FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction

Aleksandar Prokopec[1], Heather Miller[1], Tobias Schlatter[1], Philipp Haller[2], and Martin Odersky[1]

[1] EPFL, Switzerland
`firstname.lastname@epfl.ch`
[2] Typesafe, Inc.
`firstname.lastname@typesafe.com`

Abstract. Implementing correct and deterministic parallel programs is challenging. Even though concurrency constructs exist in popular programming languages to facilitate the task of deterministic parallel programming, they are often too low level, or do not compose well due to underlying blocking mechanisms. In this paper, we present the design and implementation of a fundamental data structure for composable deterministic parallel dataflow computation through the use of functional programming abstractions. Additionally, we provide a correctness proof, showing that the implementation is linearizable, lock-free, and deterministic. Finally, we show experimental results which compare our *FlowPool* against corresponding operations on other concurrent data structures, and show that in addition to offering new capabilities, FlowPools reduce insertion time by $49 - 54\%$ on a 4-core i7 machine with respect to comparable concurrent queue data structures in the Java standard library.

Keywords: dataflow, concurrent data-structure, deterministic parallelism.

1 Introduction

Multicore architectures have become ubiquitous– even most mobile devices now ship with multiple core processors. Yet parallel programming has yet to enter the daily workflow of the mainstream developer. One significant obstacle is an undesirable choice programmers must often face when solving a problem that could greatly benefit from leveraging available parallelism. Either choose a non-deterministic, but performant, data structure or programming model, or sacrifice performance for the sake of clarity and correctness.

Programming models based on *dataflow* [1, 2] have the potential to simplify parallel programming, since the resulting programs are deterministic. Moreover, dataflow programs can be expressed more declaratively than programs based on mainstream concurrency constructs, such as shared-memory threads and locks, as programmers are only required to specify data and control dependencies. This allows one to reason sequentially about the intended behavior of their program, meanwhile enabling the underlying framework to effectively extract parallelism.

In this paper, we present the design and implementation of FlowPools, a fundamental dataflow collections abstraction which can be used as a building block for larger and more complex *deterministic* and parallel dataflow programs. Our FlowPool abstraction is backed by an efficient non-blocking data structure. As a result, our data structure benefits from the increased robustness provided by lock-freedom [12], since its operations are not blocked by delayed threads. We provide a lock-freedom proof, which guarantees progress regardless of the behavior, including the failure, of concurrent threads.

In combining lock-freedom with a functional interface, we go on to show that FlowPools are *composable.* That is, using prototypical higher-order functions such as `foreach` and `aggregate`, one can concisely form dataflow graphs, in which associated functions are executed asynchronously in a completely non-blocking way, as elements of FlowPools in the dataflow graph become available.

Finally, we show that FlowPools are able to overcome practical issues, such as out-of-memory errors, thus enabling programs based upon FlowPools to run indefinitely. By using a *builder* abstraction, instead of something like iterators or streams (which can lead to non-determinism) we are able to garbage collect parts of the data structure we no longer need, thus reducing memory consumption.

Our contributions are the following:

1. The design and Scala [19] implementation[1] of a parallel dataflow abstraction and underlying data structure that is deterministic, lock-free, & composable.
2. Proofs of lock-freedom, linearizability, and determinism.
3. Detailed benchmarks comparing the performance of our FlowPools against other popular concurrent data structures.

2 Model of Computation

FlowPools are similar to a typical collections abstraction. Operations invoked on a FlowPool are executed on its individual elements. However, FlowPools do not only act as a data container of elements. Unlike a typical collection, FlowPools also act as nodes and edges of a directed acyclic computation graph (DAG), in which the executed operations are registered with the FlowPool.

Nodes in this directed acyclic graph are data containers which are first class values. This makes it possible to use FlowPools as function arguments or to receive them as return values. Edges, on the other hand, can be thought of as combinators or higher-order functions whose user-defined functions are the previously-mentioned operations that are registered with the FlowPool. In addition to providing composability, this means that the DAG does not have to be specified at compile time, but can be generated dynamically at run time instead.

This structure allows for complete asynchrony, allowing the runtime to extract parallelism as a result. That is, elements can be asynchronously inserted, all registered operations can be asynchronously executed, and new operations can be asynchronously registered. Put another way, invoking several higher-order functions in succession on a given FlowPool does not add barriers between nodes

[1] See `http://www.assembla.com/code/scala-dataflow/git/nodes`

in the DAG, it only extends the DAG. This means that individual elements within a FlowPool can *flow* through different edges of the DAG independently.

Properties of FlowPools. In our model, FlowPools have certain properties which ensure that resulting programs are deterministic.

1. Single-assignment - an element added to the FlowPool cannot be removed.
2. No order - data elements in FlowPools are unordered.
3. Purity - traversals are side-effect free (pure), except when invoking FlowPool operations.
4. Liveness - callbacks are eventually asynchronously executed on all elements.

We claim that FlowPools are deterministic in the sense that all execution schedules either lead to some form of non-termination (*e.g.*, some exception), or the program terminates and no difference can be observed in the final state of the resulting data structures. This definition is practically useful, because in the case of non-termination it is guaranteed that on some thread an exception is thrown which aids debugging, *e.g.*, by including a stack trace. For a more formal definition and proof of determinism, see section 5.

