underlying language to obtain a BoC language. The extension enriches the underlying language so that programmers can *create* cowns and *spawn* new behaviours; also, the extension introduces concurrency to the language by enabling *running* multiple behaviours at a time.

This extension requires that the underlying language provides cown identifiers, *Tag*, and the behaviour spawn evaluation relation, $\hookrightarrow_{\mathsf{when}\ (\overline{\kappa})\ \{E\}}$.

Definition 3.2 (BoC extensions). We define the Behaviour-Oriented Concurrency (BoC) extension for an underlying language, $(Context_u, Heap_u, \hookrightarrow, finished_u)$, and obtain a BoC tuple $(Context, Heap, \hookrightarrow, finished, \rightsquigarrow, Tag)$:

(1) We expect the underlying language to be extended to accomodate $\kappa \in Tag$ (and $\overline{\kappa}$ is a sequence of $\overline{\kappa}$)

(2) We require the evaluation relation to be extended to accomodate $\hookrightarrow_{\mathsf{when}\ (\overline{\kappa})\ \{E\}}$ where $\hookrightarrow_{\mathsf{when}\ (\overline{\kappa})\ \{E\}} \subseteq Context \to Context$

(3) Configurations are tuples, $Conf = PendingBehaviours \times RunningBehaviours$ where

$$P \in PendingBehaviours = (Tag^* \times Context)^*$$
$$R \in RunningBehaviours = \mathcal{P}(Tag^* \times Context)$$

(4) The evaluation relation $\rightsquigarrow \subseteq (Conf \times Heap) \to (Conf \times Heap)$ is defined in Figure 1.

Note that running behaviours form a set (thus supporting arbitrary interleaved evaluation), while the pending behaviours are a sequence (thus supporting behaviour ordering). Also, we highlight in green, behaviours that are running, and in purple, those that are pending.

Pending behaviours. The linear structure of pending behaviours creates a total order over the spawned behaviour. The reader may be concerned that this restricts the parallelism in BoC, however, we will see that we can remove this total order in Section 4 for an efficient implementation.

We will now discuss each of the rules in Figure 1.

$$\textsc{step}\ \frac{E, h \hookrightarrow E', h'}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R \uplus (\overline{\kappa}, E'), P, h'} \qquad \textsc{spawn}\ \frac{E \hookrightarrow_{\text{when } (\overline{\kappa'})\ \{E''\}} E'}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R \uplus (\overline{\kappa}, E'), P : (\overline{\kappa'}, E''), h}$$

$$\textsc{run}\ \frac{\left(\bigcup_{(\overline{\kappa'}, \_) \in (P' \cup R)} \overline{\kappa'}\right) \cap \overline{\kappa} = \varnothing}{R, P' : (\overline{\kappa}, E) : P'', h \rightsquigarrow R \uplus (\overline{\kappa}, E), P' : P'', h} \qquad \textsc{end}\ \frac{\mathit{finished}(E)}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R, P, h}$$

Fig. 1. Semantics for BoC

STEP describes a step where a running behaviour is able to make a step in the underlying language; this updates the global heap and the local context of the behaviour.

SPAWN describes spawning of a new behaviour. The underlying relation updates the spawning behaviour's context to accommodate the local effects of spawning a behaviour (such as updating stacks to reflect captured values and reducing expressions). The cowns required by the new behaviour ($\overline{\kappa'}$) and its context (E'') are added to the end of the pending behaviours list.

RUN describes running a behaviour. A behaviour can be run once two criteria are met: (1) no behaviour that appears earlier in list of pending behaviours has overlapping cowns with this behaviours cowns; (2) the cowns required by the behaviour are not in use by any running behaviour. Note, it is *not* required the first pending behaviour is selected.

END describes terminating a behaviour. This step requires that some running behaviour has reached some terminal state, it can then be removed from the running behaviours.

Ensuring happens-before. SPAWN demonstrates the linear structure of P, spawned behaviours are always appended to the end of the list of pending behaviours. In RUN a behaviour can only be made running and removed from P if there is no prior behaviour in P that requires the same cowns. This means that whenever two behaviours overlap, the earlier spawned behaviour will always be run first. Consider what this means for the example in Listing 8: $b1$ will be spawned before $b2$ and, as $b2$ spawns $b3$, also before $b3$ and so $b1$ will precede $b3$ in P, thus $b1$ will be run first.

Ensuring isolation of behaviours. These semantics provide a means to schedule behaviours such that no running behaviours are granted access to the same cowns at once. This is not in itself enough to isolate behaviours: we also need the *contents* of these cowns and the states of the behaviours to not overlap. Thus, we require the underlying language to provide a mechanism for memory isolation. These semantics permit, and in fact we strongly recommend, such a mechanism to ensure behaviour isolation.

Assume the underlying language has a mechanism for isolation, such as a type system. Consider what it means for a step to be allowed in the underlying language in STEP. One definition of this requires that a step must preserve the state of all memory from which this behaviour is isolated [Dinsdale-Young et al. 2013]. Thus a behaviour can mutate its cowns and local state, as long as it does not affect other isolated memory.

In more detail, for SPAWN we expect the type system to ensure any shared data accessed within the new context (E'') is protected (uniquely owned) by the required cowns ($\overline{\kappa}$). For END, we expect that the type system ensures that the data protected by the cowns being released ($\overline{\kappa}$) are disjoint.

Assuming this provisioned isolation, we can claim that behaviours are atomic. A behaviour will acquire its cowns, execute with isolated access to its cowns until completion and then release the cowns. There is no way one behaviour can observe another partially executed behaviour.

Ensuring deadlock freedom. BoC is deadlock-free by construction as the semantics in Figure 1 can always reduce unless all the behaviours have finished. To show this, we assume the underlying semantics cannot get stuck, that is:

$$\forall E.\ [\ (\forall h.\ \exists E', h'.\ E, h \hookrightarrow E', h')\ \vee\ (\exists E', \kappa, E''.\ E \hookrightarrow_{\mathsf{when}\ (\overline{\kappa})\ \{E''\}} E')\ \vee\ \mathit{finished}(E)\ ] \quad (1)$$

This may seem like too strong an assumption, as we expect that given a context and heap pair progress can always be made, yet this can be satisfied fairly simply by permitting error contexts that satisfy the *finished* predicate. In the presence of a heap in which a context cannot reduce, say some dangling pointer, this will step to an error which will then be *finished*. We can also satisfy this by defining well-formed configurations and proving preservation of such over the evaluation relation, but this is more involved than we require here.

We proceed by case analysis on R being empty. If it is not empty, then we can apply the assumption (1) to an element of R, which gives three cases, one for each disjunct. The three cases can reduce by STEP, SPAWN or END respectively. If R is empty and P is non-empty, then the first element of P can be moved to the running set using RUN. If R and P are both empty, then no rules apply and the program has terminated. Hence, the BoC semantics cannot get stuck before termination.

In the next section, we show how the implementations preserves deadlock freedom.

3.1 Demonstrating Parallelism and Deadlock Freedom

We now demonstrate that Listing 4 is deadlock free through the following potential execution. We use `w1`, `w2`, `d1` and `d2`, as behaviour identifiers for withdrawing from and depositing to `s1` and `s2` respectively. In the transition from Config 2 to Config 3, we use the RUN rule to start the second behaviour in the queue. This is allowed as it does not require any cowns required by an earlier behaviour. If `s1` and `s2` were the same cown, then the second behaviour could not run as it would require the same cown as the first behaviour. We then let the running behaviour complete. At this point (Config 4), there are two behaviours in the pending queue, but only the first can run, because they both require the same cown. This behaviour must now complete, before the remaining behaviour in the queue can run. Once both withdrawals are complete (Config 6), then both deposits can start in either order, and then they can run in parallel (Config 7).

$$\{(\varnothing, \texttt{transfer(s1, s2, 1); transfer(s2, s1, 2)})\}, [], h \rightsquigarrow^* \quad (1)$$
$$\varnothing, [(\{s1\}, \texttt{w1; when(s2) \{ d2 \}}), (\{s2\}, \texttt{w2; when(s1) \{ d1 \}})], h \rightsquigarrow \quad (2)$$
$$\{(\{s2\}, \texttt{w2; when(s1) \{ d1 \}})\}, [(\{s1\}, \texttt{w1; when(s2) \{ d2 \}})], h \rightsquigarrow \quad (3)$$
$$\varnothing, [(\{s1\}, \texttt{w1; when(s2) \{ d2 \}}), (\{s1\}, \texttt{d1})], h' \rightsquigarrow^* \quad (4)$$
$$\{(\{s1\}, \texttt{w1; when(s2) \{ d2 \}})\}, [(\{s1\}, \texttt{d1})], h' \rightsquigarrow^* \quad (5)$$
$$\varnothing, [(\{s1\}, \texttt{d1}), (\{s2\}, \texttt{d2})], h' \rightsquigarrow^* \quad (6)$$
$$\{(\{s1\}, \texttt{d1}), (\{s2\}, \texttt{d2})\}, [], h'' \quad (7)$$

4 IMPLEMENTATION

We have created a C++ library that functions as a runtime for BoC. While an interim solution, this library enables the execution of C++ programs that utilize the BoC approach to concurrency, providing valuable insight into the runtime's performance. Ultimately, we aim to integrate this runtime as a fundamental component of a programming language implementation that natively supports BoC as the exclusive method for concurrency, integrated with the language's type system.

