

FlowPools: Lock-Free Deterministic Concurrent Data-Flow Queues

Authors

EPFL, Switzerland
`{firstname.lastname}@epfl.ch`
<http://lamp.epfl.ch>

Abstract. Implementing correct and deterministic parallel programs is hard (additional motivation sentence after this). We present the design and implementation of a fundamental data structure for deterministic parallel data-flow computation. Additionally, we provide a proof of correctness, showing that the implementation is linearizable, lock-free, and deterministic. Finally, we provide microbenchmarks which compare our *flow pools* against corresponding operations on other popular concurrent data structures, in addition to performance benchmarks on a real *XYZ* application using real data. (Keep abstract between 70 and 150 words).

Keywords: data-flow, concurrent data-structure, determinism

1 Introduction

1.1 Motivation

- we want a deterministic model - we do not block in the programming model (i.e. there are no operations which cause blocking until a value becomes available) - we want a non-blocking data-structure (i.e. the operations on the data-structures should themselves be non-blocking) - programs run indefinitely => we need to GC parts of the data structure we no longer need - we want to reduce heap allocation and inline the datastructure as much as possible => lower memory consumption, better cache behaviour and fewer GC cycles

Obligatory multicore motivation paragraph.

Lock-free is better, and why.

Introduction and motivation for data-flow programming model.

2 Model of Computation

Producer-consumer parallelism. Description and image of queue/stream of values, producer, and multiple consumers.

3 Programming Model

The FlowPool suite supports the following operations:

- `Builder.<<(x: T): Builder`
Inserts an element into the underlying FlowPool.
- `Builder.seal(n: Int): Unit`
Seals the underlying FlowPool at `n` elements. The need for the size argument is explained below. A sealed FlowPool may only contain `n` elements. This allows for callback cleanup and termination.
- `FlowPool.doForAll(f: T => Unit): Future[Int]`
Instructs the FlowPool to execute the closure `f` exactly once for each element inserted into the FlowPool (asynchronously). The returned future contains the number of elements in the FlowPool and completes once `f` has been executed for all elements.
- `FlowPool.mappedFold[U, V <: U](acc: V)(cmb: (U,V) => V)(map: T => U): Future[(Int, V)]` Reduces the FlowPool to a single value of type `V`, by first mapping each element to an internal representation using `map` and then aggregating using `cmb`. No guarantee is given about synchronization or order. Returns a future containing a tuple with number of elements in the FlowPool and the aggregated value.
- `FlowPool.builder`
Returns a builder for this FlowPool.
- `Future.map[U](f: T => U)`
Maps this future to another future executing the function `f` exactly once when the first future completes.
- `future[T](f: () => T): Future[T]`
Asynchronously dispatch execution of `f` and return a future with its result.

Determinism of seal We will show that the final size of the FlowPool is required as an argument to the `seal` method in order to satisfy the determinism property of the FlowPool. Look at the following program:

```
val p = new FlowPool[Int]()
val b = p.builder

future {
  for (i <- 1 to 10) { b << i }
  b.seal
}

future { for (i <- 1 to 10) { b << i } }
```

Depending on which for-loop completes first, this program completes successfully or yields an error. A similar program with `b.seal(20)` will always succeed.

Generators In the following we'll present a couple of generators for FlowPools based on common generators in the Scala standard library.

```
def iterate[T](start: T, len: Int)(f: (T) => T) = {
  val p = new FlowPool[T]
  val b = p.builder
  future {
    var e = start
    for (i <- 1 to len) { b << e; e = f(e) }
    b.seal(len)
  }; p
}
```

```
def tabulate[T](n: Int)(f: (Int) => T) = {
  val p = new FlowPool[T]
  val b = p.builder
  future {
    for (i <- 0 to (n-1)) {
      b << f(i)
    }
    b.seal(n)
  }; p
}
```

```
def fill[T](n: Int)(elem: => T) = {
  val p = new FlowPool[T]
  val b = p.builder
  future {
    for (i <- 1 to n) { b << elem }
    b.seal(n)
  }; p
}
```

Monadic Operations In the following we'll present some monadic operations on top of the basic FlowPool operations. This will also show some use-cases of the futures as result type of `doForAll`.

```
def map[S](f: T => S) = {
  val fp = new FlowPool[S]
  val b = fp.builder
  doForAll { x =>
    b << f(x)
  } map { b.seal _ }
  fp
}
```

```

def filter(f: T => Boolean) = {
  val fp = new FlowPool[T]
  val b = fp.builder

  mappedFold(0)(_ + _) { x =>
    if (f(x)) { b << x; 1 } else 0
  } map { case (c,fc) => b.seal(fc) }

  fp
}

def flatMap[S](f: T => FlowPool[S]) = {
  val fp = new FlowPool[S]
  val b = fp.builder

  mappedFold(future(0))(_ <+> _) { x =>
    f(x).doForAll(b << _)
  } map { case (c,cfut) => cfut.map(b.seal _) }

  fp
}

```

where `<+>` is the future of the sum of two `Future[Int]`.