3 Programming Interface

A FlowPool can be thought of as a concurrent pool data structure, *i.e.*, it can be used similarly to a collections abstraction, complete with higher-order functions, or combinators, for composing computations on FlowPools. In this section, we describe the semantics of several of those functional combinators and other basic operations defined on FlowPools.

Append (<<). The most fundamental of all operations on FlowPools is the concurrent thread-safe append operation. As its name suggests, it simply takes an argument of type `Elem` and appends it to a given FlowPool.

Foreach and Aggregate. A pool containing a set of elements is of little use if its elements cannot be manipulated in some manner. One of the most basic data structure operations is element traversal, often provided by iterators or streams– stateful objects which store the current position in the data structure. However, since their state can be manipulated by several threads at once, using streams or iterators can result in nondeterministic executions.

Another way to traverse the elements is to provide a higher-order `foreach` operator which takes a user-specified function as an argument and applies it to every element. For it to be deterministic, it must be called for every element that is eventually inserted into the FlowPool, rather than only on those present when `foreach` is called. Furthermore, determinism still holds even if the user-specified function contains side-effecting FlowPool operations such as `<<`. For `foreach` to be non-blocking, it cannot wait until additional elements are added to the FlowPool. Thus, the `foreach` operation must execute asynchronously, and be eventually applied to every element. Its signature is `def foreach[U](f: T => U): Future[Int]`, and its return type `Future[Int]` is an integer value which becomes available once `foreach` traverses all the elements added to the pool. This integer denotes the number of times the `foreach` has been called.

The `aggregate` operation aggregates the elements of the pool and has the following signature: `def aggregate[S](zero: =>S) (cb: (S, S) => S) (op: (S, T) => S): Future[S]`, where `zero` is the initial aggregation, `cb` is an associative operator which combines several aggregations, `op` is an operator that adds an element to the aggregation, and `Future[S]` is the final aggregation of all the elements which becomes available once all the elements have been added. The `aggregate` operator divides elements into subsets and applies the aggregation operator `op` to aggregate elements in each subset starting from the `zero` aggregation, and then combines different subset aggregations with the `cb` operator. In essence, the first part of `aggregate` defines the commutative monoid and the functions involved must be non-side-effecting. In contrast, the operator `op` is guaranteed to be called only once per element and it can have side-effects.

While in an imperative programming model, `foreach` and `aggregate` are equivalent in the sense that one can be implemented in terms of the other, in a single-assignment programming model `aggregate` is more expressive. The `foreach` operation can be implemented using `aggregate`, but not vice versa.

Builders. The FlowPool described so far must maintain a reference to all the elements at all times to implement the `foreach` operation correctly. Since elements are never removed, the pool may grow indefinitely and run out of memory. However, it is important to note that appending new elements does not necessarily require a reference to any of the existing elements. This observation allows us to move the `<<` operation out of the FlowPool and into a different abstraction called a `builder`. Thus, a typical application starts by registering all the `foreach` operations, and then it releases the references to FlowPools, leaving only references to builders. In a managed environment, the GC then can automatically discard the no longer needed objects.

Seal. After deciding that no more elements will be added, further appends can be disallowed by calling `seal`. This has the advantage of discarding the registered `foreach` operations. More importantly, the `aggregate` can complete its future– this is only possible once it is known there will be no more appends.

Simply preventing append calls after the point when `seal` is called, however, yields a nondeterministic programming model. Imagine a thread that attempts to seal the pool executing concurrently with a thread that appends an element. In one execution, the append can precede the seal, and in the other the append can follow the seal, causing an error. To avoid nondeterminism, there has to be an agreement on the current state of the pool. A convenient and sufficient way to make `seal` deterministic is to provide the expected pool size as an argument. The semantics of `seal` is such that it fails if the pool is already sealed with a different size or the number of elements is greater than the desired size. Note that we do not guarantee that the same exception always occurs on the same thread– rather, if *any* thread throws *some* exception in *some* execution schedule, then in *all* execution schedules *some* thread will throw *some* exception.

Higher-Order Operators. We now show how these basic abstractions can be used to build higher-order abstractions. To start, it is convenient to have generators that create certain pool types. In a dataflow graph, FlowPools created by generators can be thought of as source nodes. As an example, `tabulate` (below) creates a sequence of elements by applying a user-specified function `f` to natural numbers. One can imagine more complex generators, which add elements from a network socket or a file, for example.

```
def tabulate[T]
  (n: Int, f: Int => T)
  val p = new FlowPool[T]
  val b = p.builder
  def recurse(i: Int) {
    b << f(i)
    if i < n recurse(i + 1)
  }
  future { recurse(0) }
  p
```

```
def map[S](f: T => S)
  val p = new FlowPool[S]
  val b = p.builder
  for (x <- this) {
    b << f(x)
  } map {
    sz => b.seal(sz)
  }
  p
```

```
def foreach[U](f: T => U)
  aggregate(0)(_ + _) {
    (acc, x) =>
    f(x)
    acc + 1
  }
```

The `tabulate` generator starts by creating a FlowPool of an arbitrary type `T` and creating its builder instance. It then starts an asynchronous computation using the `future` construct (see the companion technical report [20] for explanation and examples), which recursively applies `f` to each number and adds it to the builder. The reference to the pool `p` is returned *immediately*, before the asynchronous computation completes.

