Achieving Efficient Work-Stealing for Data-Parallel Collections

Aleksandar Prokopec
École Polytechnique Fédérale de Lausanne
EPFL IC IFF LAMP, Station 14
1015 Lausanne, Switzerland
aleksandar.prokopec@epfl.ch

Martin Odersky
École Polytechnique Fédérale de Lausanne
EPFL IC IFF LAMP, Station 14
1015 Lausanne, Switzerland
martin.odersky@epfl.ch

ABSTRACT
In modern programming high-level data-structures are an important foundation for most applications. With the rise of the multicore era, there is a growing trend of supporting data-parallel collection operations in general purpose programming languages and platforms. To facilitate object-oriented reuse these operations are highly parametric, incurring abstraction performance penalties. Furthermore, data-parallel operations must scale when used in problems with irregular workloads. Work-stealing is a proven load-balancing technique when it comes to irregular workloads, but general purpose work-stealing also suffers from abstraction penalties.

In this paper we present a generic design of a data-parallel collections framework based on work-stealing for shared-memory architectures. We show how abstraction penalties can be overcome through callsite specialization of data-parallel operations instances. Moreover, we show how to make work-stealing fine-grained and efficient when specialized for particular data-structures. We experimentally validate the performance of different data-structures and data-parallel operations, achieving up to 60× better performance with abstraction penalties eliminated and 3× higher speedups by specializing work-stealing compared to existing approaches.

Categories and Subject Descriptors
D.1.3 [Concurrent Programming]: Parallel programming; D.3.3 [Language constructs and features]: Concurrent programming structures; E.1 [Data Structures]: Trees, Arrays

General Terms
Algorithms

Keywords
data parallelism, conc-lists, work-stealing collections, callsite specialization, parallel hash-tables, parallel arrays, abstraction penalty, workload-driven, load balancing, domain-specific work-stealing

1. INTRODUCTION
While the declarative nature of data-parallel programming makes programs easier to understand and maintain, as well as to apply to a plethora of different problems [30], implementing an efficient data-parallel framework remains a challenging task. This task is only made harder by the fact that data-parallel frameworks offer generality on several levels. First, parallel operations are generic both in the type of the data records and the way these records are processed. Orthogonally, records are organized into data sets in different ways depending on how they are accessed – as arrays, hash-tables, trees or heaps. Let us consider the example of a subroutine that computes the mean of a set of measurements to illustrate these concepts. We show both its imperative and data-parallel variant.

def mean(x: Array[Int]) = {
  var sum = 0
  while (i < x.length) {
    sum += x(i); i += 1
  }
  sum.toDouble / x.length }

The data-parallel operation that the declarative-style mean subroutine relies on is fold, which aggregates multiple values into a single value. This operation is generic in the user-specified aggregation operator. The data set is represented with an array and the data records are the elements of the array; in this case integers. We show a simplified implementation of the parallel fold method [24] [41].

val subsets = x.iterator.split
val results = subsets.inParallel { subset =>
  var sum = z
  while (subset.hasNext) sum = op(sum, subset.next())
  results.foldLeft(z)(op) }

We assume collections have a method that returns an iterator and that this iterator can be efficiently split into subsets [9] [24] [41] [44]. These subsets are processed in parallel by different workers from a high-level perspective, this is done by the inParallel call. Once all the workers complete, their results can be aggregated sequentially. We focus on the work done by separate workers, namely, lines 9 through 11. Note that the while loop in those lines closely resembles the imperative variant of the method mean, with several differences. The neutral element of the aggregation z is generic and specified through an argument. Then, instead of comparing a local variable i against the array length, method hasNext is being called, which translates to a dynamic dispatch. The second dynamic dispatch updates the state of the iterator and returns the next element and another dynamic dispatch is required to apply the summation operator to the integer values.

These inefficiencies are referred to as the abstraction penalties. We can identify several abstraction penalties above. First of all, in typical object-oriented languages such as Java or C++ the dynamic dispatches amount to reading the address of the virtual method table and then the address of the appropriate method from that table. Second, and not immediately apparent, the itera-
2. RELATED WORK

Data-parallelism is a well-established concept in parallel programming languages dating back to APL [29], subsequently adopted by languages like NESL [5], High Performance Fortran [37] and ZPL [12]. With the emergence of commodity parallel hardware data parallelism is gaining more traction. Programming platforms like OpenCL and CUDA focusing mainly on GPUs are heavily oriented towards data parallelism. Chapel [11] is a parallel programming language supporting both task and data parallelism that improves the separation between data-structure implementation and algorithm description. The idea of specializing the data-parallel operation with the iterator instance itself comes from Chapel, where it was applied to efficiently traversing arrays in a platform-independent way [31]. X10 [14] is a parallel programming language with both JVM and C backends providing both task and data parallelism, and a variety of other modern concurrency constructs. Fortress [4] is another parallel programming language targetting the JVM with implicit parallelism and a highly declarative programming style. JVM-based languages like Java and Scala [38] provide data-parallel programming support as part of the standard library [24] [41]. Scala Parallel Collections support data-parallelism in a generic way for different collections and parallelize concurrent data-structures through the use of efficient lock-free snapshots [40] [42]. STAPL [9] and Intel TBB [44] are data-parallel libraries for C++ that rely on the template mechanism [48] and the STL architecture. In distributed computing-data parallel frameworks like MapReduce [18], FlumeJava [13] and Dryad [28] used for processing large data sets have been proposed in the recent years.

