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, [1], [2], [3], [4], [5], [6]

8 Conclusion

References

1. M. Bowman, S. K. Debray, and L. L. Peterson. Reasoning about naming systems. *ACM Trans. Program. Lang. Syst.*, 15(5):795–825, November 1993.
2. J. Braams. Babel, a multilingual style-option system for use with latex's standard document styles. *TUGboat*, 12(2):291–301, June 1991.
3. M. Clark. Post congress tristesse. In *TeX90 Conference Proceedings*, pages 84–89. TeX Users Group, March 1991.
4. M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.
5. L. Lamport. *LaTeX User's Guide and Document Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
6. S. Salas and E. Hille. *Calculus: One and Several Variable.* John Wiley and Sons, New York, 1978.

A Proof of Correctness

Definition 1 (Data types). A **Block** b is an object which contains an array $b.array$, which itself can contain elements, $e \in Elem$, where **Elem** represents the type of e and can be any countable set. A given block b additionally contains an index $b.index$ which represents an index location in $b.array$, a unique index identifying the array $b.blockIndex$, and $b.next$, a reference to a successor block c where $c.blockIndex = b.blockIndex + 1$. A **Terminal** $term$ is a sentinel object, which contains an integer $term.sealed \in -1 \cup \mathbb{N}$, and $term.callbacks$, a set of functions $f \in Elem \Rightarrow Unit$.

Definition 2 (FlowPool). A **FlowPool** $pool$ is an object that has a reference $pool.start$, to the first block b_0 (with $b_0.blockIndex = 0$), as well as a reference $pool.current$ typically pointing to some subsequent block b_n where $b_n = b_0$ or where b_n is reachable from b_0 following $next$ references. Initially, $pool.current = pool.start$. The pool **state** $\mathbb{S}$ is defined as the sequence of blocks reachable from $pool.start$ by following $next$ references within blocks. More formally, the relation $reachable(b, c)$ on two blocks b, c holds iff $b = c \ \vee \ b.next = c \ \vee \ \exists b' \ : \ reachable(b, b') \ \wedge \ reachable(b', c)$. A **state changing** instruction is any atomic write or CAS instruction that changes an object that can be accessed from $pool.start$.

Definition 3 (Invariants).

INV1 let $b : Block$ such that $reachable(start, b) \wedge b.next = \perp$. $\exists i$ such that $b.array(i) = Terminal \wedge \forall \ i < j \leq LASTELEMPOS$, $b.array(j) = \perp$

Definition 4 (Abstract state). An **abstract state** $\mathbb{A}$ is a tuple $(elems, sealed, callbacks)$ such that $\mathbb{A} \in \{(elems, sealed, callbacks) \mid elems \subset Elem, \ sealed \in \{-1\} \cup \mathbb{N}, \ callbacks \subset Elem \Rightarrow Unit\}$ Abstract state operations on some abstract state $\mathbb{A}$ are $append(\mathbb{A}, e) = \mathbb{A}'$ where $\mathbb{A}' = (elems \cup \{e\}, sealed, callbacks)$ if $\mathbb{A} = (elems, sealed, callbacks)$, $seal(\mathbb{A}, sealSize) = \mathbb{A}''$ where $\mathbb{A}'' = (elems, sealSize, callbacks) \ : \ sealSize \in \mathbb{N}$ if $\mathbb{A} = (elems, sealed, callbacks)$ and $sealed = -1$, the unsealed state, or

$sealed = sealSize$ already, and $doForAll(\mathbb{A}, fun) = \mathbb{A}'''$ where $\mathbb{A}''' = (elems, sealed, callbacks \cup \{fun\})$.