List. 11. Example of ordering

```
1
2  when (c1) { /* b0 */ }
3
4  when (c3) { /* b1 */ }
5
6  when (c1, c2) { /* b2 */ }
7
8  when (c1) { /* b3 */ }
9
10 when (c2, c3) { /* b4 */ }
11
12 when (c3) { /* b5 */ }
```

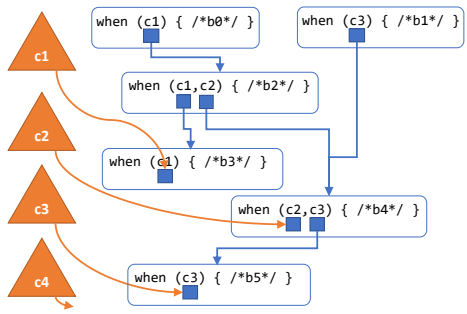


Fig. 2. Dependency graph for Listing 11

The development of the BoC runtime had to address the following challenges: correspondence between implementation and semantics, atomicity and deadlock-freedom of the operations of the runtime, and efficiency.

The BoC runtime is closely aligned with the operational semantics outlined in Section 3. In the runtime, behaviours operate within scheduler threads, and when a behaviour is ready to run (*i.e.,* it can be added to R), it is dispatched to an underlying scheduler (such as a thread pool) for execution. The pending behaviour queue (P) is represented as a dependency graph, which is updated when a behaviour terminates or a new behaviour is spawned, and used to determine if a behaviour is ready to run. Unlike more complex systems, there is no need for additional operations, such as logs, commit, or roll-back. This tight correspondence between the runtime and operational semantics is often absent in more complex systems, such as those described be Harris et al. [2005] and Moore and Grossman [2008].

4.1 High-Level Design

The *dependency graph* is at the heart of the BoC runtime. It is an directed acyclic graph (DAG) of behaviours, whose edges express the holding of a cown needed by a successor.

To provide further detail, let's consider a behaviour b that requires n cowns. In the behaviour dependency graph, b will have k predecessors, where $k \leq n$. These k predecessors will be behaviours that require one of the cowns required by b, and are either running or directly preceding b in the graph. And $n - k$ is the cowns that are not currently being used by any running behaviour and are also not required by any preceding behaviour.

In the rest of this subsection, we will expand on:

- When to start running a behaviour?
- What happens when a behaviour completes?
- How to spawn a new behaviour?

For now, we will assume that manipulations of the dependency graph are atomic with respect to themselves and each other; we will explain their detailed implementation in the next subsection.

We will use the program given in Listing 11 to illustrate how the runtime can execute a BoC program. Assuming that all the behaviours have been spawned, and none have started running yet, the dependency graph would look as shown in Figure 2.

A behaviour which has no predecessors in the dependency graph can be executed. Once it completes executing, it removes itself from the predecessor set of all it successors in the dependency graph. For instance, in the dependency graph in Figure 2, $b0$ may execute, and once it completes it

can remove itself from $b2$'s predecessors. In parallel with this, $b1$ may execute, and similarly when it completes it can also remove itself from $b4$'s predecessors. Hence, when both $b0$ and $b1$ have completed, $b2$ will have no predecessors, and thus can execute. When $b2$ completes, it will update its two successors, $b3$ and $b4$, which enables them both to be executed. Finally, $b5$ can be executed on completion of $b4$.

The runtime also handles adding new behaviours. The last behaviour scheduled on each cown is pointed to by that cown. For instance, in Figure 2, the cown `c1` refers to $b3$. This specifies where the dependency graph must be extended for the next behaviour that uses `c1`. Similarly, a new behaviour using `c2`, or `c3`, would have to be added as a successor to $b4$, or $b5$ respectively.

A cown which has no behaviours using it, such as `c4`, points to a special unused value, rather than a pointer to a behaviour. When scheduling a behaviour on a cown in this state, BoC replaces the special value by that behaviour. When the last behaviour finishes on a particular cown, it reestablishes the unused value.

Thus, if we were to schedule `when(c1,c4){/*b6 */}` on the dependency graph in Figure 2, we would update the cowns `c1` and `c4` to point to $b6$, and add $b6$ as a successor to $b3$ as that was the last value for `c1`. As `c4` had no behaviours using it, we would not need to add a successor for `c4`.

To summarize, the behaviour dependency graph evolves dynamically as new behaviours are introduced and terminated. The addition of new behaviours is represented by the addition of new leaves in the graph, which are pointed to by cowns. When a behaviour terminates, the corresponding edges in the graph are removed. The root nodes of the dependency graph are the behaviours that are either currently running or can potentially start running. It is important to note that the dependency graph may have multiple roots; also, when viewed from the perspective of any single cown, the graph appears as a linear order (or queue).

4.2 Formal Semantics Correspondence

We now connect this implementation with the formal semantics from Section 3.

In the semantics, a configuration consists of a set of running behaviours, a queue of pending behaviours and a heap. In the implementation, the running set are those behaviours executing on scheduler threads, whilst the centralised pending queue has been transformed into a decentralised DAG. The edges between two behaviours in the DAG are the happens before relation between two behaviours in the queue. As behaviours are executed from the root of the DAG, this correlates to taking a behaviour that must happen before other behaviours from the queue.

In the semantics, SPAWN appends a behaviour to the end of the pending queue. In the implementation, spawning a behaviour sets the last behaviour of each of the required cowns to the new behaviour, updating the cown's previously last behaviour to know about their new successor.

In the semantics, RUN starts any behaviour whose cowns are all currently available, and who does not require any cowns required by a behaviour that appears earlier in the queue. STEP continues the evaluation of this behaviour. In the implementation, any behaviour that has no predecessors, indicating all required cowns are available, can begin executing; in the DAG representation, the first behaviour that requires a cown must be a root of the DAG, so there is no search necessary as in the formal semantics. Once a behaviour starts running it can continue to execute until termination.

In the semantics, END terminates a behaviour, removing it from the running set. In the implementation, a behaviour terminates and decrements its successors number of pending predecessors.

```
class CownBase : StableOrder {
 volatile Request? last = null;
}

class Request {
 volatile Behaviour? next = null;
 volatile bool scheduled = false;
 CownBase target;

 Request(CownBase t) { target = t; }

 void StartAppendReq(Behaviour behaviour) {
  var prev = Exchange(ref target.last, this);
  if (prev == null) {
   behaviour.ResolveOne();
   return;
  }
  while (!prev.scheduled) { /*spin*/ }
  prev.next = behaviour;
 }

 void FinishAppendReq() { scheduled = true; }

 void Release() {
  if (next == null) {
   if (CompareExchange(ref target.last,
                       null, this) == this)
    return;
   while (next == null) { /*spin*/ }
  }
  next.ResolveOne();
}}

class Behaviour {
 Action thunk;
 int count;
 Request[] requests;

 Behaviour(Action t, CownBase[] cowns) {
  thunk = t;
  requests = new Request[cowns.Length];
  for (int i = 0; i < cowns.Length; i++)
   requests[i] = new Request(cowns[i]);
 }

 static void Schedule(Action t,
              params CownBase[] cowns) {
  Array.Sort(cowns);
  var behaviour = new Behaviour(t, cowns);
  behaviour.count = cowns.Length + 1;
  foreach (var r in behaviour.requests)
   r.StartAppendReq(behaviour);
  foreach (var r in behaviour.requests)
   r.FinishAppendReq();
  behaviour.ResolveOne();
 }

 void ResolveOne() {
  if (Decrement(ref count) != 0)
   return;
  Task.Run(() => {
   thunk();
   foreach (var r in requests)
    r.Release();
  });
}}
```

Fig. 3. C# implementation of the BoC runtime.

4.3 Model Implementation

We give a model C# implementation of the BoC runtime in Figure 3, to illustrate the core details required to schedule behaviours. In the next section, we will discuss the differences with the more performant C++ implementation.

The implementation is composed of three classes: `CownBase` – the common superclass representation of a cown (the triangles in Figure 2); `Behaviour` – the representation of the **`when`** block (the rounded squares in Figure 2); `Request` – the edges of the dependency graph (the blue squares and the blue arrows in Figure 2). The class `CownBase` has a single field `last` that either references the `Request` of the last behaviour to be scheduled on this cown, or `null` to signify there are no behaviours using nor waiting for this cown. `CownBase` also extends `StableOrder`, a class which enables sorting of its instances. A `Request` has three fields: `next` – a pointer to the next behaviour in the graph; `scheduled` – used to atomically append a request on multiple cowns; and `target` – the cown that this request is requesting. A `Behaviour` has three fields: `thunk` – the body of the **`when`**;

`requests` – the array of requests for this behaviour; and `count` – an atomic counter for how many predecessor behaviours must complete before the behaviour can execute.

Graph design. Our implementation is based on the MCS-queue lock [Mellor-Crummey and Scott 1991], which provides a guaranteed ordered spin lock. Recall, from the point of view of a cown, the behaviours form a queue. Our runtime adapts the MCS-queue lock data structure in two important ways: (1) the graph entries contain a reference to behaviours, rather than a flag for spinning on; and (2) it extends the enqueuing mechanism with a two phase enqueuing that allows an atomic enqueue on multiple cowns. The first change is how we move from *blocking* to *asynchrony*. The second enables **`when`** to be over multiple cowns, and correctly provide the order.