Most data-parallel languages rely on parallel loops, the scheduling of which bears a critical importance. The fixed-size chunking [32] technique was among the first techniques that allowed a more fine-grained load-balancing. It divides the loop into smallest possible subsets and the workers synchronize to obtain them from a central queue. A downside of this approach is that it fails to load balance the more irregular workloads well. Other variable size chunking approaches have been proposed, including guided self-scheduling [39], factoring [27] and trapezoidal self-scheduling [51], but static partitioning decisions of these techniques have proven detrimental. Work-stealing is a load balancing technique used in the Cilk programming language [6] [22] to support task parallelism. In work-stealing each worker maintains its own work queue and steals work from other workers when its own queue is empty. Work-stealing is particularly applicable to problems with irregular workloads [1] [3] [16] [19]. It has been traditionally used as the load balancing technique for task parallel programming [7] [34] [35] [49], but can be applied to data parallelism as well [43] [50].

Work-stealing tree scheduling [43] is a load balancing technique in which work is kept in a tree rather than a work queue. Each node in the tree contains a subset of the data-parallel loop and is owned by a single worker. A stealer notifies the owner of the desired leaf node that the node is invalidated and replaces it with two leaf nodes, dividing the remaining work. Due to a work-stealing mechanism specialized for data-parallel loops and its tendency to keep the worker in isolation as long as possible this technique can efficiently schedule highly irregular workloads that traditional approaches [27] [32] [39] [41] [51] cannot cope with. In the context of the JVM compilation techniques were proposed to eliminate boxing selectively, like the generic type specialization.

\[
\begin{align*}
13 & (\text{3 until } N) \ \text{filter} \ (i \Rightarrow) \\
14 & \quad (2 \text{ to } \sqrt{i}) \ \text{forall} \ (d \Rightarrow) \ i \ % \ d \ != \ 0 \\
\end{align*}
\]

For each of the numbers \(i\) between 3 and \(N\) the `filter` predicate checks if any number up to the square root of \(i\) divides \(i\). The amount of computation for each element depends on its value, making this data-parallel computation irregular. If the numbers are specified as part of the program input, then there is no way for static analysis to optimally partition the work at compile time. Similarly, not all workers might be available during execution.

While static partitioning should ideally be combined with runtime techniques [10] [47], this paper focuses on runtime workload-driven load balancing. So far, work-stealing has proven an efficient runtime load balancing technique for irregular problems [1] [3] [7] [16] [19] [22] [49], and the collections design we propose adopts work-stealing as well. It was shown that tailoring the work-stealing techniques to specific domains allows a more fine-grained work-stealing, thus better load balancing data-parallel computations [17] [43]. For this reason, our design integrates work-stealing with the shape of the data-structure, allowing the chunks that the elements are divided into to be as small as possible. As we will show, some existing approaches that ignore this potential gain in specializing work-stealing and rely only on general-purpose task work-stealing fail to parallelize irregular data-parallel workloads well [24] [41] – we will call such inefficiencies the scheduling penalty. The goal of this paper is to twofold. First, we show how the aforementioned abstraction penalties can be eliminated in a generic way for different data-structures and data-parallel operations, achieving optimal or near optimal performance. In doing so we rely on an abstraction called a kernel of a data-parallel operation, which is comprised of the specialized code for traversing and processing a chunk of data for a specific data-parallel operation instance. Second, we show how to minimize the scheduling penalties by employing fine-grained work-stealing for different data-structures in a generic, efficient and lock-free manner. We will introduce the concept of work-stealing iterators, which abstract over how work is divided into chunks and how it is stolen.

The rest of the paper is organized as follows. Section 2 presents the related work and more closely examines the work-stealing tree scheduling. Section 3 describes the work-stealing iterator and kernel abstractions in detail, as well as their implementations for different data-structures and data-parallel operations. In Section 4 we evaluate the performance of data-parallel collection operations on a range of microbenchmarks as well as on several larger benchmark applications. Finally, Section 5 concludes.
transformation used in Scala [21]. While generic type specialization can be used to eliminate boxing, it does not help in eliminating other abstraction penalties. For this reason we rely on the Scala macro system [8], but note that our technique can be applied to languages with a templating mechanism like C++ [48].

3. DESIGN AND IMPLEMENTATION

It is common that tasks recursively spawn subtasks in task parallel programming, potentially generating additional work to be stolen. This fact drives the design of many language runtimes based on work-stealing [61] [22] [34] [49] – only a single, usually oldest task is stolen at a time, the execution of which can hopefully create more subtasks. Conversely, in data parallel programming the parallelism units are not tasks but individual collection elements that do not generate more work. Thus, stealing must proceed in batches of elements to reduce the scheduling penalty.

The work-stealing tree scheduler [43] exploits this observation by dividing the remaining workload equally between the stealer and the victim when a steal occurs. In this approach each worker keeps the loop iteration index and atomically increments it to inform potential stealers of its progress. The iteration index is kept in the work-stealing node structure belonging to a specific processor $\pi$. Each work-stealing node traverses a specific subset of the parallel loop. This is shown in Figure 1 in the AVAILABLE state – the 0 and the $u$ denote the bounds of the parallel loop, and $x$ denotes the current value of the iteration index. A stealer $\psi$ invalidates this index to prevent the victim from further increments and, importantly, at the same time captures the information about its progress. This is shown in the STOLEN state in Figure 1. Subsequent updates to the iteration index are disallowed and the work-stealing node is expanded by creating two child nodes, each of which holds roughly half of the remaining elements of the original node.