A typical higher-order collection operator `map` is used to map each element of a dataset to produce a new dataset. This corresponds to chaining or pipelining the dataflow graph nodes. Operator `map` traverses the elements of `this` FlowPool and appends each mapped element to the builder. The `for` loop is syntactic sugar for calling the `foreach` method on `this`. We assume that the `foreach` return type `Future[Int]` has `map` and `flatMap` operations, executed once the future value becomes available. The `Future.map` above ensures that once the current pool (`this`) is sealed, the mapped pool is sealed to the appropriate size.

As argued before, `foreach` can be expressed in terms of `aggregate` by accumulating the number of elements and invoking the callback `f` each time. However, some patterns cannot be expressed in terms of `foreach`. The `filter` combinator filters out the elements for which a specified predicate does not hold. Appending the elements to a new pool can proceed as before, but the seal needs to know the exact number of elements added– thus, the `aggregate` accumulator is used to track the number of added elements.

```
def filter
  (pred: T => Boolean)
  val p = new FlowPool[T]
  val b = p.builder
  aggregate(0)(_ + _) {
    (acc, x) => if pred(x) {
      b << x
      1
    } else 0
  } map { sz => b.seal(sz) }
  p
```

```
def flatMap[S]
  (f: T => FlowPool[S])
  val p = new FlowPool[S]
  val b = p.builder
  aggregate(future(0))(add) {
    (af, x) =>
    val sf = for (y <- f(x))
      b << y
    add(af, sf)
  } map { sz => b.seal(sz) }
  p

def add(f: Future[Int], g: Future[Int]) =
  for (a <- f; b <- g) yield a + b
```

```
def union[T]
  (that: FlowPool[T])
  val p = new FlowPool[T]
  val b = p.builder
  val f = for (x <- this) b << x
  val g = for (y <- that) b << y
  for (s1 <- f; s2 <- g)
    b.seal(s1 + s2)
  p
```

```
type Terminal {                    type Block {              type FlowPool {
  sealed: Int                        array: Array[Elem]        start: Block
  callbacks: List[Elem => Unit]      next: Block               current: Block
}                                    index: Int              }
                                     blockindex: Int         LASTELEMPOS = BLOCKSIZE - 2
type Elem                          }                         NOSEAL = -1
```

Fig. 1. FlowPool data-types

The `flatMap` operation retrieves a pool for each element of `this` pool and adds its elements to the resulting pool. Given two FlowPools, it can be used to generate the Cartesian product of their elements. The implementation is similar to that of `filter`, but we reduce the size on the future values of the sizes– each intermediate pool may not yet be sealed. The operation `q union r`, as one might expect, produces a new pool which has elements of both pool `q` and pool `r`.

The last two operations correspond to joining nodes in the dataflow graph. Note that if we could somehow merge the two different `foreach` loops to implement the third join type `zip`, `zip` would be nondeterministic. The programming model does not allow us to do this, however. The `zip` function is better suited for data structures with deterministic ordering, such as Oz streams, which would in turn have a nondeterministic `union`.

4 Implementation

We now describe the FlowPool and its basic operations. In doing so, we omit the details not relevant to the algorithm[2] and focus on a high-level description of a non-blocking data structure. One straightforward way to implement a growing pool is to use a linked list of nodes that wrap elements. Since we are concerned about the memory footprint and cache-locality, we store the elements into arrays instead, which we call blocks. Whenever a block becomes full, a new block is allocated and the previous block is made to point to the `next` block. This way, most writes amount to a simple array-write, while allocation occurs only occasionally. Each block contains a hint `index` to the first free entry in the array, i.e. one that does not contain an element. An `index` is a hint, since it may actually reference an entry that comes earlier than the first free entry. Additionally, a FlowPool also maintains a reference to the first block called `start`. It also maintains a hint to the last block in the chain of blocks, called `current`. This reference may not always be up-to-date, but it always points to some block in the chain.

Each FlowPool is associated with a list of callbacks which have to be called in the future as new elements are added. Each FlowPool can also be in a sealed state, meaning there is a bound on the number of elements it can have. This information is stored as a `Terminal` value in the first free array entry. At all times, we maintain the invariant that the array in each block starts with a sequence of elements, followed by a `Terminal` delimiter. From a higher-level perspective, appending an element starts by copying the `Terminal` value to the next entry and then overwriting the current entry with the element being appended.