Definition 5 (Consistency). A FlowPool state $\mathbb{S}$ of *pool* with starting block *pool.start* is consistent with an abstract state $\mathbb{A} = (elems, sealed)$ iff some element $e \in elems \Leftrightarrow \exists b, i \ : \ reachable(pool.start, b) \ \wedge \ b.array(i) = e$, and $\exists c, j \ : \ c.array(j) \in Terminal \wedge c.array(j).sealed = sealed \wedge reachable(pool.start, c)$. A FlowPool operation *op* is **consistent** with the corresponding abstract state operation op' iff $\mathbb{S}' = op(\mathbb{S})$ is consistent with an abstract state $\mathbb{A}' = op'(\mathbb{A})$. A **consistency change** is a change from state $\mathbb{S}$ to state $\mathbb{S}'$ such that $\mathbb{S}$ is consistent with an abstract state $\mathbb{A}$ and $\mathbb{S}'$ is consistent with an abstract set $\mathbb{A}'$, where $\mathbb{A} \neq \mathbb{A}'$.

Definition 6 (Lock-freedom). In a scenario where some finite number of threads are executing a concurrent operation, that concurrent operation is *lock-free* if and only if that concurrent operation is completed after a finite number of steps by some thread.

Theorem 1 (Lock-freedom). *FlowPool operations append, seal, and doForAll are lock-free.*

We begin by first proving that there are a finite number of execution steps before a consistency change occurs.

By Lemma 5, after invoking *append*, a consistency change occurs after a finite number of steps. Likewise, by Lemma 7, after invoking *seal*, a consistency change occurs after a finite number of steps. And finally, by Lemma 8, after invoking $doForAll$, a consistency change likewise occurs after a finite number of steps.

By Lemma 10, this means a concurrent operation *append*, *seal*, or $doForAll$ will successfully complete. Therefore, by Definition 6, these operations are lock-free.

Lemma 1. *After invoking an operation op, if non-consistency changing CAS operations $CAS1$, $CAS3$, or $CAS4$, in the pseudocode fail, they must have already been successfully completed by another thread since op began.*

Proof. Trivial inspection of the pseudocode reveals that since $CAS1$ makes up a check that precedes $CAS2$, and since $CAS2$ is the only operation besides $CAS1$ which can change the expected value of $CAS1$, in the case of a failure of $CAS1$, $CAS2$ (and thus $CAS1$) must have already successfully completed or $CAS1$ must have already successfully completed by a different thread since *op* began executing.

Likewise, by trivial inspection $CAS3$ is the only CAS which can update the *b.next* reference, therefore in the case of a failure, some other thread must have already successfully completed $CAS3$ since the beginning of *op*.

Like above, $CAS4$ is the only CAS which can change the *current* reference, therefore in the case of a failure, some other thread must have already successfully completed $CAS4$ since *op* began. □

Lemma 2. *Invoking the expand operation will execute a non- consistency changing instruction after a finite number of steps. Moreover, it is guaranteed* *that the current reference is updated to point to a subsequent block after a finite number of steps. Finally, expand will return after a finite number of steps. .*

Proof. From inspection of the pseudocode, it is clear that the only point at which $expand(b)$ can be invoked is under the condition that for some block b, $b.index > LASTELEMPOS$, where $LASTELEMPOS$ is the maximum size set aside for elements of type $Elem$ in any block. Given this, we will proceed by showing that a new block will be created with all related references $b.next$ and $current$ correctly set.

There are two conditions under which a non-consistency changing CAS instruction will be carried out.

- **Case 1:** if $b.next = \bot$, a new block nb will be created and $CAS3$ will be executed. From Lemma 1, we know that $CAS3$ must complete successfully on some thread. Afterwards recursively calling *expand* on the original block b.
- **Case 2:** if $b.next \neq \bot$, $CAS4$ will be executed. Lemma 1 guarantees that $CAS4$ will update *current* to refer to $b.next$, which we will show can only be a new block. Likewise, Lemma 1 has shown that $CAS3$ is the only state changing instruction that can initiate a state change at location $b.next$, therefore, since $CAS3$ takes place within Case 1, Case 2 can only be reachable after Case 1 has been executed successfully. Given that Case 1 always creates a new block, therefore, $b.next$ in this case, must always refer to a new block.