This approach to scheduling data-parallel operations is particularly efficient in load-balancing irregular data-parallel operations, as well as uniform ones. Two different data-parallel workloads and the typical states of the work-stealing tree data-structure at the end of the data-parallel operation are shown in Figure 2 for illustration purposes. The uniform workload like the fold mentioned in the introduction yields a balanced work-stealing tree in which every worker processes roughly the same number of elements and works in isolation most of the time without communicating other workers. The irregular workload like the prime number computation mentioned earlier yields a fairly unbalanced work-stealing tree in which the worker 1 processes the smaller numbers much earlier than the worker 4 completes the computation on the bigger ones. Instead of remaining idle, worker 1 steals some of the expensive elements. In general, the unbalancing factor in a work-stealing subtree is proportional to the workload irregularity in the corresponding part of the parallel loop. Importantly, when the irregularity is high, there is enough work per each element to amortize the scheduling penalties of creating new work-stealing tree nodes. Conversely, when the irregularity is low the per element work may be low too, but there are less nodes being created. The scheduling is thus fully adaptive and occurs at runtime – we say that it is workload-driven.

We omit the details of how the scheduler uses the work-stealing tree, i.e. expands it or assigns workers to specific nodes – this was already discussed in detail in related work [43]. We instead focus on the code that the workers and stealers execute. The pseudocode we show closely resembles Scala, but relies on language features available in modern general-purpose programming languages.

Let's start by showing the pseudocode for a worker executing a parallel loop. We assume that the worker is assigned a chunk determined by the integers $\text{start} \geq 0$ and $\text{until} \geq \text{start}$. It also maintains a globally visible integer $\text{progress}$ which it updates atomically with a CAS. This value denotes the first loop element within $\langle \text{start}, \text{until} \rangle$ that the worker is not obliged to process.

```python
def work() {
    var loop = true
    var step = 0
    while (loop) {
        step = update(step)
        val p = READ(progress)
        if (p ≥ until || p < 0) loop = false else
            if (CAS(progress, p, min(until, p + step)))
                apply(p, min(until, p + step) )
    }
}
```

The algorithm uses a value $\text{step}$ to decide how many loop elements to try to commit to in each iteration. Updating $\text{step}$ in line 5 and its effect on scheduling has been studied elsewhere [27] [32] [39] [43] [51] and is outside of the scope of this work, but it suffices to say that this value has to be varied to amortize the scheduling costs and achieve the best speedup [43]. In each loop iteration the worker reads the value of $\text{progress}$ and tries to atomically increment it with a CAS. If it succeeds, it is committed to process all the elements smaller than the last value written to $\text{progress}$. It does so by calling $\text{apply}$ in line 9, which executes a user-specified operation on each element within the specified range. Section 3.2 shows how $\text{apply}$ corresponds to a specific operation instance.

The stealer invalidates the $\text{progress}$ by executing the following.

```python
def markStolen() {
    val p = READ(progress)
    if (p < until ∧ p ≥ 0)
        if (~CAS(progress, p, -p - 1)) markStolen()
}
```

Note that replacing the current value of $\text{progress}$ with a negative value allows decoding the previous state uniquely. Also, neither the worker nor any of the stealers write to $\text{progress}$ after it becomes negative. We do now show how the remaining work is split after $\text{markStolen}$ completes – at this point there is sufficient information to reach a consensus on that in a lock-free way. Note that while this kind of execution of arbitrary parallel loops is not itself lock-free because a specific worker commits to processing specific elements, the work-stealing process is, as stealers proceed without the help of the victim as long as there are elements left in $\text{progress}$.

3.1 Work-stealing iterators

The goal of this section is to augment the iterator abstraction [36] with the facilities that support work-stealing. The $\text{progress}$ value
Contract advance and stolen states, respectively. If an invocation returns a pair $(n_1, n_2)$ at time $t_0$ then the call to state returned $S$ at some time $t_1 < t_0$. **Traversal contract.** Define $X = x_1 x_2 \ldots x_m$ as the sequence of return values of next invocations at times $t'_1 < t'_2 < \ldots < t'_m$. If a call to state at $t > t'_m$ returns $C$ then $e(i) = X$. Otherwise, let an invocation of expanded on an iterator $i$ return $(i_1, i_2)$. Then $e(i) = X \cdot e(i_1) \cdot e(i_2)$, where $\cdot$ is concatenation. There exists a fixed $E$ such that $E = e(i)$ for all valid sequences of advance and next invocations.

While the last contract may seem complicated, it merely formalizes the notion that every iterator always traverses the same elements in the same order. We show several iterator implementations next.

**IndexIterator.** This is a simple iterator implementation following from refactorings in Figure 3. It is applicable to parallel ranges, arrays, vectors and data-structures where indexing is fast. We show a generic implementation in Figure 5. The CAS instructions are the linearization points for linearizable methods. Note that the IndexIterator contains a private nextProgress and nextUntil fields that the tail-recursive advance updates after a successful CAS in line 30. These fields are also used by next and hasNext in a non-atomic way. The contracts specify that those methods are only called by the owner in isolation, so there is no need to make the fields globally visible. This improves performance since next is used in generic operation implementations (see Section 3.2).