[2] Specifically the builder abstraction and the `aggregate` operation. The `aggregate` can be implemented using `foreach` with a side-effecting accumulator.

```
def create()
  new FlowPool {
    start = createBlock(0)
    current = start
  }

def createBlock(bidx: Int)
  new Block {
    array = new Array(BLOCKSIZE)
    index = 0
    blockindex = bidx
    next = null
  }

def append(elem: Elem)
  b = READ(current)
  idx = READ(b.index)
  nexto = READ(b.array(idx + 1))
  curo = READ(b.array(idx))
  if check(b, idx, curo) {
    if CAS(b.array(idx+1), nexto, curo) {
      if CAS(b.array(idx), curo, elem) {
        WRITE(b.index, idx + 1)
        invokeCallbacks(elem, curo)
      } else append(elem)
    } else append(elem)
  } else {
    advance()
    append(elem)
  }

def check(b: Block, idx:Int, curo:Object)
  if idx > LASTELEMPOS return false
  else curo match {
    elem: Elem =>
      return false
    term: Terminal =>
      if term.sealed = NOSEAL 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, b.array(idx))
  else {
    obj = READ(b.array(idx))
    if obj is Elem WRITE(b.index, idx + 1)
  }

def expand(b: Block, t: Terminal)
  nb = READ(b.next)
  if nb is null {
    nb = createBlock(b.blockindex + 1)
    nb.array(0) = t
    if CAS(b.next, null, nb)
      expand(b, t)
  } else {
    CAS(current, b, nb)
  }

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

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

def seal(size: Int)
  b = READ(current)
  idx = READ(b.index)
  if idx <= LASTELEMPOS {
    curo = READ(b.array(idx))
    curo match {
      term: Terminal =>
        if ¬tryWriteSeal(term, b, idx, size)
          seal(size)
      elem: Elem =>
        WRITE(b.index, idx + 1)
        seal(size)
      null =>
        error("unreachable")
    }
  } else {
    expand(b, b.array(idx))
    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 = NOSEAL {
    nterm = new Terminal {
      sealed = size
      callbacks = term.callbacks
    }
    return CAS(b.array(idx), term, nterm)
  } else if term.sealed ≠ size {
    error("already sealed with different size")
  } else return true

def foreach(f: Elem => Unit)
  future {
    asyncFor(f, start, 0)
  }

def asyncFor(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)
          asyncFor(f, b, idx)
      elem: Elem =>
        f(elem)
        asyncFor(f, b, idx + 1)
      null =>
        error("unreachable")
    }
  } else {
    expand(b, b.array(idx))
    asyncFor(f, b.next, 0)
  }
```

Fig. 2. FlowPool operations pseudocode

The `append` operation starts by reading the `current` block and the `index` of the free position. It then reads `nexto` after the first free entry, followed by a read of the `curo` at the free entry. The `check` procedure checks the conditions of the bounds, whether the FlowPool was already sealed or if the current array entry contains an element. In either of these events, the `current` and `index` values need to be set– this is done in the `advance` procedure. We call this the **slow path** of the `append` method. Notice that there are several situations which trigger the slow path. For example, if some other thread completes the `append` method but is preempted before updating the value of the hint `index`, then the `curo` will have the type `Elem`. The same happens if a preempted thread updates the value of the hint `index` after additional elements have been added, via unconditional write in line 23. Finally, reaching an end of block triggers the slow path.

Otherwise, the operation executes the **fast path** and appends an element. It first copies the `Terminal` value to the next entry with a CAS instruction in line 21, with `nexto` being the expected value. If it fails (e.g. due to a concurrent CAS), the append operation is restarted. Otherwise, it proceeds by writing the element to the current entry with a CAS in line 22, the expected value being `curo`. On success, it updates the `b.index` value and invokes all the callbacks (present when the element was added) with the `future` construct. In the implementation, we do not schedule an asynchronous computation for each element. Instead, the callback invocations are batched to avoid the scheduling overhead– the array is scanned for new elements until the first free entry is reached.

Interestingly, note that inverting the order of the reads in lines 18 and 19 would cause a race in which a thread could overwrite a `Terminal` value with some older `Terminal` value if some other thread appended an element in between.

The `seal` operation continuously increases the `index` in the block until it finds the first free entry. It then tries to replace the `Terminal` value there with a new `Terminal` value which has the seal size set. An error occurs if a different seal size is set already. The `foreach` operation works in a similar way, but is executed asynchronously. Unlike `seal`, it starts from the first element in the pool and calls the callback for each element until it finds the first free entry. It then replaces the `Terminal` value with a new `Terminal` value with the additional callback. From that point on the `append` method is responsible for scheduling that callback for subsequently added elements. Note that all three operations call `expand` to add an additional block once the current block is empty, to ensure lock-freedom.

Multi-lane FlowPools. Using a single block sequence (i.e. lane) to implement a FlowPool does not take full advantage of the lack of ordering guarantees and may cause slowdowns due to collisions when multiple concurrent writers are present. Multi-Lane FlowPools overcome this limitation by having a lane for each CPU, where each lane has the same implementation as the normal FlowPool.

This has several implications. First of all, CAS failures during insertion are avoided to a high extent and memory contention is decreased due to writes occurring in different cache-lines. Second, `aggregate` callbacks are added to each lane individually and aggregated once all of them have completed. Finally, `seal` needs to be globally synchronized in a non-blocking fashion.

Once `seal` is called, the remaining free slots are split amongst the lanes equally. If a writer finds that its lane is full, it writes to some other lane instead. This raises the frequency of CAS failures, but in most cases happens only when the FlowPool is almost full, thus ensuring that the `append` operation scales.

5 Correctness

We give an outline of the correctness proof here. More formal definitions, and a complete set of lemmas and proofs can be found in the tech report [20].

We define the notion of an abstract pool $\mathbb{A} = (elems, callbacks, seal)$ of elements in the pool, callbacks and the seal size. Given an abstract pool, abstract pool operations produce a new abstract pool. The key to showing correctness is to show that an abstract pool operation corresponds to a FlowPool operation– that is, it produces a new abstract pool corresponding to the state of the FlowPool after the FlowPool operation has been completed.