Therefore, since from Lemma 1 we know that both $CAS3$ and $CAS4$ can only fail if already completed guaranteeing their finite completion, and since $CAS3$ and $CAS4$ are the only state changing operations invoked through *expand*, the *expand* operation must complete in a finite number of steps.

Finally, since we saw in Case 2 that a new block is always created and related references are always correctly set, that is both $b.next$ and *current* are correctly updated to refer to the new block, it follows that $numBlocks$ strictly increases after some finite number of steps. □

Lemma 3. *After invoking append(elem), if $CAS2$ fails, then some thread has successfully completed $CAS2$ or $CAS5$ (or likewise, $CAS6$) after some finite number of steps.*

Proof. First, we show that a thread attempting to complete $CAS2$ can't fail due to a different thread completing $CAS1$ so long as *seal* has not been invoked after completing the read of $currobj$. We address this exception later on.

Since after *check*, the only condition under which $CAS1$, and by extension, $CAS2$ can be executed is the situation where the current object $currobj$ with index location idx is the $Terminal$ object, it follows that $CAS1$ can only ever serve to duplicate this $Terminal$ object at location $idx + 1$, leaving at most two $Terminal$s in block refered to by *current* momentarily until $CAS2$ can be executed. By Lemma 1, since $CAS1$ is a non-consistency changing instruction, it follows that any thread holding any element $elem'$ can execute this instruction without changing the expected value of $currobj$ in $CAS2$, as no new object is ever created and placed in location idx. Therefore, $CAS2$ cannot fail due to

$CAS1$, so long as *seal* has not been invoked by some other thread after the read of $currobj$.

This leaves only two scenarios in which consistency changing $CAS2$ can fail:

- **Case 1:** Another thread has already completed $CAS2$ with a different element $elem'$.
- **Case 2:** Another thread completes an invocation to the *seal* operation after the current thread completes the read of $currobj$. In this case, $CAS2$ can fail because $CAS5$ (or, likewise $CAS6$) might have completed before, in which case, it inserts a new $Terminal$ object *term* into location *idx* (in the case of a *seal* invocation, *term.sealed* $\in \mathbb{N}$, or in the case of a $doForAll$ invocation, $\{$*term.callbacks* $\in\}$).

We omit the proof and detailed discussion of $CAS6$ because it can be proven using the same steps as were taken for $CAS5$. □

Lemma 4. *All operations with the exception of append, seal, and* $doForAll$ *execute only a finite number of steps between each state changing instruction*

Proof. The **advance**, **check**, **totalElems**, **invokeCallbacks**, and **tryWriteSeal** operations have a finite number of execution steps, as they contain no recursive calls, loops, or other possibility to restart.

While the **expand** operation contains a recursive call following a CAS instruction, it was shown in Lemma 2 that an invocation of *expand* is guaranteed to execute a state changing instruction after a finite number of steps. □

Lemma 5. *After invoking append*(*elem*)*, a consistency changing instruction will be completed after a finite number of steps.*

Proof. The **append** operation can be restarted in three cases. We show that in each case, it's guaranteed to either complete in a finite number of steps, or leads to a state changing instruction:

- **Case 1:** The call to *check*, a finite operation by Lemma 4, returns $false$, causing a call to *advance*, also a finite operation by Lemma 4, followed by a recursive call to *append* with the same element *elem* which in turn once again calls *check*.
  We show that after a finite number of steps, the *check* will evaluate to *true*, or some other thread will have completed a consistency changing operation since the initial invocation of *append*. In the case where *check* evaluates to *true*, Lemma 3 applies, as it guarantees that a consistency changing CAS is completed after a finite number of steps.
  When the call to the finite operation *check* returns $false$, if the subsequent *advance* finds that a $Terminal$ object is at the current block index idx, then the next invocation of *append* will evaluate *check* to *true*. Otherwise, it must be the case that another thread has moved the Terminal to a subsequent index since the initial invocation of append, which is only possible using a consistency changing instruction.
  Finally, if *advance* finds that the element at *idx* is an *Elem*, by Lemma 9, *b.index* will be incremented after a finite number of steps. By $INV1$,

this can only happen a finite number of times until a $Terminal$ is found. In the case that *expand* is meanwhile invoked through *advance*, by Lemma 2 it's guaranteed to complete state changing instructions $CAS3$ or $CAS4$ in a finite number of steps. Otherwise, some other thread has moved the $Terminal$ to a subsequent index. However, this latter case is only possible by successfully completing $CAS2$, a consistency changing instruction, after the initial invocation of append.