All method contracts are straightforward to verify and follow from the linearizability of CAS. For example, if state returns $S$ or $C$ at time $t_0$, then the progress was either negative or equal to until at $t_0$. All the writes to progress are CAS instructions that check that progress is neither negative nor equal to until. Therefore, progress has the same value $\forall t > t_0$ and state returns the same value $\forall t > t_0$.

**HashIterator.** Hash tables are an ubiquitous data structure in programming languages and in a variety of applications that rely

---

**Figure 3: The generalized loop scheduling algorithm**

```java
1  def work(it: StealIterator[T]) = {
2     var step = 0
3     while (it.state() == A) {
4         step = update(step)
5         val chunk = it.advance(step)
6         if (chunk > 0) {
7             res = combine(res, apply(it, chunk))
8         }
9         it.result = res
10     }
11 }
```

---

**Figure 4: The StealIterator interface**

```scala
10  trait StealIterator[T] {
11     def owner(): Int
12     def state(): A ∨ S ∨ C
13     def advance(step: Int): Int
14     def markStolen(): Unit
15     def hasNext: Boolean
16     def next(): T
17     def expanded(): {StealIterator[T], StealIterator[T]}
```

---

**Figure 5: The IndexIterator implementation**

```scala
16  abstract class IndexIterator[T](val owner: Int,
17     @volatile var progress: Int, val until: Int)
18  extends StealIterator[T] {
19  private var nextProgress = -1
20  private var nextUntil = -1
21  def state() = READ(progress)
22  def expanded: {StealIterator[T], StealIterator[T]}
23  def owner(): Int = {
24      case _ => A
25  }
26  def advance(s: Int) = if (state() ≠ A) -1 else {
27      val p = READ(progress)
28      val np = math.min(p + s, until)
29      if (~CAS(progress, p, np)) advance(s) else {
30          nextProgress = p
31          nextUntil = np
32          np - p |
33      } else {
34          markStolen() = ... // as before
35          hasNext: Boolean = nextProgress < nextUntil
36          next(): T = {
37              nextProgress = 1
38              elemAt(nextProgress - 1)
39          }
40      }
41  }
42  def elemAt(idx: Int): T = // returns element at idx
```

---

**Figure 6: The IndexIterator implementation**

```scala
def work(it: StealIterator[T]) = {
    var step = 0
    while (it.state() == A) {
        step = update(step)
        val chunk = it.advance(step)
        if (chunk > 0) {
            res = combine(res, apply(it, chunk))
        }
        it.result = res
    }
```
on efficient set membership or key-based lookup operations. The implementation of work-stealing iterators for flat hash-tables we show in this section is similar to the iterators for data-structures with fast indexing. Thus, the iteration state can still be represented with a single integer field progress, and invalidated with markStolen in the same way as with IndexIterator. The advance has to compute the expected number of elements between to array entries using the load factor if as follows:

```
40  def advance(step: Int) = {
41    val p = READ(progress)
42    val np = math.min(p + (step / 1f).toInt, until)
43    if (~CAS[progress, p, np]) advance(s)
44    else { nextProgress = p; nextUntil = np; np - p } }
```

We change the next and hasNext implementations so that they traverse the range between nextProgress and nextUntil as a regular single-threaded hash-table iterator implementation. This implementation relies on the hashing function to achieve good load-balancing, which is the common case with hash-table operations.

**TreeIterator.** Tree-like collections are of interest in parallelism because computation on them can easily be partitioned. Having shown that work-stealing iterators can be implemented for flat data structures, we turn our attention to a lock-free iterator implementation for tree data structures. For reasons of clarity, we focus on binary trees that store elements in external nodes, but this technique can be extended to n-ary trees with elements in internal nodes. We assume trees do not contain parent pointers.

Tree iterators typically mimic a tree traversal continuation by maintaining a stack of node references that describes the path from the root to the currently visited leaf. To implement advance and markStolen, the state of this stack needs to be visible. However, known concurrent stack implementations either rely on heap allocation for every push and pop operation [20], are specialized for high loads and scalability [25] or rely on a DCAS operation [2]. Fortunately, our problem is somewhat different – there is only a single worker invoking push and pop operations to implement advance and an unbounded number of stealers that call markStolen. We show an implementations using single-word CAS instructions and the amount of storage proportional to the depth of the tree.

A state diagram with several execution scenarios is shown in Figure 6. Horizontal movement depicts progress of the stealer, while vertical movement depicts worker progress. Each iterator contains an array serving as a stack. Consider a subtree with nodes A, B and C shown in Figure 6-1. The worker traverses the tree by pushing and popping nodes on the stack. To start traversing the subtree it pushes the node A, bringing the stack into the state 2. By subsequently pushing the leaf B it arrives into the state 3. The worker then decides to process the element stored in the leaf B. To commit to processing B, it pops it and replaces it with a special value TD (tree done), which denotes that all the elements below B will be processed, and arrives in state 4. More generally, the worker can choose to commit to an entire subtree in the same way. After processing B the worker goes into state 5 by switching the top of the stack A with a special value IN (inner node done), which denotes that the rest of the traversal proceeds in the right child. Worker then pushes C to the stack, arriving in the state 6. Again, it commits to processing the leaf C by popping it and replacing it with null, arriving in the state 7. Note that the worker now replaces a node with null, not TD. The rule is to replace left children with TD, and right with null. Otherwise, an observer cannot disambiguate between states (e.g., 5 and 7). The stealer steals by invalidating the stack entries. It starts from the bottom of the stack and replaces entries with special values that denote that the entry was stolen and encode the traversal direction. We say that the stealer snatches the entry. The stealer uses four special values SL, SR, SC and SN: SL and SR denote that the tree traversal at the corresponding tree level goes left or right, respectively. SC denotes that work on the corresponding subtree is completed. SN serves as a terminator. From any of the states 1-8 the stealer can snatch a value from the stack and replace it with one...
of these Stolen values, arriving into one of the states 9-16. Figure 7 shows the TreeIterator definition and the basic primitives needed to implement the algorithm. The worker uses push, pop and switch to atomically change the state of the stack. These operations take the previously observed stack value and replace it atomically. If successful, they update the private stack depth dep and the lastSwitch inner node that was last switched.