Resolving a `Request`. The instance method `ResolveOne` from class `Resource` is called when *one predecessor* can be removed from the dependency graph. It decrements the `count` of predecessors, and if `count` is decremented to zero (meaning that the behaviour has no dependencies), it passes the behaviour to `Task.Run` for execution. The scheduled task runs the behaviour body and once the body completes, it `Releases` each of its associated requests (Lines 63 and 64).

Spawning a behaviour. This is the responsibility of the static method `Schedule` from class `Behaviour`. Unlike our abstract semantics with a single queue to update atomically, the implementation must atomically update multiple cowns to extend the dependency graph.

To achieve atomicity of appending a request across a set of cowns (as required in Section 4.1), we separate the operation into two phases [Eswaran et al. 1976]. To achieve deadlock freedom (also required in Section 4.1), we sort the cowns into a globally agreed order (Line 48). This ensures that the append cannot deadlock [Havender 1968] due to a cycle in waiting on the `scheduled` flag.

The first phase calls `StartAppendReq` on every request (Lines 51 and 52). `StartAppendReq` uses an exchange to atomically set the current request as the last request for a required cown, and get the previous value (Line 13). If there is no previous value, then this dependency can be immediately resolved (Line 15). Otherwise, it waits for the `scheduled` flag on the predecessor's request to be set (Line 18), before linking the request in the graph (Line 19). The wait prevents this behaviour from appending on further cowns, until the prior behaviour has completed its append.

The second phase calls `FinishAppendReq` on every request (Lines 53 and 54). We want to ensure that the second phase does indeed run, before the behaviour is run (executed by `Task.Run`). This is why we set `count` to one higher than the number of requested cowns – Line 50. After the end of the second phase, the call of `ResolveOne` restores `count` its correct value, and runs the behaviour if indeed all its cowns were available – Line 54.

Completing a behaviour. The function `Release` from class `Request` notifies the next element in the graph (if a next element exists) that this predecessor has been resolved. There are three cases that the code must deal with, (1) the `next` pointer has been set, (2) this is the last request for the cown, and (3) this is not the last request for the cown, and the `next` pointer has not been set. This is identical to the cases for releasing a lock in the MCS lock. If the `next` pointer has been set, simply notify the successor by calling `ResolveOne`. If it has not been set, then attempt using a `CompareExchange` to set the cown to point back to `null`. If this succeeds, then there are no more behaviours on the cown and nothing more is required. If `CompareExchange` fails, then this is not the last behaviour on the cown and another behaviour is in the process of being scheduled. In this case, the thread spins until the `next` pointer is set (line 29), and then notifies it.

4.4 Optimised Implementation

We have developed a high-performance C++ implementation of the BoC runtime. There are a few key implementation differences with the C# implementation we have just described.

The main differences is that the C++ implementation packs all the objects associated with a **when** into a single allocation. The C# implementation for a **when** with n cowns, results in $n + 2$ allocations: one for each request, one for the array, and one for the behaviour. The C++ implementation lays out all the objects inside a single allocation. All the objects have a shared lifetime, so this is a simple optimisation. This is not possible in the C# implementation as the .NET GC does not support interior pointers on the heap. Similarly, the C++ implementation encodes the `next` and `scheduled` field into a single pointer using standard bit borrowing tricks.

The C++ implementation must handle the manual deallocation of the heap allocations. It is safe to deallocate the behaviour once it has completed releasing its successors. This is safe because there are three ways a behaviour can be reached by its predecessors, its successors and the cowns it is scheduled on. For the behaviour to have executed, the `count` must be zero, so there are no predecessors left. If a request has a `next` that has been set, then it is not reachable from the cown, and will not be accessed again by that successor. Note that it is important that the wait on scheduled occurs before setting `next`, if they were in the other ordered then there would be a use after free. The `Release` call also removes any direct reference from a cown if it still exists. Hence, once `Release` has been called on each request the object is unreachable and may be deallocated.

The C++ runtime has a work stealing scheduler. To improve performance, we keep the most recently runnable behaviour in thread local state, so that we do not have to pay scheduling costs. This has the effect of running several related behaviours in a batch. After n thread-local running behaviours, we always use the shared scheduler to ensure we do not starve the rest of the system.

4.5 Correctness: Deadlock Freedom, Atomicity, and Progress

In Section 3 we demonstrated that the BoC semantics guarantees both deadlock freedom and atomicity of behaviours with respect to one another. However, the question arises as to whether these guarantees still hold when the behaviours are executed within the BoC runtime.

In Section 4.3 we established that operations for manipulating the behaviour dependency graph shown in Figure 3 are atomic with respect to each other. When running the `thunk` (Line 62), the BoC runtime may affect the contents of the cowns, but not the behaviour dependency graph. All other functions may affect the behaviour dependency graph, but not the contents of the cowns. And while running the `thunk`, they only affect the contents of the cowns. As a result, the footprint of the behaviour dependency graph manipulating operations and that of the `thunk` are disjoint. Therefore, the behaviours as executed by our BoC runtime are atomic with respect to one another.

Similarly, the dependency graph manipulation operations are deadlock-free. By the same argument of footprint disjointness, behaviours as executed by our BoC runtime are also deadlock-free.

Finally, due to the acyclic nature of the dependency graph and the fact that the roots of the graph are either executing or can be started, we also have progress.

5 EVALUATION

The two main contributions of this paper are (a) a new concurrency paradigm and (b) an implementation of it. In order to evaluate these, we needed a baseline. Savina is a benchmark suite designed to compare implementations of the actor paradigm [Imam and Sarkar 2014]. This suite is widely known in the actors community and has been used to compare the performance of several actor languages [Blessing et al. 2019]. We chose Savina for our evaluation for two reasons:

(1) Actors model programs have close relation to BoC and there is a straightforward translation from actors to BoC. Thus, we can compare performance of our runtime against existing runtimes.

(2) There are patterns in actor model programs which we have identified as places where BoC is a better fit; thus, we can compare tradeoffs between these paradigms.

All our experiments were run on an Azure F72s v2 instance, which has 72 hardware threads.[1]

5.1 Evaluation of the Implementation

To evaluate the implementation, we started by comparing the performance of the programs in the Savina suite written in an actor language, and written in "BoC (Actor)" – *i.e.,* BoC where each behaviour runs on exactly one cown. Such a comparison allows us to investigate the costs of the BoC's more complex messaging mechanism to that of actors'.

We chose to compare with the actor language Pony [PonySite [n. d.]] because the latest comparison [Blessing et al. 2019] showed Pony to be comparable in performance with both Akka [AkkaSite [n. d.]; Haller and Odersky 2009] and CAF [CAFSite [n. d.]; Hiesgen et al. 2016]. Thus, by being comparable with Pony we are comparable across the space of actor-based languages, and demonstrate that BoC does not introduce high overheads. Furthermore, Pony and BoC share some agreeable language features, *e.g.,* message ordering guarantees, and no need to explicitly terminate actors (poison pills). Finally, many of the implementation design decisions were taken from the Pony implementation which makes for more direct comparison [Clebsch 2018; Clebsch et al. 2015].

To make the comparison truly informative, we developed a systematic mapping from Pony to BoC (Actor), and applied it for all the programs. Each Pony actor is mapped to a cown, and each Pony behaviour is mapped to a method with the same arguments as the behaviour, as well as the cown corresponding to the receiver, and the method body starting with a `when` on that cown.

We present the results in Table 1, where we run each program with 1 and 8 cores. The Savina benchmark suite consists of 30 programs. In accordance with Blessing et al. [2019], we dropped 8 programs because they relied on language specific, non-standard libraries, and one has problems with termination; thus, we have 22 programs. We ran each benchmark 100 times and report the average and an approximation of the confidence interval (the standard error times 1.96). We use Pony version 0.53.0. To give additional insights into the benchmarks, we present the number of cowns and behaviours in each benchmark (obtained by instrumenting the runtime to detect cown/behaviour allocation and logging the result on termination).

The "BoC (Actor)" implementations are faster than Pony on 17 out of 22 benchmarks. The majority of results are similar between the two implementations suggesting the dependency graph used by BoC has negligible affect on performance. We investigated the larger outliers to understand what caused the performance differences.

Where Pony has better performance. The worst performance of our runtime relative to Pony is the Sleeping Barber. This involves busy waiting, which causes a lot of pointless work on our runtime. Pony has a mechanism for back pressure that gives the busy waiting a low priority, so the rest of the system makes progress.

The second worst performance is Sieve of Eratosthenes. When we analysed the code, using the profiling information, we found that Pony was using 32-bit division, whereas we were using 64-bit division. We are unsure why Pony used 32-bit division, but if we change our implementation to use 32-bit division, we get the similar performance to Pony.

The single core Count works better for Pony as the overhead of allocating a message is lower. The example is completely single threaded, there is at most one Actor/Cown that can be executed at a time. For the multi-threaded run, the BoC batching scheme works better on this example than Pony's. Pony moves the work between multiple threads slowing down the processing.

[1] Our, anonymized, benchmarks can be found at https://anonymous.4open.science/r/benchmarks-0CF1/

Table 1. Average runtime (ms) on Savina benchmarks. Parenthesised values give logarithm relative overheads to single core Pony, $\log(\frac{x}{\text{1 core Pony}})$.