Lemma 5.1. Given a FlowPool consistent with some abstract pool, CAS instructions in lines 21, 63 and 66 do not change the corresponding abstract pool.

Lemma 5.2. Given a FlowPool consistent with an abstract pool $(elems, cbs, seal)$, a successful CAS in line 22 changes it to the state consistent with an abstract pool $(\{elem\} \cup elems, cbs, seal)$. There exists a time $t_1 \geq t_0$ at which every callback $f \in cbs$ has been called on $elem$.

Lemma 5.3. Given a FlowPool consistent with an abstract pool $(elems, cbs, seal)$, a successful CAS in line 124 changes it to the state consistent with an abstract pool $(elems, (f, \emptyset) \cup cbs, seal)$ There exists a time $t_1 \geq t_0$ at which f has been called for every element in $elems$.

Lemma 5.4. Given a FlowPool consistent with an abstract pool $(elems, cbs, seal)$, a successful CAS in line 105 changes it to the state consistent with an abstract pool $(elems, cbs, s)$, where either $seal = -1 \wedge s \in \mathbb{N}_0$ or $seal \in \mathbb{N}_0 \wedge s = seal$.

Theorem 5.5. [Safety] Operations `append`, `foreach` and `seal` are consistent with the abstract pool semantics.

Theorem 5.6. [Linearizability] Operations `append` and `seal` are linearizable.

Lemma 5.7. After invoking a FlowPool operation `append`, `seal` or `foreach`, if a non-consistency changing CAS in lines 21, 63, or 66 fails, they must have already been completed by another thread since the FlowPool operation began.

Lemma 5.8. After invoking a FlowPool operation `append`, `seal` or `foreach`, if a consistency changing CAS in lines 22, 105, or 124 fails, then some thread has successfully completed a consistency changing CAS in a finite number of steps.

Lemma 5.9. After invoking a FlowPool operation `append`, `seal` or `foreach`, a consistency changing instruction will be completed after a finite number of steps.

$t ::=$	terms	$p \in \{(vs, \sigma, cbs) \mid vs \subseteq Elem, \sigma \in \{-1\} \cup \mathbb{N},$ $cbs \subset Elem \Rightarrow Unit\}$
`create` p	pool creation	
$p << v$	append	$v \in Elem$
p `foreach` f	foreach	$f \in Elem \Rightarrow Unit$
p `seal` n	seal	$n \in \mathbb{N}$
t_1 ; t_2	sequence	

Fig. 3. Syntax

Theorem 5.10. [Lock-freedom] FlowPool operations `append`, `foreach` and `seal` are lock-free.

Determinism. We claim that the FlowPool abstraction is *deterministic* in the sense that a program computes the same result (possibly an error) regardless of the interleaving of execution steps. Here we give an outline of the determinism proof. A complete formal proof can be found in the technical report [20].

The following definitions and the determinism theorem are based on the language shown in Figure 3. The semantics of our core language is defined using reduction rules which define transitions between *execution states*. An execution state is a pair $T \mid P$ where T is a set of concurrent threads and P is a set of FlowPools. Each thread executes a *term* of the core language (typically a sequence of terms). State of a thread is represented as the (rest of) the term that it still has to execute; this means there is a one-to-one mapping between threads and terms. For example, the semantics of `append` is defined by the following reduction rule (a complete summary of all the rules can be found in the appendix):

$$\frac{t = p << v \ ;\ t' \quad p = (vs, cbs, -1) \quad p' = (\{v\} \cup vs, cbs, -1)}{t, T \mid p, P \ \longrightarrow\ t', T \mid p', P} \ (\textsc{Append1})$$

Append simply adds the value v to the pool p, yielding a modified pool p'. Note that this rule can only be applied if the pool p is not sealed (the seal size is -1). The rule for *foreach* modifies the set of callback functions in the pool:

$$\frac{\begin{array}{c} t = p \texttt{ foreach } f \ ;\ t' \quad p = (vs, cbs, n) \\ T' = \{g(v) \mid g \in \{f\} \cup cbs, v \in vs\} \quad p' = (vs, \{f\} \cup cbs, n) \end{array}}{t, T \mid p, P \ \longrightarrow\ t', T, T' \mid p', P} \ (\textsc{Foreach2})$$

This rule only applies if p is sealed at size n, meaning that no more elements will be appended later. Therefore, an invocation of the new callback f is scheduled for each element v in the pool. Each invocation creates a new thread in T'.

Programs are built by first creating one or more FlowPools using `create`. Concurrent threads can then be started by (a) appending an element to a FlowPool, (b) sealing the FlowPool and (c) registering callback functions (`foreach`).

Definition 5.11. [Termination] A term t terminates with result P if its reduction ends in execution state $\{t : t = \{\epsilon\}\} \mid P$.

Definition 5.12. [Interleaving] Consider the reduction of a term t: $T_1 \mid P_1 \longrightarrow T_2 \mid P_2 \longrightarrow \ldots \longrightarrow \{t : t = \{\epsilon\}\} \mid P_n$. An *interleaving* is a reduction of t starting in $T_1 \mid P_1$ in which reduction rules are applied in a different order.