Implementations of advance and markStolen are shown in Figure 8. Method advance starts by checking if the entire tree was already processed (TD) and returns -1 if so. Otherwise, it reads the top of the stack t, and the previous and the next entries p and t. After that, it checks if the bottom of the stack was stolen. If it was, it helps complete the stealing and returns -1. Otherwise, it compares the top entry t and the next entry n of the stack against the following patterns. In case the current entry is some subtree tree and the next entry is null (line 78), the worker will attempt process all the elements in tree by popping it, given that tree is a leaf or there are less than step elements in it. This corresponds to the transition from the state 3 to 4 in Figure 6. The advance pushes the node on the private nextStack array, which the next and hasNext can then use. Note that for balanced trees we can always find the bound on the number of elements in a subtree from the number of elements in the entire tree and the depth of the subtree – we abstract this with a call to sizeBound. If neither of the conditions for processing a subtree holds, the worker descends by pushing the left child to the stack in line 86, going from state 2 to state 3. The remaining stack patterns, namely, (tree, TD), (IN, null) and (IN, TD) deal with the state transitions 4 to 5, 7 to 8 and 5 to 6, respectively. The worker uses a bigger range of data-structures more easily. In such cases, work-stealing iterators can also be implemented in a naive way using plain locks. Here, the worker acquires a lock during the execution of advance, and the stealers do the same during calls to markStolen. In most cases the state method can read the state of the iterator without requiring a lock, as shown in Figure 9 where locking has been used to implement a work-stealing iterator over ranges. While this approach excludes the possibility of lock-free work-stealing, it has the advantage of being applicable to a bigger range of data-structures more easily.

3.2 Operation kernels

We have seen in Figure 3 that the worker uses the work-stealing iterator to commit to processing chunks of elements. The apply call in line 8 cancels the details of how elements are processed. In this section we show that the apply implementation depends on a specific data-parallel operation instance. We focus our attention on the previously mentioned kernel abstraction.

Each data-parallel operation invocation site creates a kernel object, which describes how a chunk of elements is processed and what the resulting value is, how to combine values computed by different
workers and what the neutral element for the result is. The kernel interface is shown in Figure 10. The method `apply` takes the iterator and the number of elements estimated returned by `advance`. It uses the iterator to traverse those elements and compute the result of type `R`. The method `combine` is used to merge two different results and `zero` returns the neutral element. How these methods work is best shown through an example of a concrete data-parallel operation. The `foreach` operation takes a user-specified function object `f` and applies it in parallel to every element of the collection. Assume we have a collection `xs` of integers and we want to assert that each integer is positive:

```
x.foreach(x => assert(x > 0))
```

The generic `foreach` implementation is as follows:

```
val k = new Kernel[Int, Unit] {
  def apply(it: StealIterator[Int], chunk: Int) = {
    val sum = 0
    while (it.hasNext) sum += it.next() 
  }
}
```

Another example is the `fold` operation mentioned in the introduction and computing the sum of a sequence of numbers `xs`:

```
x.fold(0)((acc, i) => acc + i)
```

Operation `fold` computes a resulting value, which has the integer type in this case. Results computed by different workers have to be added together using `combine` before returning the final result. After inlining the code for the neutral element and the body of the folding operator, we obtain the following kernel:

```
new Kernel[Int, Int] {
  def apply(it: StealIterator[Int], chunk: Int) = {
    val sum = 0
    while (it.hasNext) sum += it.next() 
  }
}
```

Where `fold` returns a scalar value, some operations return entire collections as results. These operations use data-structure-specific `combiners` [41] to build the resulting collections. Combiners define methods `+=` for adding elements and `combine` for merging the elements of two combiners into a new combiner. The map operation transforms each element of the initial collection into a different element in the resulting collection by applying a user-specified transformation function `f`. In the following example a real vector `xs` is multiplied with a scalar value `c`:

```
x.map(x => x * c)
```

The generated kernel lazily creates the combiner and stores it into the result field of the work-stealing iterator. It then traverses the chunk, multiplies each element with `c` and adds it to the combiner:

```
def apply(i: TreeIterator[T], chunk: Int) = {
  def traverse(t: Tree): Int = {
    if (t.isLeaf) t.element
    else traverse(t.left) + traverse(t.right) 
  }
  val root = i.nextStack(0)
  traverse(root)
}
```

While the inlining shown in the previous examples avoids a dynamic dispatch to a function object, the while loop still contains two virtual calls to the work-stealing iterator. Generally, maintaining the iterator requires writes to memory instead of registers. It also prevents optimisations like loop-invariant code motion, e.g. hoisting the array bounds check that may be necessary when the iterator traverses an array.