	Pony				BoC (Actor)							
Benchmark	LoC	1 core	8 cores		LoC	1 core		8 cores		cowns	behs	
Banking	105	184 ± 1.5	48.5 ± 2.0	(-0.6)	117	**19.8** ± 0.6	(-1.0)	22.6 ± 1.6	(-0.9)	1001	$10^{5.4}$	
Big	65	250 ± 0.3	501 ± 0.4	(0.3)	83	247 ± 0.2	(-0.0)	**202** ± 0.3	(-0.1)	121	$10^{6.7}$	
Bounded Buffer	132	2984 ± 0.1	406 ± 0.1	(-0.9)	142	1593 ± 0.1	(-0.3)	**342** ± 0.1	(-0.9)	81	$10^{5.2}$	
Chameneos	103	110 ± 0.5	522 ± 0.1	(0.7)	105	**46.1** ± 0.2	(-0.4)	94.5 ± 1.4	(-0.1)	101	$10^{5.9}$	
Cig Smokers	45	1.9 ± 4.0	2.2 ± 0.7	(0.1)	66	**0.7** ± 0.4	(-0.4)	1.4 ± 0.4	(-0.1)	201	$10^{3.5}$	
Conc Dict	65	256 ± 0.4	366 ± 0.4	(0.2)	84	**22.7** ± 0.3	(-1.1)	66.5 ± 1.2	(-0.6)	22	$10^{5.6}$	
Conc Sorted List	90	13653 ± 0.2	15059 ± 1.0	(0.0)	108	**6206** ± 0.1	(-0.3)	9913 ± 0.8	(-0.1)	22	$10^{5.5}$	
Count	28	**25.1** ± 1.4	120 ± 9.4	(0.7)	40	46.1 ± 0.4	(0.3)	47.7 ± 0.4	(0.3)	2	$10^{6.0}$	
Dining Phils	78	142 ± 0.5	560 ± 2.9	(0.6)	94	**73.3** ± 0.2	(-0.3)	165 ± 0.9	(0.1)	21	$10^{6.1}$	
Fib	44	322 ± 0.8	46.7 ± 1.4	(-0.8)	51	29.3 ± 0.3	(-1.0)	**19.4** ± 0.7	(-1.2)	150049	$10^{5.5}$	
Filterbank	225	2787 ± 0.1	380 ± 4.2	(-0.9)	247	1170 ± 0.1	(-0.4)	**349** ± 1.6	(-0.9)	62	$10^{6.2}$	
FJ Create	28	50.2 ± 1.6	31.3 ± 2.2	(-0.2)	42	**10.5** ± 0.4	(-0.7)	12.4 ± 0.4	(-0.6)	40001	$10^{4.9}$	
FJ Throughput	42	**50.2** ± 3.6	207 ± 5.9	(0.6)	53	53.4 ± 0.9	(0.0)	101 ± 0.6	(0.3)	61	$10^{6.1}$	
Map Series	137	832 ± 0.1	44.1 ± 12	(-1.3)	133	**39.0** ± 0.4	(-1.3)	41.2 ± 4.0	(-1.3)	21	$10^{5.9}$	
Ping Pong	29	7.2 ± 0.4	51.3 ± 3.4	(0.9)	43	**4.6** ± 0.5	(-0.2)	5.4 ± 0.4	(-0.1)	2	$10^{4.9}$	
Quicksort	126	234 ± 0.2	67.1 ± 5.2	(-0.5)	121	125 ± 0.3	(-0.3)	**58.8** ± 0.4	(-0.6)	1935	$10^{3.6}$	
Radixsort	77	180 ± 0.5	210 ± 0.8	(0.1)	105	236 ± 0.2	(0.1)	**149** ± 0.8	(-0.1)	61	$10^{6.8}$	
Matrix Mul	142	9825 ± 2.6	1158 ± 9.3	(-0.9)	156	1614 ± 0.1	(-0.8)	**563** ± 1.5	(-1.2)	22	$10^{3.4}$	
Sieve	64	222 ± 66	**45.2** ± 9.9	(-0.7)	68	331 ± 0.2	(0.2)	104 ± 3.8	(-0.3)	10	$10^{5.0}$	
Sleeping Barber	113	3652 ± 0.3	**261** ± 4.7	(-1.1)	127	1660 ± 0.1	(-0.3)	3544 ± 0.9	(-0.0)	5003	$10^{7.4}$	
Thread Ring	39	11.0 ± 1.1	95.1 ± 2.3	(0.9)	46	**4.7** ± 0.6	(-0.4)	5.5 ± 0.3	(-0.3)	100	$10^{5.0}$	
Trapezoid	62	842 ± 0.1	**115** ± 0.9	(-0.9)	72	970 ± 0.1	(0.1)	172 ± 0.1	(-0.7)	101	$10^{2.3}$	

Observe that the LoC for each benchmark is often lower in Pony; yet, the largest delta is 28 LoC in Radix Sort. Here are some reasons for the difference: Pony doesn't use braces to delineate scope; In BoC each `when` is wrapped in a function call to match the style of behaviours in Pony; Pony provides union types and parametric polymorphism that C++ and the BoC runtime do not. Radix Sort is strongly affected by these last two issue. However, we will see in Table 2 in Section 5.3 that BoC improves for multi-cown behaviours.

Where BoC (Actor) has better performance. The Bounded Buffer, Concurrent SortedList and Matrix Mult all had poor performance on Pony relative to our runtime. When we investigated profiling information, each of these examples had generated less optimal inner loops for the core computation.

The Banking, Chameneous and Concurrent Dictionary examples pass references to actors in messages. In our C++ BoC runtime, this is supported with reference counting. In Pony this uses remembered sets and tracing. Profiling showed that these memory management costs caused the overhead. Similarly, Fib's overheads for Pony are primarily due to the cost of deallocating actors, which is cheaper on our runtime.

Overall, the majority of difference are not due to the BoC runtime, so there is no evidence that BoC introduces high overheads.

5.2 Evaluation of the BoC Paradigm

To evaluate the BoC paradigm, we make qualitative and quantitative comparisons based on the Savina suite.

For our comparison based on the Savina suite, we considered which of the Savina programs would benefit from the extra power afforded by behaviours that support more than one cowns. Not all Savina programs offer scope for change (for example, actors sending messages in a ring or pinging each other), but there are several programs which do have scope. They fall broadly into the following two categories (*c.f.,* Table 2): Multi-actor operations, where **when** helps, and Parallelising workloads where behaviour ordering helps.

Table 2. Overview of Savina benchmarks we consider.

Pattern	Benchmarks
Multi-actor	Banking, Barber, Dining Philosophers, Chameneos
Parallelism	Logistic Map Series, Count, Fibonacci, Fork-Join Create, Fork-Join Throughput, Quicksort, Trapezoid

Multi-actor operations. These are problems such as two-phase commit and rendezvous of actors. Two (or more) actors must atomically update state through message exchange; for example, a transaction manager coordinates updates between two bank accounts, such that an account is only involved in one operation at a time, and either both or neither accounts are updated. In Section 5.3 we will discuss this pattern in detail. Using BoC (Full) reduces the required message through a **when** that acquires multiple cowns at once.

Parallelising workloads. These are problems in the vein of divide and conquer, fork-join, and map reduce, where a problem is decomposed into smaller problems that can be solved in parallel and combined. In actor systems, each parallel solution needs to send a message back to an aggregating actor to recompose the solution. Using BoC (Full) reduces the required message, as causal order ensures sub-solutions are only aggregated once they have been computed, and thus avoids sending messages back to an aggregating actor. Note, this pattern requires the work to be divided in a single behaviour, so whether this is faster depends on the amount of work performed in the asynchronous behaviours. For Fibonacci this change improves performance whereas for Quicksort it does not, thus there are related decisions regarding which algorithms should be used for a program.

In Table 3 we compare the performance of these programs written in BoC (Actor) and written in BoC (Full). We see that 6 benchmarks demonstrate significant improvement in performance. We also present the lines of code for the implementations of the benchmarks as an approximation of the complexity of the solutions. Again, we see that 9 of the benchmarks are smaller than the Actor based implementations. We present the behaviour count for each benchmark, and additionally for BoC (Full) we separate this by the number of cowns used in each behaviour. We observe that all the benchmarks with significant speedups have a significantly lower behaviour count, and a high percentage of these use two cowns. In some benchmarks the number of cowns required changes when using BoC (Full); often this number gets lower, e.g. Fib, as BoC constructs the computation bottom up and so is able to reuse cowns, whilst in Pony the computation is top down with actors waiting in place for subproblem results; in the Dining Philosophers, in Pony, forks are not modelled as actors whilst they are in BoC, so as to decentralise the fork management and provide a better solution using BoC. We can also see that the total number of behaviours remains the same or decreases from BoC (Actor) to BoC (Full); this always comes from creating a single behaviour that accesses multiple cowns in place of multiple messages to coordinate access over multiple cowns.

We see improvements in both size and speed when adopting BoC (Full).