Definition 5.13. [Determinism] The reduction of a term t is *deterministic iff* either (a) t does not terminate for any interleaving, or (b) t always terminates with the same result for all interleavings.

Theorem 5.14. [FlowPool Determinism] Reduction of terms t is deterministic.

6 Evaluation

We evaluate our implementation (single-lane and multi-lane FlowPools) against the LinkedTransferQueue [14] for all benchmarks and the ConcurrentLinkedQueue [17] for the insert benchmark, both found in JDK 1.7, on three different architectures; a quad-core 3.4 GHz i7-2600, 4x octa-core 2.27 GHz Intel Xeon x7560 (both with hyperthreading) and an octa-core 1.2GHz UltraSPARC T2 with 64 hardware threads. In this section, we focus on the scaling properties of the above-mentioned data structures, Figures 4 & 5.

In the *Insert* benchmark, Figure 4, we evaluate concurrent insert operations, by distributing the work of inserting N elements into the data structure concurrently across P threads. In Figure 4, it's evident that both single-lane FlowPools and concurrent queues do not scale well with the number of concurrent threads, particularly on the i7 architecture. They quickly slow down, likely due to cache line collisions and CAS failures. On the other hand, multi-lane FlowPools scale well, as threads write to different lanes, and hence different cache lines, meanwhile also avoiding CAS failures. This appears to reduce execution time for insertions up to 54% on the i7, 63% on the Xeon and 92% on the UltraSPARC.

The performance of higher-order functions is evaluated in the *Reduce*, *Map* (both in Figure 4) and *Histogram* benchmarks (Figure 5). It's important to note that the *Histogram* benchmark serves as a "real life" example, which uses both the `map` and `reduce` operations that are benchmarked in Figure 4. Also note that in all of these benchmarks, the time it takes to insert elements into the FlowPool is also measured, since the FlowPool programming model allows one to insert elements concurrently with the execution of higher-order functions.

In the *Histogram* benchmark, Figure 5, P threads produce a total of N elements, adding them to the FlowPool. The `aggregate` operation is then used to produce 10 different histograms concurrently with a different number of bins. Each separate histogram is constructed by its own thread (or up to P, for multi-lane FlowPools). A crucial difference between queues and FlowPools here, is that with FlowPools, multiple histograms are produced by invoking several `aggregate` operations, while queues require writing each element to several queues– one for each histogram. Without additional synchronization, reading a single queue is not an option, since elements have to be removed from the queue eventually, and it is not clear to each reader when to do this. With FlowPools, elements are automatically garbage collected when no longer needed.

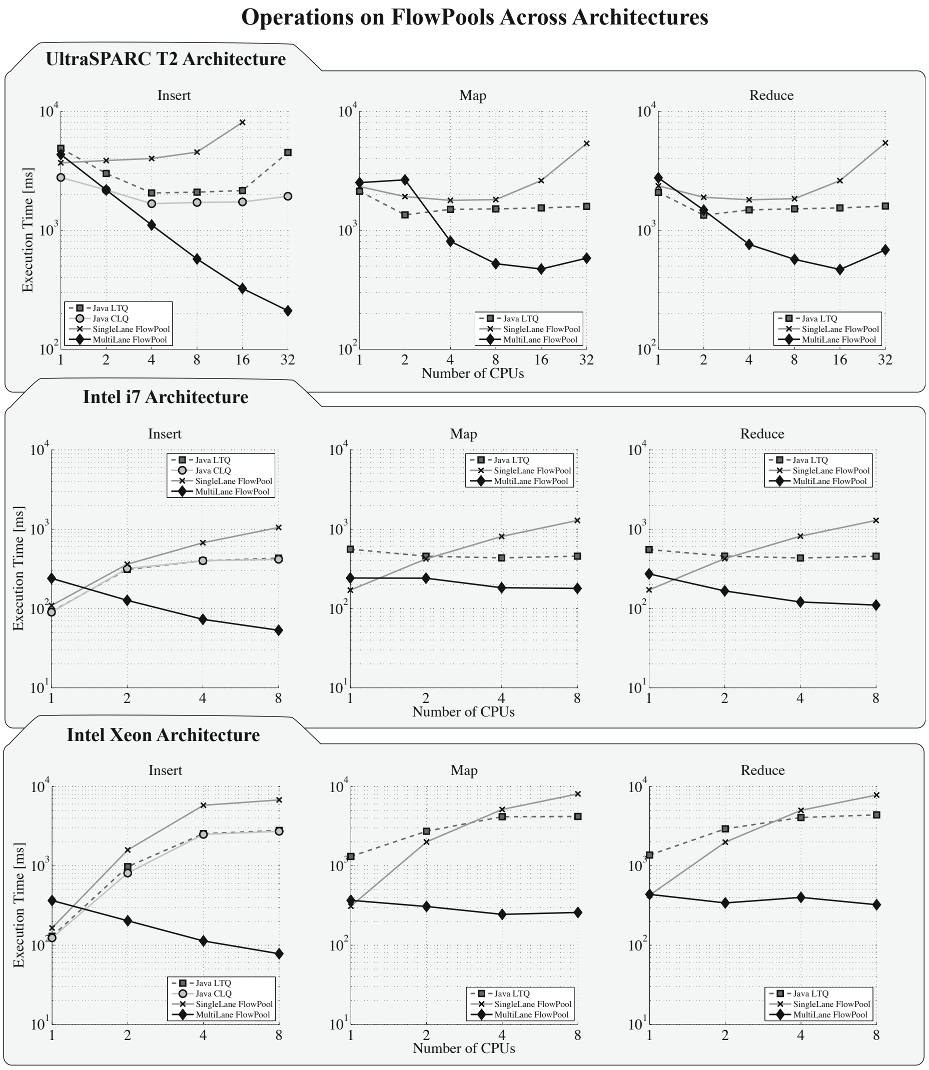

Fig. 4. Execution time vs parallelization across three different architectures on three important FlowPool operations; insert, map, reduce