- **Case 2:** $CAS1$ fails, which we know from Lemma 1 means that it must've already been completed by another thread, guaranteeing that $CAS2$ will be attempted. If $CAS2$ fails, by Lemma 3, after a finite number of steps, a consistency changing instruction will be completed. If $CAS2$ succeeds, as a consistency changing instruction, consistency will have clearly been changed.
- **Case 3:** $CAS2$ fails, which, by Lemma 3, indicates that either some other thread has already completed $CAS2$ with another element, or another consistency changing instruction, $CAS5$ or $CAS6$ has successfully completed.

Therefore, *append* itself as well as all other operations reachable via an invocation of *append* are guaranteed to have a finite number of steps between *consistency* changing instructions.

Lemma 6. *After invoking seal*$(size)$*, if* $CAS5$ *fails, then some thread has successfully completed* $CAS5$ *or* $CAS2$ *after some finite number of steps.*

Proof. Since $CAS1$ only duplicates an existing $Terminal$, it can not be the cause for a failing $CAS5$. This leaves only two cases in which $CAS5$ can fail:

- **Case 1:** Another thread has already completed $CAS5$.
- **Case 2:** Another thread completes an invocation to the *append*$(elem)$ operation after the current thread completes the read of $currobj$. In this case, $CAS5$ can fail because $CAS2$ might have completed before, in which case, it inserts a new *Elem* object *elem* into location *idx*.

Lemma 7. *After invoking seal*$(size)$*, a consistency changing instruction will be completed after a finite number of steps, or the initial invocation of seal*$(size)$ *completes.*

Proof. The *seal* operation can be restarted in two scenarios.

- **Case 1:** The check $idx \leq LASTELEMPOS$ succeeds, indicating that we are at a valid location in the current block b, but the object at the current index location *idx* is of type *Elem*, not $Terminal$, causing a recursive call to *seal* with the same size $size$.
  In this case, we begin by showing that the atomic write of $idx+1$ to $b.index$, required to iterate through the block b for the recursive call to $seal$, will be correctly incremented after a finite number of steps.
  Therefore, by both the guarantee that, in a finite number of steps, $b.index$ will eventually be correctly incremented as we saw in Lemma 9, as well as by $INV1$ we know that the original invocation of *seal* will correctly iterate through b until a $Terminal$ is found. Thus, we know that the call to $tryWriteSeal$ will be invoked, and by both Lemma 4 and Lemma 5, we

know that either $tryWriteSeal$, will successfully complete in a finite number of steps, in turn successfully completing $seal(size)$, or $CAS2$, another consistency changing operation will successfully complete.

- **Case 2:** The check $idx \leq LASTELEMPOS$ fails, indicating that we must move on to the next block, causing first a call to *expand* followed by a recursive call to *seal* with the same size $size$.
  We proceed by showing that after a finite number of steps, we must end up in Case 1, which we have just showed itself completes in a finite number of steps, or that a consistency change must've already occurred.
  By Lemma 2, we know that an invocation of *expand* returns after a finite number of steps, and *pool.current* is updated to point to a subsequent block.
  If we are in the recursive call to *seal*, and the $idx \leq LASTELEMPOS$ condition is $false$, trivally, a consistency changing operation must have occurred, as, the only way for the condition to evaluate to *true* is through a consistency changing operation, in the case that a block has been created during an invocation to *append*, for example.
  Otherwise, if we are in the recursive call to *seal*, and the $idx \leq LASTELEMPOS$ condition evaluates to *true*, we enter Case 1, which we just showed will successfully complete in a finite number of steps.