For these reasons, we would like to inline the iteration into the `apply` method itself. This, however, requires knowing the specifics of the data layout in the underlying data-structure. Within this paper we rely on the macro system [8] to inline the body of the function `f` into the `Kernel` at the call-site:

```
def apply(it: StealIterator[T], chunk: Int) = {
  val sum = 0
  while (it.hasNext) sum += it.next() 
}
```

Another example is the `fold` operation introduced in the introduction and computing the sum of a sequence of numbers `xs`:

```
x.fold(0)((acc, i) => acc + i)
```

Operation `fold` computes a resulting value, which has the integer type in this case. Results computed by different workers have to be added together using `combine` before returning the final result. After inlining the code for the neutral element and the body of the folding operator, we obtain the following kernel:

```
new Kernel[Int, Combiner[Int]] {
  def combine(a: Int, b: Int) = a + b
  def zero = 0
  def apply(it: StealIterator[Int], chunk: Int) = {
    val sum = 0
    while (it.hasNext) sum += it.next() 
  }
}
```

Where `fold` returns a scalar value, some operations return entire collections as results. These operations use data-structure-specific `combiners` [41] to build the resulting collections. Combiners define methods `+=` for adding elements and `combine` for merging the elements of two combiners into a new combiner. The map operation transforms each element of the initial collection into a different element in the resulting collection by applying a user-specified transformation function `f`. In the following example a real vector `xs` is multiplied with a scalar value `c`:

```
x.map(x => c * x)
```

The generated kernel lazily creates the combiner and stores it into the result field of the work-stealing iterator. It then traverses the chunk, multiplies each element with `c` and adds it to the combiner:

```
def apply(i: IndexIterator[T], chunk: Int) = {
  def apply(it: StealIterator[T], chunk: Int) = {
    val u = it.nextUntil
    val v = it.nextUntil
    while (p < u) {
      sum = sum + p
      p += 1
    }
    sum
  }
}
```

While the inlining shown in the previous examples avoids a dynamic dispatch to a function object, the while loop still contains two virtual calls to the work-stealing iterator. Generally, maintaining the iterator requires writes to memory instead of registers. It also prevents optimisations like loop-invariant code motion, e.g. hoisting the array bounds check that may be necessary when the iterator traverses an array.

For these reasons, we would like to inline the iteration into the `apply` method itself. This, however, requires knowing the specifics of the data layout in the underlying data-structure. Within this paper we rely on the macro system [8] to inline the body of the function `f` into the `Kernel` at the call-site:

```
def apply(it: StealIterator[T], chunk: Int) = {
  val sum = 0
  while (it.hasNext) sum += it.next() 
}
```

Another example is the `fold` operation introduced in the introduction and computing the sum of a sequence of numbers `xs`:

```
x.fold(0)((acc, i) => acc + i)
```

Operation `fold` computes a resulting value, which has the integer type in this case. Results computed by different workers have to be added together using `combine` before returning the final result. After inlining the code for the neutral element and the body of the folding operator, we obtain the following kernel:

```
new Kernel[Int, Int] {
  def combine(a: Int, b: Int) = a + b
  def zero = 0
  def apply(it: StealIterator[Int], chunk: Int) = {
    val sum = 0
    while (it.hasNext) sum += it.next() 
  }
}
```

Where `fold` returns a scalar value, some operations return entire collections as results. These operations use data-structure-specific `combiners` [41] to build the resulting collections. Combiners define methods `+=` for adding elements and `combine` for merging the elements of two combiners into a new combiner. The map operation transforms each element of the initial collection into a different element in the resulting collection by applying a user-specified transformation function `f`. In the following example a real vector `xs` is multiplied with a scalar value `c`:

```
x.map(x => c * x)
```

The generated kernel lazily creates the combiner and stores it into the result field of the work-stealing iterator. It then traverses the chunk, multiplies each element with `c` and adds it to the combiner:

```
def apply(i: IndexIterator[T], chunk: Int) = {
  def apply(it: StealIterator[T], chunk: Int) = {
    val u = it.nextUntil
    val v = it.nextUntil
    while (p < u) {
      sum = sum + p
      p += 1
    }
    sum
  }
}
```

While the inlining shown in the previous examples avoids a dynamic dispatch to a function object, the while loop still contains two virtual calls to the work-stealing iterator. Generally, maintaining the iterator requires writes to memory instead of registers. It also prevents optimisations like loop-invariant code motion, e.g. hoisting the array bounds check that may be necessary when the iterator traverses an array.

For these reasons, we would like to inline the iteration into the `apply` method itself. This, however, requires knowing the specifics of the data layout in the underlying data-structure. Within this paper we rely on the macro system [8] to inline the body of the function `f` into the `Kernel` at the call-site:

```
def apply(it: StealIterator[T], chunk: Int) = {
  val u = it.nextUntil
  val v = it.nextUntil
  while (p < u) {
    sum = sum + p
    p += 1
  }
  sum
}
```

Another example is the `fold` operation introduced in the introduction and computing the sum of a sequence of numbers `xs`:

```
x.fold(0)((acc, i) => acc + i)
```

Operation `fold` computes a resulting value, which has the integer type in this case. Results computed by different workers have to be added together using `combine` before returning the final result. After inlining the code for the neutral element and the body of the folding operator, we obtain the following kernel:

```
new Kernel[Int, Int] {
  def combine(a: Int, b: Int) = a + b
  def zero = 0
  def apply(it: StealIterator[Int], chunk: Int) = {
    val sum = 0
    while (it.hasNext) sum += it.next() 
  }
}
```

Where `fold` returns a scalar value, some operations return entire collections as results. These operations use data-structure-specific `combiners` [41] to build the resulting collections. Combiners define methods `+=` for adding elements and `combine` for merging the elements of two combiners into a new combiner. The map operation transforms each element of the initial collection into a different element in the resulting collection by applying a user-specified transformation function `f`. In the following example a real vector `xs` is multiplied with a scalar value `c`:

```
x.map(x => c * x)
```
times when compared with the iterator approach.