4 Implementation

5 Proofs

5.1 Abstract Pool Semantics

Alex

5.2 Linearizability

Alex

5.3 Lock-Freedom

5.4 Determinism

6 Experimental Results

7 Related Work

Things to probably cite: Oz, gpars, Java CLQ, our futures writeup.

Things we should probably have a look at: Microsoft TPL, Dataflow Java, FlumeJava...

Forcing a bib, [?], [?], [?], [?], [?], [?]

8 Conclusion

```

1  def append(elem: Elem)
2    b = READ(current)
3    idx = READ(b.index)
4    nextobj = READ(b.array(idx + 1))
5    curobj = READ(b.array(idx))
6    if (check(b, idx, curobj, nextobj))
7      if (CAS(b.array(idx + 1), nextobj, curobj))
8        if (CAS(b.array(idx), curobj, elem))
9          WRITE(b.index, idx + 1)
10         invokeCallbacks(elem, curobj)
11       else append(elem)
12     else append(elem)
13   else
14     advance()
15     append(elem)
16
17
18 def check(b: Block, idx: Int, curobj: Object, nextobj: Object)
19   // The check on the index is done implicitly in the real code
20   if (idx > LASTELEMPOS) return false
21   else curobj match
22     elem: Elem =>
23       return false
24     term: Terminal =>
25       if (term.sealed == NOTSEALED) return true
26     else
27       if (totalElems(b, idx) < term.sealed) return true
28       else error("sealed")
29
30   null =>
31     error("unreachable")
32
33
34 def advance()
35   b = READ(current)
36   idx = READ(b.index)
37   if (idx > LASTELEMPOS) expand(b)
38   else
39     obj = READ(b.array(idx))
40     if (obj is Elem) WRITE(b.index, idx + 1)
41
42
43 def expand(b: Block)
44   nb = READ(b.next)
45   if (nb is null)
46     nb = createBlock(b.blockindex + 1)
47     if (CAS(b.next, null, nb))
48       CAS(current, b, nb)
49   else
50     CAS(current, b, nb)
51
52
53 def totalElems(b: Block, idx: Int)
54   return b.blockindex * (BLOCKSIZE - 1) + idx
55
56 def invokeCallbacks(elem: Elem, term: Terminal)
57   for (f <- term.callbacks) future
58     f(elem)
59

```

Definition 1 (FlowPool). The **FlowPool** is defined as the references *start* and *current*. The **FlowPool state** is defined as the configuration of objects transitively reachable from the reference *start*.

We define the following relations:

$$following(b : Block) = \begin{cases} \emptyset & \text{if } b.next = null, \\ b.next \cup following(b.next) & \text{otherwise} \end{cases}$$

$$reachable(b : Block) = \{b\} \cup following(b)$$

$$last(b : Block) = b' : b' \in reachable(b) \wedge b'.next = null$$

$$size(b : Block) = |\{x : x \in b.array \wedge x \in Elem\}|$$

We say that the FlowPool **has** an element *e* at some time t_0 if and only if the relation $hasElem(start, e)$ holds.

We say that a callback *f* in a FlowPool **will be called** for the element *e* at some time t_0 if and only if the relation $willBeCalled(start, e, f)$ holds.

We say that the FlowPool is **sealed** at the size *s* at time t_0 if and only if the relation $sealedAt(start, s)$ holds.

TODO formal definitions of these relations (involving the flowpool datatypes)

FlowPool operations are **append**, **foreach** and **seal**, and are defined by pseudocodes in figures ...

Definition 2 (Invariants). We define the following invariants for the **Flow-Pool**:

INV1 $start = b : Block, b \neq null, current \in reachable(start)$

INV2 $\forall b \in reachable(start), b \notin following(b)$

INV3 $\forall b \in reachable(start), b \neq last(start) \Rightarrow size(b) = LASTELEMPOS \wedge b.array(BLOCKSIZE - 1) \in Terminal$

INV4 $\forall b = last(start), b.array = p \cdot t \cdot n$, where:

$$p = X^P, t = t_1 \cdot t_2, n = null^N$$

$$x \in Elem, t_1 \in Terminal, t_2 \in \{null\} \cup Terminal$$

$$P + N + 2 = BLOCKSIZE$$

INV5 $\forall b \in reachable(start), b.index > 0 \Rightarrow b.array(b.index - 1) \in Elem$

Definition 3 (Validity). A FlowPool state \mathbb{S} is **valid** if and only if the invariants [INV1-5] hold for that state.

