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. Aditionally, 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` an 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

```
def append(elem: Elem)
  b = READ(current)
  idx = READ(b.index)
  nextobj = READ(b.array(idx + 1))
  curobj = READ(b.array(idx))
  if (check(b, idx, curobj, nextobj))
    if (CAS(b.array(idx + 1), nextobj, curobj))
      if (CAS(b.array(idx), curobj, elem))
        WRITE(b.index, idx + 1)
        invokeCallbacks(elem, curobj)
       else append(elem)
     else append(elem)
   else
    advance()
    append(elem)


def check(b: Block, idx: Int, curobj: Object, nextobj: Object)
  // The check on the index is done implicitly in the real code
  if (idx > LASTELEMPOS) return false
  else curobj match
    elem: Elem =>
      return false
    term: Terminal =>
      if (term.sealed == NOTSEALED) return true
      else
        if (totalElems(b, idx) < term.sealed) return true
        else error("sealed")

    null =>
      error("unreachable")


def advance()
  b = READ(current)
  idx = READ(b.index)
  if (idx > LASTELEMPOS) expand(b)
  else
    obj = READ(b.array(idx))
    if (obj is Elem) WRITE(b.index, idx + 1)


def expand(b: Block)
  nb = READ(b.next)
  if (nb is null)
    nb = createBlock(b.blockindex + 1)
    if (CAS(b.next, null, nb))
      expand(b)
   else
    CAS(current, b, nb)


def totalElems(b: Block, idx: Int)
  return b.blockindex * (BLOCKSIZE - 1) + idx

def invokeCallbacks(elem: Elem, term: Terminal)
  for (f <- term.callbacks) future
    f(elem)


def seal(size: Int)
  b = READ(current)
  idx = READ(b.index)
  if (idx <= LASTELEMPOS)
    curobj = READ(b.array(idx))
```

```
    curobj match
      term: Terminal =>
        tryWriteSeal(term, b, idx, size)
      elem: Elem =>
        WRITE(b.index, idx + 1)
        seal(size)
      null =>
        error("unreachable")

   else
    expand(b)
    seal(size)


def tryWriteSeal(term: Terminal, b: Block, idx: Int, size: Int)
  val total = totalElems(b, idx)
  if (total > size) error("too many elements")
  if (term.sealed == NOTSEALED)
    nterm = new Terminal
      sealed = size
      callbacks = term.callbacks

    CAS(b.array(idx), term, nterm)
   else if (term.sealed != size)
    error("already sealed with different size")


def doForAll(f: Elem => Unit)
  future
    asyncDoForAll(f, start, 0)


def asyncDoForAll(f: Elem => Unit, b: Block, idx: Int)
  if (idx <= LASTELEMPOS)
    obj = READ(b.array(idx))
    obj match
      term: Terminal =>
        nterm = new Terminal
          sealed = term.sealed
          callbacks = f :: term.callbacks

        if (!CAS(b.array(idx), term, nterm)) asyncDoForAll(f, b, idx)
      elem: Elem =>
        f(elem)
        asyncDoForAll(f, b, idx + 1)
      null =>
        error("unreachable")

   else
    // In the real code we take a shortcut when preparing the new block
    expand(b)
    asyncDoForAll(f, b.next, 0)
```

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:

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

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

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

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

*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 tome 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 **FlowPool**:*

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:*

$seal \in \{\emptyset, \{w\}\}, w \in \mathbb{N}$
$callbacks \subset \{(f : Elem => Unit, called)\}$
$called \subseteq elems \subseteq Elem$

*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:*

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

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* $\mathbb{A}_2$.

Proposition 1. *Every valid state is consistent with some abstract pool.*

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 (End of life). *For all blocks* $b \in reachable(start)$, *if value* $v \in Elem$ *is written to b.array at some position idx at some time* t_0, *then* $\forall t > t_0, b.array(idx) = v$.

Proof. The CAS in line 8 is the only CAS which writes an element. No other CAS has a value of type *Elem* as the expected value. This means that once the CAS in line 8 writes a value of type *Elem*, no other write can change it.

Lemma 2 (Valid hint). *For all blocks* $b \in reachable(start)$, *if* $b.index > 0$ *at some time* t_0, *then* $b.array(b.index - 1) \in Elem$ *at time* t_0.

Proof. Observe every write to *b.index* – they are all unconditional. However, at every such write occurring at some time t_1 that writes some value *idx* we know that some previous value at *b.array* entry $idx-1$ at some time $t_0 < t_1$ was of type *Elem*. Hence, from lemma 1 it follows that $\forall t \geq t_1, b.array(idx - 1) \in Elem$.