Finally, to validate the last claim of garbage being automatically collected, in the *Communication/Garbage Collection* benchmark, Figure 5, we create a pool in which a large number of elements N are added concurrently by P threads. Each element is then processed by one of P threads through the use of the `aggregate` operation. We benchmark against linked transfer queues, where P threads concurrently remove elements from the queue and process it. For each run, we vary the size of the N and examine its impact on the execution time. Especially in the cases of the Intel architectures, the multi-lane FlowPools perform considerably better than the linked transfer queues. As a matter of fact, the

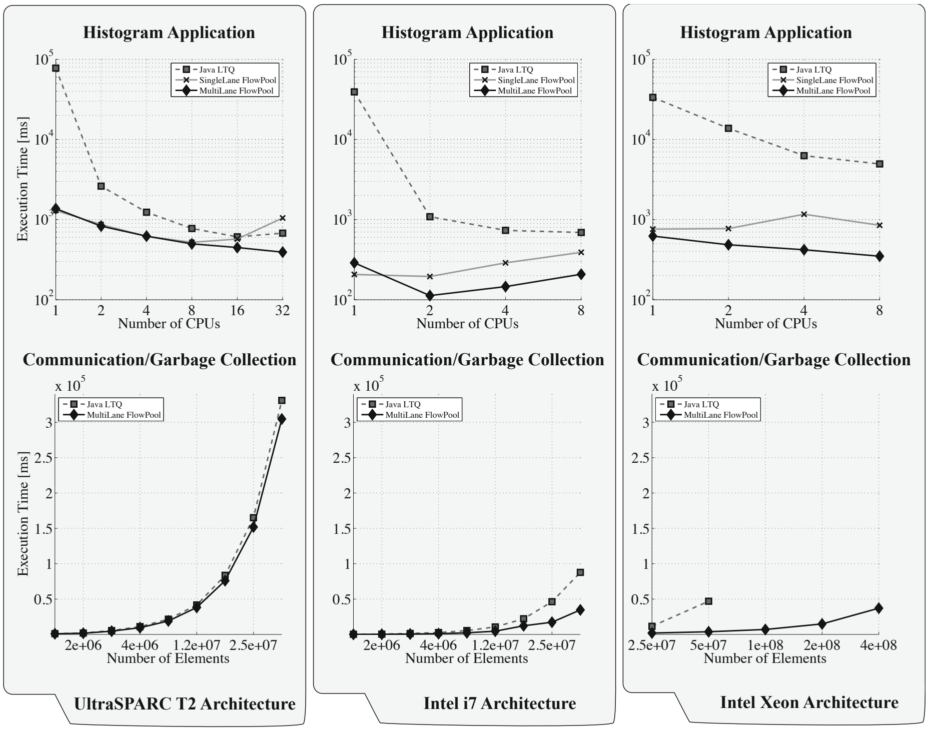

Fig. 5. Execution time vs parallelization on a real histogram application (top), & communication benchmark (bottom) showing memory efficiency, across all architectures.

linked transfer queue on the Xeon benchmark ran out of memory, and was unable to complete, while the multi-lane FlowPool scaled effortlessly to 400 million elements, indicating that unneeded elements are properly garbage collected.

7 Related Work

An introduction to linearizability and lock-freedom is given by Herlihy and Shavit [13]. A detailed overview of concurrent data structures is given by Moir and Shavit [18]. To date, concurrent data structures remain an active area of research– we restrict this summary to those relevant to this work.

Concurrently accessible queues have been present for a while, an implementation is described by [16]. Non-blocking concurrent linked queues are described by Michael and Scott [17]. This CAS-based queue implementation is cited and used widely today, a variant of which is present in the Java standard library. More recently, Scherer, Lea and Scott [14] describe synchronous queues which internally hold both data and requests. Both approaches above entail blocking (or spinning) at least on the consumer's part when the queue is empty.

While the abstractions above fit well in the concurrent imperative model, they have the disadvantage that the programs written using them are inherently nondeterministic. Roy and Haridi [21] describe the Oz programming language, a subset of which yields programs deterministic by construction. Oz dataflow streams are built on top of single-assignment variables and are deterministically ordered. They allow multiple consumers, but only one producer at a time. Oz has its own runtime which implements blocking using continuations.