**HashKernel.** The hash-table kernel is based on an efficient while loop like the array and range kernels, but must account for empty array entries. Assuming flat hash-tables with linear collision resolution, the while loop in the kernel implementation of the previously mentioned fold is as follows:

```java
while (p < u) {
    val elem = array(p)
    if (elem != null) sum = sum + elem
    p += 1
}
```

Work-stealing iterator implementations for hash-tables based on closed addressing are similar.

## 4. PERFORMANCE EVALUATION

Our design aims to reduce abstraction and scheduling penalties to a level where they are no longer noticeable, so we must verify that these goals are fulfilled. We will show a breakdown of performance improvements to identify the contributions of eliminating boxing, dynamic dispatch and the iterator abstraction. Doing so shows that these overheads really exist and that they can be efficiently eliminated. We will also introduce a range of different workloads to evaluate the efficiency of our load-balancing approach.

We compare against imperative sequential programs written in Java, against existing Scala Parallel Collections and a corresponding imperative C version where the two implementations can easily be compared. We start with microbenchmarks addressing specific data structures and data-parallel operations, and then move on to larger data-parallel applications. We rely on the established performance evaluation methodologies [23]. We perform the evaluation on the Intel i7-2600 quad-core 3.4 GHz processor with hyperthreading and an 8-core 1.2 GHz UltraSPARC T2 with 64 hardware threads. Aside from the different number of cores and processor clock, another important difference between these two architectures is in the memory throughput - i7-1600 has a single dual-channel memory controller, while the UltraSPARC T2 has four dual-channel memory controllers.

![Figure 13: Uniform workload microbenchmarks I on Intel i7 and UltraSPARC T2; A - ParRange.foreach, B - ParRange.fold](image1)

![Figure 14: Uniform workload microbenchmarks II on Intel i7 and UltraSPARC T2; C - ParArray.fold, D - Conc.fold](image2)

We start by showing several microbenchmarks for specific data-parallel operations on specific collection types. The microbenchmarks in Figures 13 and 14 have a cheap, uniform workload – the amount of computation per each element is fixed and small enough to notice any abstraction penalties discussed earlier. The microbenchmark in Figure 13A consists of a data-parallel foreach loop that occasionally sets a volatile flag (without a potential side-effect the JIT compiler may optimize away the loop in the kernel).

Due to the genericity of the existing Scala Parallel Collections framework boxing occurs in this microbenchmark. The speed gain for a range-specialized work-stealing kernel and a work-stealing kernel specialized for ranges from Figure 11. In this benchmark Parallel Collections do not instantiate primitive types and hence do not incur the costs of boxing, but still suffer from iterator and function object abstraction penalties. Inlining the function object into the while loop for the generic kernel shows a considerable performance gain. However, the range-specialized kernel outperforms the generic kernel by 25% on the i7 and 15% on the UltraSPARC (note the log scale). Figure 13B shows the same comparison for parallel ranges and the fold operation shown in the introduction:

```java
for (i <- (0 until N).par) {
    if ((i + i) & 0xffffff == 0) flag = true
}
```

Figure 13A shows a comparison between Parallel Collections, a generic work-stealing kernel and a work-stealing kernel specialized for ranges from Figure 11. In this benchmark Parallel Collections do not instantiate primitive types and hence do not incur the costs of boxing, but still suffer from iterator and function object abstraction penalties. Inlining the function object into the while loop for the generic kernel shows a considerable performance gain. However, the range-specialized kernel outperforms the generic kernel by 25% on the i7 and 15% on the UltraSPARC (note the log scale). Figure 13B shows the same comparison for parallel ranges and the fold operation shown in the introduction:

```java
(0 until N).par.map(fold(_ + _))
```
The Parallel Collections rely on a Splitter abstraction that determines their set membership. Rendering a high resolution image of the Mandelbrot set in serial, the prime number computation mentioned in the introduction selects a sequence of potentially up to a single element, which occurs in parts outside the circle \( x^2 + y^2 = 4 \) – the first 97% of elements have little or no work associated with them, while the rest of the elements require a high amount of computation. Since most of the work is located in a sequence of elements smaller than the threshold, the existing Parallel Collections scheduler only yields a speedup on UltraSPARC when the number of processors used exceeds 16.

More benign irregularities present in some problems have workloads increasing monotonically, described by a function such as \( \sqrt{N} \). The prime number computation mentioned in the introduction is shown in Figure 15B – a performance difference is 15% on i7 and 10% on the UltraSPARC in favour of specialized work-stealing. However, as the irregularity grows, this difference becomes larger as shown in Figure 15C, where the workload of the \( n \)-th element grows with the function \( n^{16} \).