Definition 4 (Abstract pool). An **abstract pool** \mathbb{P} is a function from time *t* to a tuple of sets (*elems*, *callbacks*, *seal*) such that:

$$seal \in \{\emptyset, \{w\}\}, w \in \mathbb{N}$$

$$callbacks \subset \{(f : Elem \Rightarrow Unit, called)\}$$

$$called \subseteq elems \subseteq Elem$$

We say that an abstract pool \mathbb{P} **is in state** $(elems, callbacks, seal)$ at time t if and only if $\mathbb{P}(t) = (elems, callbacks, seal)$.

Definition 5 (Abstract pool operations). We say that an abstract pool operation op applied to an abstract pool $P_0 = (elems_0, callbacks_0, seal_0)$ at some time t **changes** the state of the abstract pool to $P = (elems, callbacks, seal)$ if $\exists t_0, \forall \tau, t_0 < \tau < t, \mathbb{P}(\tau) = P_0$ and $\mathbb{P}(t) = P$.

Abstract pool operation $foreach(f)$ changes the state at t_0 from $(elems, callbacks, seal)$ to $(elems, (f, \emptyset) \cup callbacks, seal)$. Furthermore:

$\exists t_1 > t_0, \forall t_2 > t_1, \mathbb{P}(t_2) = (elems_2, (f, called_2) \cup callbacks_2, seal_2)$, where:
 $called_2 \subseteq elems_2 \subseteq elems$

Append...

Seal...

Definition 6 (Consistency). A FlowPool state \mathbb{S} is **consistent** with an abstract pool $\mathbb{P} = (elems, callbacks, seal)$ at t_0 if and only if \mathbb{S} is a valid state and:

$\forall e \in Elem, hasElem(start, e) \Leftrightarrow e \in elems$
 $\forall f \in Elem \Rightarrow Unit, \forall e \in Elem, willBeCalled(start, e, f) \Leftrightarrow \exists t_1 \geq t_0, \mathbb{P}(t_1) = (elems_1, (f, called_1) \cup callbacks_1, seal_1), elems \subseteq called_1$
 $\forall s \in \mathbb{N}, sealedAt(start, s) \Leftrightarrow s \in seal$

A FlowPool operation op completing at some time t_0 is consistent with an abstract pool operation op' if and only if op changes the state of the FlowPool from \mathbb{S}_1 to \mathbb{S}_2 , where \mathbb{S}_1 and \mathbb{S}_2 are consistent with the abstract pool states \mathbb{A}_1 and \mathbb{A}_2 , respectively, and op' changes the state of the abstract pool from \mathbb{A}_1 to \mathbb{A}_2 .

Theorem 1 (Abstract pool semantics). FlowPool operation **create** creates a new FlowPool consistent with the abstract pool $\mathbb{P} = (\emptyset, \emptyset, \emptyset)$. FlowPool operations **foreach**, **append** and **seal** are consistent with the abstract pool semantics.

Lemma 1. Given a FlowPool in a valid state, all writes produce a FlowPool in a valid state.

Proof.

Definition 7 (Transition). If for a function $f(t)$ there exist times t_0 and t_1 such that $\forall t, t_0 < t < t_1, f(t) = v_0$ and $f(t_1) = v_1$, then we say that the function f goes through a transition at t_1 . We denote this as:

$f : v_0 \xrightarrow{t_1} v_1$

Or, if we don't care about the exact time t_1 , simply as:

$f : v_0 \rightarrow v_1$

Lemma 2 (Lifecycle). For all blocks $b \in reachable(start)$, and for all $x \in b.array$, function x goes through and only through the prefix of the following transitions:

$null \rightarrow cb_1 \rightarrow \dots \rightarrow cb_n \rightarrow elem$, where:

$cb_i \in Terminal, i \neq j \Rightarrow cb_i \neq cb_j, elem \in Elem$

Proof. First of all, it is obvious from the code that each block that becomes an element of $reachable(start)$ at some time t_0 has the value of all $x \in b.array$ set to *null*.

Lemma 3. *Given a FlowPool in state \mathbb{S} consistent with some abstract pool state \mathbb{A} , CAS instructions in lines 47, 48 and 50 do not change the state of the abstract pool.*

Proof.

Lemma 4. *Given a FlowPool in state \mathbb{S} consistent with some abstract pool state \mathbb{A} , a successful CAS in line 8 at some time t_0 changes the state of the FlowPool to \mathbb{S}_0 consistent with an abstract pool state \mathbb{A}_0 , such that:*

$$\mathbb{A} = (elems, callbacks, seal)$$

$$\mathbb{A}_0 = (\{elem\} \cup elems, callbacks, seal)$$

*Furthermore, given a fair scheduler, there exists a time $t_1 > t_0$ such that all the callback functions are called for *elem*.*

Proof.