Table 3. Average runtime (ms) on selected Savina benchmarks

	BoC (Actor)					BoC (Full)						
										behaviours		
Benchmark	LoC	1 core (ms)	8 cores (ms)	cowns	behaviours	LoC	1 core (ms)	8 cores (ms)	cowns	1 cown	2 cowns	total
Banking	117	19.8 ± 0.6	22.6 ± 1.6	1001	$10^{5.4}$	78	**7.9** ± 1.2	9.3 ± 4.5	1001	$10^{4.7}$	$10^{4.7}$	$10^{5.0}$
Chameneos	105	**46.1** ± 0.2	94.5 ± 1.4	101	$10^{5.9}$	85	49.2 ± 0.3	122.1 ± 1.7	101	$10^{5.6}$	$10^{5.3}$	$10^{5.8}$
Count	40	46.1 ± 0.4	47.7 ± 0.4	2	$10^{6.0}$	30	**45.4** ± 0.4	46.9 ± 0.6	2	$10^{6.0}$	$10^{0.0}$	$10^{6.0}$
Dining Phils	94	73.3 ± 0.2	164.9 ± 0.9	21	$10^{6.1}$	61	22.2 ± 0.7	**16.6** ± 0.8	41	$10^{5.3}$	$10^{5.3}$	$10^{5.6}$
Fib	51	29.3 ± 0.3	19.4 ± 0.7	150049	$10^{5.5}$	28	**10.9** ± 0.4	13.5 ± 0.8	75025	0	$10^{4.9}$	$10^{4.9}$
FJ Create	42	10.5 ± 0.4	12.4 ± 0.4	40001	$10^{4.9}$	42	**9.6** ± 0.3	11.8 ± 0.3	40001	$10^{4.6}$	$10^{4.6}$	$10^{4.9}$
FJ Throughput	53	53.4 ± 0.9	100.8 ± 0.6	61	$10^{6.1}$	52	**35.8** ± 1.2	45.8 ± 2.2	61	$10^{5.8}$	$10^{1.8}$	$10^{5.8}$
Map Series	133	39.0 ± 0.4	41.2 ± 4.0	21	$10^{5.9}$	52	**17.4** ± 1.0	19.8 ± 0.9	21	0	$10^{5.4}$	$10^{5.4}$
Quicksort	121	124.7 ± 0.3	**58.8** ± 0.4	1935	$10^{3.6}$	85	105.9 ± 0.3	78.6 ± 0.4	968	$10^{3.0}$	$10^{3.0}$	$10^{3.3}$
Sleeping Barber	127	1660 ± 0.1	3544 ± 0.9	5003	$10^{7.4}$	106	**12.5** ± 7.7	16.5 ± 10.2	5003	$10^{4.8}$	$10^{4.0}$	$10^{4.9}$
Trapezoid	72	970.0 ± 0.1	**172.0** ± 0.1	101	$10^{2.3}$	68	958.3 ± 0.1	172.6 ± 0.1	100	$10^{2.0}$	$10^{2.0}$	$10^{2.3}$

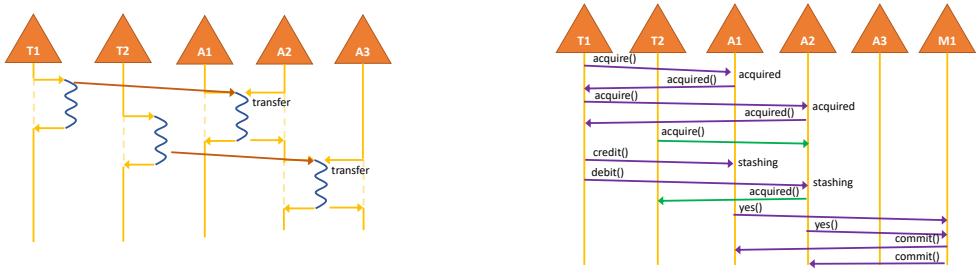
5.3 `ReasonableBanking`: The Banking Example Revisited

We now revisit the banking example, adding requirements that epitomise those often found in concurrency applications:

- Tellers can issue transactions between any accounts.
- There can be several Tellers issuing transactions to shared accounts.
- No livelocks and no deadlocks.
- Operations over multiple accounts are atomic.
- Any two transactions issued by one teller which involve the same account must be executed in the order they were issued.

We call the collection of these requirements the `ReasonableBanking`. The `ReasonableBanking` guarantees are inherently achieved by the BoC paradigm. As already shown in Section 2, the behaviour `transfer` involves two account cowns, and transfers the money. Any number of tellers issue such `transfer` transactions between the accounts. Figure 4a shows two such transactions.

Whilst it is possible to achieve the `ReasonableBanking` guarantees in an actor language (e.g. Pony), it is difficult. The Savina banking benchmark does not satisfy the guarantees because (a) it is difficult (b) the benchmarks are micro-benchmarks designed to stress components of an actor system. One design to achieve this is shown in Figure 4b (which presents only part of the protocol for two transactions for brevity), and goes as follows: There are three types of actor involved in a transaction, `Tellers`, `Accounts` and `Managers`; `Accounts` have two state flags (`acquired` and `stashing`) and associated queues which are used to track incoming requests. A `Teller` selects two accounts at random and sends a message `acquire()` to the lowest `Account` address of the two, and records the other `Account` to be acquired. When the `Account` processes the `acquire()` message, if `acquired` is set then the acquire request is enqueued, otherwise `acquired` is set and the `Account` replies `acquired()` to the `Teller`. The `Teller` will then repeat this process with the other `Account`. Once both `Accounts` are acquired, the `Teller` will create a `Manager` actor, and send `credit()` and `debit()` messages to the `Accounts`, including a reference to the `Manager` in the message payload. Each `Account` will test whether it is processing an operation through `stashing`, if it is, then the message will be enqueued, otherwise the account will decide whether the operation will succeed or not and inform the `Manager` through a `yes()` or `no()` message. The `Account` will also mark itself as stashing; this also releases the acquisition by a `Teller` and the account will process the next pending acquisition (if one exists). The `Manager` will aggregate the responses and reply `commit()` or `abort()` to the `Accounts` and inform the `Teller` of a completed transaction. The `Accounts` will then commit or rollback the operation.



(a) with BoC (Full)

(b) with Actors

Fig. 4. `ReasonableBanking`: Protocols for bank transfers

This is, evidently, a complicated protocol that requires careful construction. Moreover, it requires the actors (here `Teller`, and `Account`) to be aware of the protocol, thus mixing business logic with the protocol. Furthermore, if a new set of actors had similar protocol requirements, then the behaviours described above would need to be mixed into *their* logic.

Table 4. Average runtime (ms) and lines of code (LoC) of ReasonableBanking benchmark

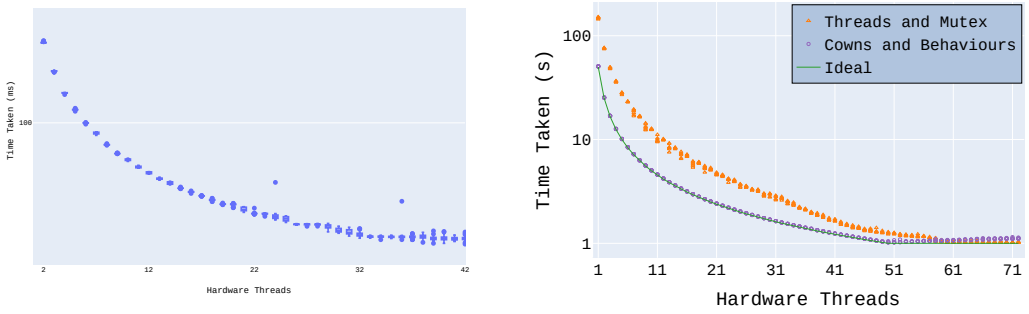
BoC (Full)			Pony		
LoC	1 core (ms)	8 cores (ms)	LoC	1 core (ms)	8 cores (ms)
78	7.9 ± 1.2	9.3 ± 4.5	226	987.9 ± 0.33	344.717 ± 0.35

Table 4 compares `ReasonableBanking` implemented in BoC (Full) and in Pony. Neither of the implementations scales with more cores; this is due to the very little work done by the each `transfer` – a common feature of the Savina suite. The BoC version is over 100 times faster than Pony on 1 core, and over 35 times faster than Pony on 8 cores. The Savina benchmarks focus on the runtime overheads, which causes the larger churn of messages to have a drastic effect on performance. In a realistic application, the improvement would be considerably smaller. The simplicity of the BoC version, already demonstrated in Figure 4, results in code that is 35% the length of that of Pony.

`ReasonableBanking` *in related work.* We discuss related work tackling the issue of updating multiple actors atomically in Section 6; here we will use the `ReasonableBanking` example to elucidate some of the points more deeply.

In Aeon [Sang et al. 2016], atomicity is achieved through the use of dominators. Actors are arranged in a DAG, and atomicity is guaranteed by an actor that dominates all the actors required in a particular operation. This unavoidably restricts parallelism as a dominator is responsible for serializing events for a set of actors. If we were to attempt the `ReasonableBankingBanking` example, then we would need to introduce an actor that dominated all the accounts. This is an additional actor that is required by the paradigm, but not by the business logic. More importantly, this additional actor would introduce a single point of contention, which could harm performance. BoC, does not need this single point of contention.

Chocola [Swalens et al. 2021] proposes that actors state is isolated, but actors can manipulate shared transactional state through transactions. This requires the programmer to decide if state should be transaction based or actor based. Transaction based access is synchronous whilst Actor access is asynchronous. Consider a sequence of three transfers: `src1` to `dst1`, `src2` to `dst2` and `dst1` to `dst2` in Chocola; the state of each account would need to be transactional for a behaviour to transfer funds (each transfer accesses two accounts, thus a single account cannot be an internal actor state). Moreover, we need that the third transfer should start executing only after the first two



(a) BusyBanking scaling

(b) Dining Philosophers scaling

Fig. 5. Benchmarking results

have completed. The first two transactions can execute in parallel as they access disjoint accounts. BoC is able to exploit the implicit parallelism whilst retaining order; in Chocola, the two readily available options are to start each transaction effectively in its own "thread" and lose all order, or execute them sequentially and lose all concurrency. To achieve both parallelism and order is the programmers burden through ad hoc coordination. These concerns affect `ReasonableBanking`, a series of ordered transfers.