Lemma 8. *After invoking* $doForAll(fun)$*, a consistency changing instruction will be completed after a finite number of steps.*

We omit the proof for $doForAll$ since it proceeds in the exactly the same way as does the proof for *seal* in Lemma 7.

Lemma 9. *After updating b.index using* $WRITE2$ *or* $WRITE3$*, b.index is guaranteed to be incremented after a finite number of STEPS.*

Proof. For some index, *idx*, both calls to $WRITE2$ and $WRITE3$ attempt to write $idx+1$ to *b.index*. In both cases, it's possible that another thread could complete either $WRITE2$ or $WRITE3$, once again writing *idx* to *b.index* after the current thread has completed, in effect overwriting the current thread's write with $idx + 1$. By inspection of the pseudocode, both $WRITE2$ and $WRITE3$ will be repeated if *b.index* has not been incremented. However, since the number of threads operating on the FlowPool is finite, p, we are guaranteed that in the worst case, this scenario can repeat at most p times, before a write correctly updates *b.index* with $idx + 1$. □

Lemma 10. *Assume some concurrent operation is started. If some thread completes consistency changing CAS instruction, then some concurrent operation is guaranteed to be completed.*

Proof.

By trival inspection of the pseudocode, if $CAS2$ successfully completes on some thread, then that thread is guaranteed to complete the corresponding invocation of *append* in a finite number of steps.

Likewise by trivial inspection, if $CAS5$ successfully completes on some thread, then by Lemma 4, $tryWriteSeal$ is guaranteed to complete in a finite number of steps, and therefore, that thread is guaranteed to complete the corresponding invocation of *seal* in a finite number of steps.

The case for $CAS6$ is omitted since it follows the same steps as for the case of $CAS5$

A.1 Scratchpad

Lock-freedom.

By Lemma 2, we know that when we call *expand*, it's guaranteed that there will be a finite number of steps executed followed by a state changing operation.

After invoking append, if check is false, a consistency changing CAS (CAS2, CAS5, or CAS6) is completed after a finite number of steps.

We have to show that we'll update *current* after a finite number of steps.

The *expand* operation contains two CAS instructions; $CAS3$ and $CAS4$.

Lemma 1. *In each operation there is a finite number of execution steps between consecutive CAS instructions..*

Proof. The *append* operation is restarted in three cases. Case 1: iff *check* returns $true\ \wedge\ CAS1$ fails. Case 2: . Case 3:

$expand$

$tryWriteSeal$

$asyncDoForAll$

Then, the operation corresponding to the consistency-changing CAS instruction is guaranteed to be eventually completed

If there's a CAS failing, then some other thread completes the CAS. Consistency change means progress.

OLD: *Proof.* Case 1: The failing $CAS4$ happens after a successful $CAS3$. From lemma 1, we know that $CAS3$, which is a check that precedes $CAS4$, is guaranteed to be successfully completed by some thread, so we focus on the implications of failure of $CAS4$. Case 2: $CAS5$ takes place if *nb* is $null$. Therefore, in both cases, $CAS4$ and $CAS5$ successfully complete.

Lemma 3. *consistency-changing CAS operations ... will successfully complete.*

Proof. Uses $\mathbb{A}$.

Lemma 4. *Assume that the FlowPool is consistent with some abstract state* $\mathbb{A}$. *If one of the operations advance or expand succeeds, the FlowPool will remain consistent with the abstract state* $\mathbb{A}$ *following the operation.*

Proof. The CAS operations, denoted $CAS3$, and $CAS4$ in the pseudo-code, within the *expand* operation neither affect *elems* nor *sealed*, thus by Definition 4, causes no consistency change. Likewise, the *advance* operation either calls *expand* once, or it invokes $CAS1$ it may update the index of the current block, neither of which cause a consistency change.

Lemma 5 (). *If a consistency changing CAS completes, then the operation is guaranteed to successfully complete.*

Lemma 6 (). append operation is lock-free.

Lemma 7 (). seal operation is lock-free.

Lemma 8 (). doForAll operation is lock-free.