The concept of single-assignment variables is used to provide logical variables in concurrent logic programming languages [23]. It is also embodied in futures proposed by Baker and Hewitt [11], and promises first mentioned by Friedman and Wise [7]. Futures were first implemented in MultiLISP [10], and have been employed in many languages and frameworks since. Scala 2.10 futures [9] and Twitter futures [6] are of interest, because they define monadic operators and a number of high-level combinators that create new futures. These APIs avoid blocking. Futures have been generalized to data-driven futures, which provide additional information to the scheduler [24]. Many frameworks have constructs that start an asynchronous computation and yield a future holding its result, for example, Habanero Java [3] (`async`) and Scala [19] (`future`).

A number of other models and frameworks recognized the need to embed the concept of futures into other data-structures. Single-assignment variables have been generalized to I-Structures [1] which are essentially single-assignment arrays. CnC [4, 2] is a parallel programming model influenced by dynamic dataflow, stream-processing and tuple spaces [8]. In CnC the user provides high-level operations along with the ordering constraints that form a computation dependency graph. FlumeJava [5] is a distributed programming model which relies heavily on the concept of collections containing futures. An issue that often arises with dataflow programming models are unbalanced loads. This is often solved using bounded buffers which prevent the producer from overflowing the consumer.

Opposed to the correct-by-construction determinism described thus far, a type-systematic approach can also ensure that concurrent executions have deterministic results. Recently, work on Deterministic Parallel Java showed that a region-based type system can ensure determinism [15]. X10's constrained-based dependent types can similarly ensure determinism and deadlock-freedom [22].

8 Conclusion

The abstraction for concurrent dataflow programming we presented provides a composable deterministic programming model. It can be implemented in a provably non-blocking manner and is efficient as well, as shown in experiments.

As future work, we plan developing other concurrent collection types with deterministic semantics, which enrich the correct-by-construction single-assignment model, such as bounded buffers, streams and maps. On the implementation level, we anticipate the need of embedding the callbacks within the data-structure itself, as is the case with callback-based futures and FlowPools – this has a particular benefit on platforms which do not support efficient continuations.

References

1. Arvind, Nikhil, R.S., Pingali, K.K.: I-structures: Data structures for parallel computing. ACM Trans. Prog. Lang. and Sys. 11(4), 598–632 (1989)
2. Budimlic, Z., Burke, M.G., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D.M., Sarkar, V., Schlimbach, F., Tasirlar, S.: Concurrent collections. Scientific Programming 18(3-4), 203–217 (2010)
3. Budimlic, Z., Cavé, V., Raman, R., Shirako, J., Tasirlar, S., Zhao, J., Sarkar, V.: The design and implementation of the Habanero-Java parallel programming language. In: OOPSLA Companion, pp. 185–186 (2011)
4. Burke, M.G., Knobe, K., Newton, R., Sarkar, V.: Concurrent collections programming model. In: Encyclopedia of Parallel Computing, pp. 364–371 (2011)
5. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: FlumeJava: easy, efficient data-parallel pipelines. ACM SIGPLAN Notices 45(6), 363–375 (2010)
6. Eriksen, M., Kallen, N.: Twitter Finagle: Futures, `http://twitter.github.com/finagle/`
7. Friedman, D., Wise, D.: The impact of applicative programming on multiprocessing. In: International Conference on Parallel Processing (1976)
8. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems 7(1), 80–112 (1985)
9. Haller, P., Prokopec, A., Miller, H., Klang, V., Kuhn, R., Jovanovic, V.: Scala improvement proposal: Futures and promises, SIP-14 (2012), `http://docs.scala-lang.org/sips/pending/futures-promises.html`
10. Halstead, J.R.H.: MultiLISP: A language for concurrent symbolic computation. ACM Trans. Prog. Lang. and Sys. 7(4), 501–538 (1985)
11. Henry, J., Baker, C., Hewitt, C.: The incremental garbage collection of processes. In: Proc. Symp. on Art. Int. and Prog. Lang. (1977)
12. Herlihy, M.: A methodology for implementing highly concurrent data structures. In: PPoPP, pp. 197–206 (1990)
13. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming (April 2008)
14. Scherer III, W.N., Lea, D., Scott, M.L.: Scalable synchronous queues. Commun. ACM 52(5), 100–111 (2009)
15. Bocchino Jr., R.L., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A type and effect system for deterministic parallel Java. In: OOPSLA, pp. 97–116 (2009)
16. Mellor-Crummey, J.M.: Concurrent queues: Practical fetch-and-Φ algorithms (1987)
17. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC, pp. 267–275 (1996)
18. Moir, Shavit: Concurrent data structures. In: Mehta, Sahni (eds.) Handbook of Data Structures and Applications, Chapman & Hall/CRC (2005)
19. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Press, Mountain View (2010)
20. Prokopec, A., Miller, H., Schlatter, T., Haller, P., Odersky, M.: Flowpools: A lock-free deterministic concurrent dataflow abstraction– proofs. Technical Report EPFL-REPORT-181098, EPFL, Lausanne (June 2012)

21. Roy, P.V., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)
22. Saraswat, V.A., Sarkar, V., von Praun, C.: X10: concurrent programming for modern architectures. In: PPOPP, p. 271 (2007)
23. Shapiro, E.: The family of concurrent logic programming languages. ACM Computing Surveys 21(3), 412 (1989)
24. Tasirlar, S., Sarkar, V.: Data-driven tasks and their implementation. In: ICPP, pp. 652–661 (2011)