The Akka Transactors [Akk [n. d.]] approach was built on STM, so again required splitting the world in transactional (STM) state and Actor state, and similar to Chocola performs synchronous STM operations. So would suffer the same negatives as described above. [2]

5.4 Scalability

Although the Savina benchmark suite is targeting highly concurrent Actor frameworks, many of the benchmarks do not have sufficient parallel work to benefit from multiple threads. For both Pony and BoC (Actor and Full), we see that about half of the benchmarks are faster with 1 core than 8 cores. This is due to the very small amount of work that is actually parallelised in the examples relative to the overhead of scheduling work. Additionally, our runtime (and the Pony runtime) performs work stealing, which can get in the way if there is insufficient work.

To illustrate our runtime's potential for scaling, we ran a modified version of the Savina Banking example using BoC (Full), which we call `BusyBanking`. We placed a $10\mu s$ busy loop in each transaction. We present the results in Figure 5a, and see the runtime scales well.

Our second scalability investigation was for the Dining Philosphers. We compare our BoC runtime with an implementation using standard C++ abstractions (*e.g.,* `std::lock`). We set the number of Philosophers to 100, which means that the maximum parallelism is 50 concurrent eats. While holding both forks, a Philosopher spin-waits for 1ms. We added this processing delay because it enables defining the optimal processing time of the experiment on an overhead-free system given the number of philosophers, the number of times they eat, and the available system-level parallelism. The choice of 1ms was made to reduce the impact of system overheads, either coming from the operating system and its scheduler or from the BoC runtime implementation, on the experiment results. We time how long it takes for all the Philosophers to eat 500 times each. We present the time taken for each number of hardware threads in Figure 5b. We present the "Ideal", which we calculated by dividing the 50 seconds of busy work by the number of hardware threads up to the maximum parallelism of 50.

[2]Note that Akka has dropped the support of transactors. https://github.com/akka/akka/pull/1878

The experiment has three main purposes. First, it demonstrates that **BoC can be efficiently implemented on a real system** and achieves close to the optimal performance. Second, it **showcases the implicit parallelism that BoC offers** the programmer compared to plain mutual exclusion and how this can be leveraged to achieve the optimal performance. Third, it compares BoC with vanilla C++ abstractions offering mutual exclusion, *i.e.,* mutexes, and **shows how the OS scheduler that lacks application level-insights can hinder scalability**, as opposed to BoC where happens-before ordering makes this insights explicit to the BoC runtime.

We present "Cowns and Behaviours", in Figure 5b, as the performance of our runtime. In this configuration we leverage BoC's happens-before ordering and control the philosopher execution order. By scheduling alternating Philosophers, first the odd and then the even, we are able to ensure a really fair order of eating. All the odd philosophers will have a turn, and then all the even. The happens-before order of the runtime and the fixed and common eat time ensure this continues for the rest of the execution. As you can see, the performance closely matches the ideal.

As the example reaches the limit of the concurrency, we see the runtime starts to slow down slightly. We believe this is because the current implementation of work-stealing does not backoff when there is insufficient work (harming performance). To ensure our runtime could scale beyond the 50 core mark, we additionally ran a version with 200 Philosophers, which was able to scale to the size of the machine (not shown).

Comparison to C++ threads and mutexes. The results for C++ implementation using `std::lock` are labelled as "Threads and Mutex". The mutexes are acquired using `std::lock`, which ensures deadlock freedom and uses a back-off strategy, when it fails to acquire the set of locks. The backoff strategy takes a significant amount of the computation time. This is why the code starts significantly above 50s for the single hardware thread case. As the number of hardware threads increases, the performance improves, and the backoff becomes a much smaller percentage of the runtime. This approach does reach the maximum parallelism for some runs with high hardware thread count.

In conclusion, our evaluation suggests the BoC paradigm can support the construction of powerful protocols in a convenient way, and that the BoC runtime is competitive with Pony (and therefore with other actor implementations), and has the potential to be much faster for examples that need the full power of BoC.

6 RELATED WORK AND REFLECTIONS

We have already positioned BoC with respect to actors [Agha 1985; Hewitt et al. 1973]. To elaborate on the pain point of actors, it is *often* remarked that actors are not a good fit for operating over multiple actors atomically [Bernstein 2018; Kraft et al. 2022; Plociniczak and Eisenbach 2010; Sang et al. 2016][3]. BoC directly supports operations over multiple cowns.

Another consideration is the order in which messages are delivered. The actor model does not specify an order of delivery and so different languages have chosen to provide different guarantees. In the actor language Pony, messages to the same actor are delivered in causal order [Blessing et al. 2017; Clebsch et al. 2015]. In contrast, the actor language SALSA gives no guarantees about ordering of delivery; only that a message will eventually be delivered. Any required ordering must be achieved through token-passing continuations that enable subsequent behaviours [Varela and Agha 2001]. Our happens before relation enables causal ordering over multiple cowns.

Similar to BoC's cowns, De Koster et al. propose extending actors and behaviours with *domains* and *views* as a solution to the shortcomings of isolation in the actor model [De Koster 2015; De Koster et al. 2012]. Views can acquire access to a domain during the execution of a behaviour. Like BoC's

[3] http://doc.akkasource.org/transactors

nested behaviours, dynamically nested views do not have access to the domain(s) acquired by the textually enclosing view. The views programmer may wrongly assume a nested view has access to the outer view, and may obtain runtime errors if the outer view is accessed but has expired. Such an error can be prevented statically via a type system, such as we propose for BoC in this work.

Multi-actor programming as in Aeon [Sang et al. 2020, 2016] has a similar remit to that of BoC: that programmers often want to reason about the composition of several messages for collaboration. It proposes that actors are placed in an acyclic ownership graph, messages are grouped into units which can be executed in a serializable manner. Like BoC, events are asynchronous; unlike BoC actors are organized in a DAG, which is used to enforce serialization by locking the actors as the events traverse the DAG. In the case where a large number of messages is sent concurrently, and where each of these messages has more than one different actors receivers, Aeon requires the receiver actors to be dominated by one single actor. In that case, all these essentially concurrent messages need to be coordinated by that single actor, thus introducing a single point of contention. Such a pattern is discussed in the `ReasonableBanking` example in Section 5.3.

Like BoC, *Transactors*, as they appear in [Field and Varela 2005], address the consistency of the state of several actors involved in processes with several stages. Unlike BoC, transactors are concerned with networks, and their failures. Thus, transactors extend the actor model by explicitly modeling node failures, network failures, persistent storage, and state immutability. Unlike BoC, messages are sent to/executed by a single actor.

The term *transactors* are also being proposed in Akka[3], but with slightly different meaning. These transactors, like BoC, address the flexibility of coordination across actors by enabling STM across multiple actors. Unlike BoC, this does not decouple the state and behaviour of actors and, so, an actor must present an interface through which transactions can be used. Contrast this with the ah-hoc creation of behaviours that acquire cowns and use them as necessary. Transactors do not so much present a single abstraction; rather the unification of two. When a multi-actor message/operation needs to update state, then that state will need to be transactional, even if part of it logically belonged to the first, and another part logically belonged to the second actor – we discussed such an example in Section 5.3. Transactions on that state would need to either each start on separate threads, thus losing ordering guarantees, or we would have to run all the transactions operating on that state on the same thread, thus losing concurrency.

On a similar note, the unification of futures, actors, and transactions has been studied in [Swalens et al. 2021] with a model and with an implementation on top of Closure. The remit here is to offer a model that supports all three paradigms in a faithful manner, i.e. to preserve the guarantees of each constituent paradigm whenever possible. Again, this is unlike BoC as multiple parallelism and coordination abstractions are presented to a programmer instead of one. This means the programmer must be aware of which intersection they are using and the guarantees it provides. With respect to the pattern of a multi-actor message updating some state, the same limitations apply as in the earlier paragraph: If the state is made transactional, then one will either lose concurrency or lose ordering guarantees - more under the discussion of `reasonableBanking` in section Section 5.3.

The idea of atomically acquiring more than one resource has been tried in the synchronous setting in AJ [Dolby et al. 2012; Vaziri et al. 2006] which advocates that instead of focusing on the flow of control, programmers should identify sets of memory locations that share some consistency property and group those locations in atomic sets that will be updated atomically. AJ also allows for a method to coarsen the granularity of atomicity for some of its arguments by annotating them with the `unitfor` keyword: the atomic sets of all these objects will be locked atomically. However, nesting of such functions may create deadlocks, while the asynchronous nature of **`when`**s in BoC

prevents any such deadlocks. Again because of its synchronous nature, AJ is not (and cannot be) concerned with ordering of method calls with `unitfor` annotations.

Behaviours in BoC are similar to *transactions* in that they both provide a powerful abstraction supporting the execution of several units of concurrency in parallel, while giving the appearance of each unit being executed sequentially. However, unlike BoC, optimistic transactions might abort and transaction roll-backs are expensive.