To show that these microbenchmarks are not just contrived examples, we show several larger benchmark applications as well. Cheap, uniform workloads occur in practice with linear algebra applications and numerical computations. In Figure 16 we show performance results for an application computing a standard deviation of a set of measurements. The relevant part of it is as follows:

```scala
val mean = measurements.sum / measurements.size
val variance = measurements.aggregate(0.0)(_ + _)
acc => acc + (x - mean) * (x - mean)
```

As in previous experiments, Parallel Collections scale but have a large constant penalty due to boxing. On UltraSPARC boxing additionally causes excessive memory accesses resulting in non-linear speedups for higher parallelism levels (\( P = 32 \) and \( P = 64 \)). Irrregular workloads exist in practical applications as well. We first show an application that renders an image of the Mandelbrot set in parallel. The Mandelbrot set is irregular in the sense that all points outside the circle \( x^2 + y^2 = 4 \) are not a part of the set, but all the points within the circle require some amount of computation to determine their set membership. Rendering a high resolution image a
4. CONCLUSION

The conclusions from the previous section are twofold. First, the abstraction penalties associated with generic data-parallel frameworks can be eliminated. This is important from the perspective of achieving optimal parallelization—additional processors should not be wasted on compensating for the abstraction overheads. Furthermore, schedulers that work by preemptively creating chunks of elements and scheduling them for execution to allow work-stealing incur higher scheduling penalties. These scheduling penalties are usually overcome by setting a threshold on the chunk size, but this in turn makes them less applicable to highly irregular workloads. Such a scheduling approach is a direct consequence of the choice of abstraction in alternative frameworks—Intel TBB relies on the split operation, Scala Parallel Collections rely on Splitters and the upcoming Java 8 parallel collections rely on Spliterator. This is a potential cause for concern, since those frameworks yield a suboptimal speedup for certain workloads.

In the same way as the parallel application authors using a high-level data-parallel framework should not be concerned with the abstraction penalties in their code, they should not worry about optimizing the code to fit a specific workload pattern, particularly when that the irregularity is the property of the data itself.

Figure 17: Mandelbrot set computation on Intel i7 and UltraSPARC T2

Figure 18: Raytracing on Intel i7 and UltraSPARC T2

Figure 19: Triangular matrix multiplication on Intel i7 and UltraSPARC T2

The last application we choose is triangular matrix multiplication, in which a triangular $N \times N$ matrix is multiplied with a vector of size $N$. Both the matrix and the vector contain arbitrary precision values. This application has a less irregular workload shown in Figure 2—the amount of work to compute the $n$-th element in the resulting vector is $w(n) = n$. We call this workload is triangular. Figure 19 shows a comparison of the existing Parallel Collections scheduler and specialized work-stealing. The performance gap is smaller but still exists, Parallel Collections being 18% slower on the i7 and 20% slower on the UltraSPARC. The downsides of fixed size threshold and preemptive chunking are thus noticeable even for less irregular workloads, although less pronounced.

part of which contains the described circle thus results in an irregular workload.

We show the running times of rendering two different Mandelbrot set images in Figure 17. In Figure 17A the aforementioned computationally demanding circle is in the lower left part of the image, whereas in Figure 17B the same circle is situated in upper right part of the image. In both cases the fixed threshold on the chunk sizes proves detrimental. We can see a similar effect as in the Figure 15A—with a fixed threshold there is only a 50% to 2x speedup until $P$ becomes larger than 16. The subsequent speedup due to the chunk size threshold being inversely proportional to the number of processors remains suboptimal for $P > 16$ since only a subset of all the processors gets to work on the more expensive chunks.

In Figure 18 we show the performance of a parallel raytracer, implemented using existing Parallel Collections and specialized work-stealing. Raytracing renderers project a ray from each pixel of the image being rendered, and compute the intersection between the ray and the objects in the scene. The ray is then reflected several times up until a certain threshold. This application is inherently data-parallel—computation can proceed independently for different pixels. The workload characteristics depend on the placement of the objects in the scene. If the objects are distributed uniformly throughout the scene, the workload will be uniform. The particular scene we choose contains a large number of objects concentrated in one part of the image, making the workload highly irregular. The fixed threshold on the chunk sizes causes the region of the image containing most of the objects to end up in a single chunk, thus eliminating most of the potential parallelism. On the i7 Parallel Collections barely manage to achieve the speedup of 2x, while the specialized work-stealing easily achieves up to 4x speedups. For higher parallelism levels the chunk size becomes small enough to divide the computationally expensive part of the image between processors, so the plateau ends at $P = 32$ on UltraSPARC. The speedup gap still exists at $P = 64$—existing Parallel Collections scheduler is 3x slower than specialized work-stealing.

The last application we choose is triangular matrix multiplication, in which a triangular $N \times N$ matrix is multiplied with a vector of size $N$. Both the matrix and the vector contain arbitrary precision values. This application has a less irregular workload shown in Figure 2—the amount of work to compute the $n$-th element in the resulting vector is $w(n) = n$. We call this workload is triangular. Figure 19 shows a comparison of the existing Parallel Collections scheduler and specialized work-stealing. The performance gap is smaller but still exists, Parallel Collections being 18% slower on the i7 and 20% slower on the UltraSPARC. The downsides of fixed size threshold and preemptive chunking are thus noticeable even for less irregular workloads, although less pronounced.
6. REFERENCES