Corollary 1 (Compactness). *For all blocks* $b \in reachable(start)$, *if for some idx* $b.array(idx) \in Elem$ *at time* t_0 *then* $b.array(idx - 1) \in Elem$ *at time* t_0. *This follows directly from the lemmas 1 and 2, and the fact that the CAS in line 8 only writes to array entries idx for which it previously read the value from b.index.*

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 3 (Freshness). *For all blocks $b \in reachable(start)$, and for all $x \in b.array$, function x has unique elements in its string of transitions.*

Proof. CAS instruction in line 8 writes a value of type $Elem$. No CAS instruction has a value of type $Elem$ as the expected value.

Trivial analysis of CAS instructions in lines 88 and 107, shows that their expected values are of type $Terminal$ or are $null$. Their new values are always freshly allocated.

The more difficult part is to prove that CAS instruction in line 7 respects the statement of the lemma.

Since the CAS instructions in lines 88 and 107 are preceeded by a read of $idx = b.index$, from 2 it follows that $b.array(idx - 1)$ contains a value of type $Elem$. These are also the only CAS instructions which replace a $Terminal$ value with another $Terminal$ value. The new value is always unique, as shown above.

A successful CAS in line 7 overwrites a value cb_0 at $idx + 1$ read in line 4 at t_0 with a new value cb_2 at time t_2. Value cb_2 was read in line 5 at t_1 from the entry idx. The string of transitions of values at idx is composed of unique values at least since t_1, since there is a value of type $Elem$ at the index $idx - 1$.

Now assume that a thread T1 successfully CAS in line 7 at $idx + 1$ at time t_2 overwrites cb_0 with a value cb_2 read from idx at t_1, and that another thread T2 subsequently successfully completes the CAS in line 7 at $idx + 1$ at time t_2 with a value cb_{prev} which was at $idx + 1$ at some time $t < t_0$. That would mean that the thread T2 read the value cb_{prev} in line 5 at some time $t_{prev1} < t_1$ and successfully completed the CAS at time $t_{prev2} > t_2$. If the CAS was successful, then the read in line 4 by T2 occured at $t_{prev0} < t_{prev1} < t_1$. TODO finish this

Lemma 4 (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:*

$null \rightarrow cb_1 \rightarrow \cdots \rightarrow cb_n \rightarrow elem$, *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 $null$.

Next, we inspect all the CAS instructions that operate on entries of $b.array$.

The CAS in line 8 has a value $curobj \in Terminal$ as an expected value and writes an $elem \in Elem$. This means that the only transition that this CAS can cause is of type $cb_i \in Terminal \rightarrow elem \in Elem$.

We will now prove that the CAS in line 7 at time t_2 is successful if and only if the entry at $idx+1$ is *null* or $nextobj \in Terminal$. We know that the entry at $idx+1$ does not change $\forall t, t_0 < t < t_2$, where t_0 is the read in line 4, because of lemma 3 and the fact that CAS in line 7 is assumed to be successful. We know that during the read in line 5 at time t_1, such that $t_0 < t_1 < t_2$, the entry at idx was $curobj \in Terminal$, by trivial analysis of the `check` procedure. It follows from corollary 1 that the array entry $idx+1$ is not of type *Elem* at time t_1, otherwise array entry idx would have to be of type *Elem*. Finally, we know that the entry at $idx+1$ has the same value during the interval $\langle t_1, t_2 \rangle$, so its value is not *Elem* at t_2.

The above reasoning shows that the CAS in line 7 always overwrites a one value of type $Terminal$ (or *null*) with another value of type $Terminal$. We still have to show that it never overwrites the value cb_0 with a value cb_2 that was at $b.array(idx)$ at an earlier time. This follows from the 3 directly.

Finally, note that the statement for CAS instructions in lines 88 and 107 follows directly from the proof for lemma 3.

Lemma 5 (Valid writes). *Given a FlowPool in a valid state, all writes in all operations produce a FlowPool in a valid state.*

Proof. A new FlowPool is trivially in a valid state.

Otherwise, assume that the FlowPool is in a valid state $\mathbb{S}$. We analyze each write.

Lemma 6 (Housekeeping). *Given a FlowPool in state $\mathbb{S}$ consistent with some abstract pool state $\mathbb{A}$, CAS instructions in lines 7, 47 and 50 do not change the state of the abstract pool.*

Proof.

Lemma 7 (Append correctness). *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.

Definition 8 (Obstruction-freedom). *Given a FlowPool in a valid state, any operation*

Lemma 8 (Obstruction-free operations). *The FlowPool operations are obstruction-free.*

Proof. By trivial sequential code analysis.