More seriously, not all effects of a transaction can be reversed, e.g. I/O. As a result [Welc et al. 2008] propose *irrevocable* transactions, i.e. transactions which cannot be rolled back, with the restriction that at most one such transaction may be running at a time – thus reducing parallelism. A similar approach is taken in [Harris et al. 2005]. Fully pessimistic transactions [Matveev and Shavit 2012] expand on irrevocable transactions, eliminating the need for any roll-backs; however, they require either a single writing transaction (with versioning) or allowing for potential deadlock.

Considerable effort has been devoted to conflict avoidance and resolution in transactions [Herlihy et al. 2003; Huang et al. 2022; Qin et al. 2021]. Ordering transactions can be used to alleviate contention and conflict; even better, ordering can improve overall parallelism and throughput [Qin et al. 2021]. We see ordering appear also in deterministic databases systems where it is not enough that transactions will commit in some order, but they must commit in a single predetermined order [Abadi and Faleiro 2018]. BoC implements out of the box causal ordering of behaviours.

Furthermore, the space for handling nested transactions is vast and seemingly unresolved [Koskinen and Herlihy 2008; Ni et al. 2007]. Contrast this with BoC, which provides a single clear semantics for nested **`when`**s, that is to say they spawn a new behaviour.

In contrast to BoC behaviours, transactions are not units of concurrent execution, rather they coordinate concurrent execution, e.g., threads. As such, in contrast to BoC they do not specify an order of execution. Moreover, nesting in transactions is synchronous, and therefore roll-back of the enclosing and the nested transaction are intertwined, leading to a large design space, with implications for the semantics and the implementations [Ni et al. 2007].

In summary, BoC and transactions are at different ends of the design spectrum when it comes to how much explicit work is required: *BoC is explicit*, in that it enlists the help of the type system to achieve isolation, and requires the programmer to declare upfront which cowns are required by a behaviour, while *transactions are implicit*, as the programmer need only declare what is a transaction, and all the scheduling and conflict resolution is expected to be done by the implementation. The implicit nature of the transactions model makes programming easier, at the expense of more demands on the implementation, and more complex semantics.

The acquisition of multiple cowns in a **`when`** is reminiscent of the chord in the Join calculus [Fournet and Gonthier 1996, 2000]. The join calculus has messages containing values that are sent on named channels; several such messages may be consumed by join patterns, thus decoupling state from concurrent units, and supporting the coordinated processing of messages from several channels. The Join calculus has inspired languages such as JoCaml, JErlang and Polyphonic C$^\sharp$ with Chords [Benton et al. 2002; Conchon and Le Fessant 1999; Plociniczak and Eisenbach 2010]. The tight coupling of state and channels makes the paradigm well-suited for distributed programming. However, this coupling introduces the need for careful management of messages so as to accurately represent state, and to maintain isolation of the state. BoC naturally captures isolated state through cowns.

The cowns in BoC provide a natural notion of sequencing of a state, something that has to be built on top of the join-calculus by threading a state by repeated messaging. Moreover, the join-calculus encodes order through messages, rather than he implicit order we get from BoC; we believe that this implicit order would make a significant difference in terms of programmability.

In some sense, BoC and the join calculus can be seen as dual. Join calculus is about `n` message sends triggering a join or chord, whereas BoC is about one message being received by `n` cowns triggering a behaviour. Taking this analogy a bit further, the end of a behaviour's scope would correspond to the end of a chord. On the other hand, each behaviour spawned (nested or otherwise) completes independently and does not join with any other unit of execution. A deeper study of the correspondence between BoC and the join calculus is fascinating further work.

The importance of determinism in parallelism is highlighted by Gonnord et al. [2022]. BoC guarantees causal ordering which can lead to determinism if all behaviours are causally interconnected. Examples of completely deterministic systems can be found in Deterministic Parallel Java [Bocchino Jr et al. 2009], DThreads [Liu et al. 2011] and Block-STM[Gelashvili et al. 2022]. In Deterministic Parallel Java, the programmer is required to explicitly express the parallelization opportunities, which is not always possible given the dynamic nature of the problem. BoC's implicit ordering can be used to express the dynamic order dependencies.

There are many systems that provide elaborate dataflow ordering such as MPI [Walker 1994], CnC [Budimlić et al. 2010] and Naiad [Murray et al. 2013]. These approaches are great at splitting one task into multiple pieces and combining the results, but do not provide support for mediating access to resources in the way that BoC can with cowns.

Habanero provides explicit ordering of asynchronous tasks through Phasers[Cavé et al. 2011], contrast with BoC's implicit order through program order and cown use. More deeply, Habanero provides ordering tailored for data parallel tasks, whilst BoC is ordering over resource access. This means for the pedagogical use of phasers presented in [Shirako et al. 2008], BoC does not have as intuitive a solution (note, it is still achievable). Conversely, the fine-grained ordering of log access in Listing 8 in this paper, does not have an intuitive solution using Phasers (one would require many Phasers to essentially build our implementation on top of Phasers).

Reflections on BoC. We believe BoC is a very natural paradigm for many applications, and the evaluation demonstrates that BoC compares well with other approaches both qualitatively and quantitatively. However, there are several programming patterns for which BoC is not intended, or will be difficult to adapt.

One such programming pattern is a dynamically growing set of resources, where the identity of the resources to acquire would only be known dynamically. Say we wanted to first acquire `r1`, and then, under some circumstances *also* acquire *either* `r2` *or* `r3`. With locks this can be done in a very natural manner; with STM this can be done with a transaction that just accesses more transactional objects as it proceeds. Programming this pattern in BoC would be less elegant, and would not have exactly the same meaning: It would require an outer **`when`** that acquires cown `r1`, and an inner **`when`** that acquires `r1` and *also either* `r2` *or* `r3`. But then, the inner **`when`** will not run until the outer **`when`** has terminated and released `r1`. Moreover, there is no guarantee that no other (locally unknown) behaviour will acquire `r1` between the outer **`when`** releasing it, and the inner **`when`** acquiring it.

Another such programming pattern is increasing parallelism through temporarily allowing a cown to be accessed by several behaviours concurrently in a read-only mode. Once all these behaviours have terminated, the cown can be acquired uniquely, in a read-write mode. Such programming patterns are needed, e.g, when many different concurrent units read from a shared large datastructure (e.g. a map), and send updates to another concurrent unit. Once the latter collects all the updates, it can modify the large datastructure, and notify the former.

Returning results through promises is currently not supported in BoC. Promises can be encoded through cowns, thus their inclusion would not add to the expressive power; but their inclusion would add to the usability of the language.

One of the more fundamental open questions around the design of BoC is whether the paradigm can be adapted to distributed programming. The actor model seamlessly extends to distributed programming, as shown by Erlang and Akka. Could BoC be used as the basis for distributed programming? Because of isolation of actor state, and because behaviours only access the state of the particular actor, there is no immediate need to migrate actors – unless you want to co-locate actors which communicate often. But for distributed BoC to be efficient, we would need this migration, so that the collection of cowns needed by a behaviour becomes co-located. This might lead to interesting patterns of migration where commonly used together cowns will reside on the same node. But we are a long way from making any claims in this space.

7 CONCLUSION

In this paper, we introduced BoC, arguing that it provides parallelism and *flexible* coordination though a single powerful abstraction. Furthermore, we argued that it provides a causal ordering for concurrent units, provides implicit parallelism, and is both deadlock and data-race free. Also, we have shown the applicability of BoC through examples and challenges. We discussed how to implement BoC, provided insights into doing so efficiently, and shown that BoC compares well with actors: it simplifies the design of many problems, while remaining comparable, if not better, in performance than actors.

We initially developed BoC as part of the programming language Verona, a statically typed, object-oriented, programming language currently being designed and implemented at Microsoft, Imperial, Uppsala and Wellington. Crucial aspects of the Verona type system are described in the OOPSLA companion paper []. The Verona runtime is available in the open github repo[4]. The repository contains a C++ DSL-like template library that can be used to evaluate examples. We are also currently working on an introduction of BoC to Python, where the type guarantees will be checked dynamically.

In future work, we want to further develop the formal model of BoC (e.g. explore the isolation guarantees an underlying language must provide), reasoning about the correctness of BoC programs, study of further aspects of the implementation, *e.g.*, backpressure. In this paper we have shown the paradigm's utility on small examples, in the future we aim to push this demonstration to use in large real-world applications.

In our opinion, the **`when`** construct is simple, intuitive, and expressive. We hope future languages will adopt behaviour-oriented concurrency, and that **`when`** will become as ubiquitous as **`if`**.

8 ACKNOWLEDGEMENTS

We thank the EPSRC for financial support of some aspects of this work. This work was partially supported by a grant from the Swedish Research Council (2020-05346). We thank Sean Allen, Wes Filardo, Adrien Ghosn, James Noble and the anonymous reviewers for OOPSLA-22, PLDI-23 and OOPLSA-23 for many interesting questions and feedback. We are deeply grateful to Elisa Gonzalez Boix for acting as our shepherd, and for giving us prompt, insightful and extensive feedback.

9 DATA-AVAILABILITY STATEMENT

We have submitted an artifact for our work, see [Cheeseman et al. 2023].

REFERENCES

[n. d.]. Transactors. https://doc.akka.io/docs/akka/2.2/scala/transactors.html

Daniel J Abadi and Jose M Faleiro. 2018. An overview of deterministic database systems. *Commun. ACM* 61, 9 (2018), 78–88. https://doi.org/10.1145/3181853

[4]https://github.com/Microsoft/verona-rt

Gul A Agha. 1985. *Actors: A model of concurrent computation in distributed systems.* Technical Report. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab. https://doi.org/10.7551/mitpress/1086.001.0001

Developers AkkaSite. [n. d.]. Akka repo. https://akka.io/docs/

Gene M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '67 (Spring))*. Association for Computing Machinery, New York, NY, USA, 483–485. https://doi.org/10.1145/1465482.1465560

Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023). https://doi.org/10.1145/3622846

Nick Benton, Luca Cardelli, and Cédric Fournet. 2002. Modern concurrency abstractions for C#. In *European Conference on Object-Oriented Programming*. Springer, 415–440. https://doi.org/10.1007/3-540-47993-7_18

Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. 2008. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep* 15 (2008), 181.

Philip A Bernstein. 2018. Actor-oriented database systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 13–14. https://doi.org/10.1109/icde.2018.00010

Sebastian Blessing, Sylvan Clebsch, and Sophia Drossopoulou. 2017. Tree topologies for causal message delivery. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 1–10. https://doi.org/10.1145/3141834.3141835

Sebastian Blessing, Kiko Fernandez-Reyes, Albert Mingkun Yang, Sophia Drossopoulou, and Tobias Wrigstad. 2019. Run, actor, run: towards cross-actor language benchmarking. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 41–50. https://doi.org/10.1145/3358499.3361224

Robert L Bocchino Jr, Vikram S Adve, Danny Dig, Sarita V Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 97–116. https://doi.org/10.1145/1639949.1640097

Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. 2010. Concurrent collections. *Scientific Programming* 18, 3-4 (2010), 203–217.

Developers CAFSite. [n. d.]. CAF repo. https://www.actor-framework.org

Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. 51–61. https://doi.org/10.1145/2093157.2093165

Luke Cheeseman, Matthew Parkinson, Slyvan Clebsch, Marios Kogias, Sophia Drossopoulou, David Chisnall, Tobias Wrigstad, and Paul Liétar. 2023. Artifact for "When Concurrency Matters: Behaviour Oriented Concurrency". https://doi.org/10.5281/zenodo.8320212

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness is Unique Enough. In *European Conference of Object Oriented Programming*. https://doi.org/10.1007/978-3-540-45070-2_9

S Clebsch. 2018. *Pony: Co-designing a Type System and a Runtime*. Ph. D. Dissertation. Ph. D. thesis, Imperial College London.

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for safe, fast, actors. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 1–10. https://doi.org/10.1145/2824815.2824816

S. Conchon and F. Le Fessant. 1999. Jocaml: mobile agents for Objective-Caml. In *Proceedings. First and Third International Symposium on Agent Systems Applications, and Mobile Agents*. 22–29. https://doi.org/10.1109/ASAMA.1999.805390

Joeri De Koster. 2015. *Domains: Language abstractions for controlling shared mutable state in actor systems*. Ph. D. Dissertation. PhD thesis, Vrije Universiteit Brussel.

Joeri De Koster, Tom Van Cutsem, and Theo D'Hondt. 2012. Domains: Safe sharing among actors. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*. 11–22. https://doi.org/10.1145/2414639.2414644

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 287–300. https://doi.org/10.1145/2429069.2429104

Julian Dolby, Cristian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. 2012. A Data-Centric Approach to Synchronization. *TOPLAS* (2012). https://doi.org/10.1145/2160910.2160913

Kapali Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. trainer. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633. https://doi.org/10.1145/360363.360369

John Field and Carlos Varela. 2005. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *POPL*. https://doi.org/10.1145/1040305.1040322

Cédric Fournet and Georges Gonthier. 1996. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 372–385. https://doi.org/10.1145/237721.237805

Cédric Fournet and Georges Gonthier. 2000. The join calculus: A language for distributed mobile programming. In *International Summer School on Applied Semantics*. Springer, 268–332. https://doi.org/10.1007/3-540-45699-6_6

Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Yu Xia, Runtian Zhou, and Dahlia Malkhi. 2022. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. *arXiv preprint arXiv:2203.06871* (2022). https://doi.org/10.1145/3572848.3577524

Laure Gonnord, Ludovic Henrio, Lionel Morel, and Gabriel Radanne. 2022. A Survey on Parallelism and Determinism. *ACM Computing Surveys (CSUR)* (2022). https://doi.org/10.1145/3564529

Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. *ACM SIGPLAN Notices* 47, 10 (2012), 21–40. https://doi.org/10.1145/2398857.2384619

Philip Haller and Martin Odersky. 2009. Scala actors: Unifying thread-based and actor-based programming. (2009), 202–220. https://doi.org/10.1016/j.tcs.2008.09.019

Tim Harris, Simon Marlow, and Simon Peyton Jones. 2005. Composable Memory Transactions. In *PPoPP*. ACM Press, 48–60. https://doi.org/10.1145/1065944.1065952

James W. Havender. 1968. Avoiding deadlock in multitasking systems. *IBM systems journal* 7, 2 (1968), 74–84. https://doi.org/10.1147/sj.72.0074

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. 92–101. https://doi.org/10.1145/872035.872048

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, Vol. 3. Stanford Research Institute Menlo Park, CA, 235.

Raphael Hiesgen, Dominik Charousset, and Thomas Schmidt. 2016. Reconsidering reliability in distributed actor systems. 31–32. https://doi.org/10.1145/2984043.2989218

Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2022. Opportunities for optimism in contended main-memory multicore transactions. *The VLDB Journal* (2022), 1–23. https://doi.org/10.14778/3377369.3377373

Shams M Imam and Vivek Sarkar. 2014. Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*. 67–80. https://doi.org/10.1145/2687357.2687368

Dan Kegel. 2014. The C10K problem. http://www.kegel.com/c10k.html

Eric Koskinen and Maurice Herlihy. 2008. Checkpoints and continuations instead of nested transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. 160–168. https://doi.org/10.1145/1378533.1378563

Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. 2022. Data-Parallel Actors: A Programming Model for Scalable Query Serving Systems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1059–1074. https://www.usenix.org/conference/nsdi22/presentation/kraft

Tongping Liu, Charlie Curtsinger, and Emery D Berger. 2011. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 327–336. https://doi.org/10.1145/2043556.2043587

Alexander Matveev and Nir Shavit. 2012. Towards a fully pessimistic stm model. https://doi.org/10.1145/2486159.2486166

Michael D. McCool, Arch D. Robison, and James Reinders. 2012. Structured parallel programming patterns for efficient computation. https://doi.org/10.1145/2382756.2382773

John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (feb 1991), 21–65. https://doi.org/10.1145/103727.103729

Katherine F. Moore and Dan Grossman. 2008. High-level small-step operational semantics of transactions. In *POPL*. ACM Press. https://doi.org/10.1145/1328438.1328448

Derek Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (proceedings of the 24th acm symposium on operating systems principles (sosp) ed.). ACM. https://doi.org/10.1145/2517349.2522738

Yang Ni, Vijay S Menon, Ali-Reza Adl-Tabatabai, Antony L Hosking, Richard L Hudson, J Eliot B Moss, Bratin Saha, and Tatiana Shpeisman. 2007. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 68–78. https://doi.org/10.1145/1229428.1229442

James Noble, Dave Clarke, and John Potter. 1998. Object Ownership for Dynamic Alias Protection. In *OOPSLA*. https://doi.org/10.1109/tools.1999.809424

Hubert Plociniczak and Susan Eisenbach. 2010. JErlang: Erlang with joins. In *International Conference on Coordination Languages and Models*. Springer, 61–75. https://doi.org/10.1007/978-3-642-13414-2_5

Developers PonySite. [n. d.]. Pony Github Repo. https://github.com/ponylang/

Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 180–194. https://doi.org/10.1145/3477132.3483591

Bo Sang, Patrick Eugster, Gustavo Petri, Srivatsan Ravi, and Pierre-Louis Roman. 2020. Scalable and Serializable Networked Multi-Actor Programming. *Proc. ACM Program. Lang.*, Article 198 (nov 2020), 30 pages. https://doi.org/10.1145/3428266

Bo Sang, Gustavo Petri, Masoud Saeida Ardekani, Srivatsan Ravi, and Patrick Eugster. 2016. Programming Scalable Cloud Services with AEON. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*. ACM, 16. https://doi.org/10.1145/2988336.2988352

Jun Shirako, David M Peixotto, Vivek Sarkar, and William N Scherer. 2008. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*. 277–288. https://doi.org/10.1145/1375527.1375568

Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. 2021. Chocola: Composable Concurrency Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 4 (2021), 1–56. https://doi.org/10.1145/3427201

Samira Tasharofi, Peter Dinges, and Ralph E Johnson. 2013. Why do scala developers mix the actor model with other concurrency models?. In *ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27*. Springer, 302–326. https://doi.org/10.1007/978-3-642-39038-8_13

Carlos Varela and Gul Agha. 2001. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices* 36, 12 (2001), 20–34. https://doi.org/10.1145/583960.583964

Mandana Vaziri, Frank Tip, and Julian Dolby. 2006. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 334–345. https://doi.org/10.1145/1111037.1111067

David W Walker. 1994. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Comput.* 20, 4 (1994), 657–673. https://doi.org/10.1016/0167-8191(94)90033-7 Message Passing Interfaces.

Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. 2008. Irrevocable transactions and their applications. In *SPAA*. https://doi.org/10.1145/1378533.1378584

WhatsApp. 2012. 1 million is so 2011. https://blog.whatsapp.com/1-million-is-so-2011

Received 2023-04-14; accepted 2023-08-27