Translating Imperative Code to MapReduce

Cosmin Radoi
University of Illinois
cos@illinois.edu

Stephen J. Fink Rodric Rabbah
IBM T.J. Watson Research Center
{sjfink,rabbah}@us.ibm.com

Manu Sridharan
Samsung Research America
m.sridharan@samsung.com

Abstract

We present an approach for automatic translation of sequential, imperative code into a parallel MapReduce framework. Automating such a translation is challenging: imperative updates must be translated into a functional MapReduce form in a manner that both preserves semantics and enables parallelism. Our approach works by first translating the input code into a functional representation, with loops succinctly represented by `fold` operations. Then, guided by rewrite rules, our system searches a space of equivalent programs for an effective MapReduce implementation. The rules include a novel technique for handling irregular loop-carried dependencies using group-by operations to enable greater parallelism. We have implemented our technique in a tool called MOLD. It translates sequential Java code into code targeting the Apache Spark runtime. We evaluated MOLD on several real-world kernels and found that in most cases MOLD generated the desired MapReduce program, even for codes with complex indirect updates.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors—Code generation, Compilers, Optimization; D.1.3 [*Programming techniques*]: Concurrent Programming; I.1.4 [*Symbolic and algebraic manipulation*]: Applications; I.2.2 [*Artificial intelligence*]: Automatic Programming—Program transformation, Program synthesis

General Terms Performance, Languages

Keywords Program Translation; Rewriting; Imperative; Functional; MapReduce; Scala

1. Introduction

Over the past decade, the MapReduce programming model has gained traction both in research and in practice. Mainstream MapReduce frameworks [1, 9] provide significant advantages for large-scale distributed parallel computation. In particular, MapReduce frameworks can transparently support fault-tolerance, elastic scaling, and integration with a distributed file system.

Additionally, MapReduce has attracted interest as a parallel programming model, independent of difficulties of distributed computation [24]. MapReduce has been shown to be capable to express important parallel algorithms in a number of domains, while still abstracting away low-level details of parallel communication and coordination.

This paper addresses the challenge of automatically translating sequential imperative code into a parallel MapReduce framework. An effective translation tool could greatly reduce costs when re-targeting legacy sequential code for MapReduce. Furthermore, a translator could simplify the process of targeting MapReduce in new programs: a developer could concentrate on sequential code, letting the translator handle parallel computation using MapReduce.

Translating an imperative loop to the MapReduce model inherits many of the difficulties faced by parallelizing compilers, such as proving loops free of loop-carried dependencies. However, the MapReduce framework differs substantially from a shared-memory parallel loop execution model. Notably, MapReduce implies a *distributed memory* programming model: each mapper and reducer can operate only on data which is "local" to that function. So, an automatic translator must at least partition memory accesses in order to create local mapper and reducer functions which do not rely on shared memory.

Additionally, the communication model in MapReduce is more limited than traditional distributed memory parallel programming with message-passing. Instead, mappers and reducers communicate via a *shuffle* operation, which routes mapper outputs to reducer inputs based on on key fields in the data. These restrictions allow practical MapReduce frameworks to run relatively efficiently at large scale; however, they also introduce constraints on the programmer and challenges for an automatic translator.

Figure 1 illustrates the design of our translator. An input program is first translated into Array SSA form [17] which facilitates the derivation into an lambda-calculus-style functional representation. In contrast with Appel's and Kelsey's

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10…$15.00.
http://dx.doi.org/10.1145/2660193.2660228

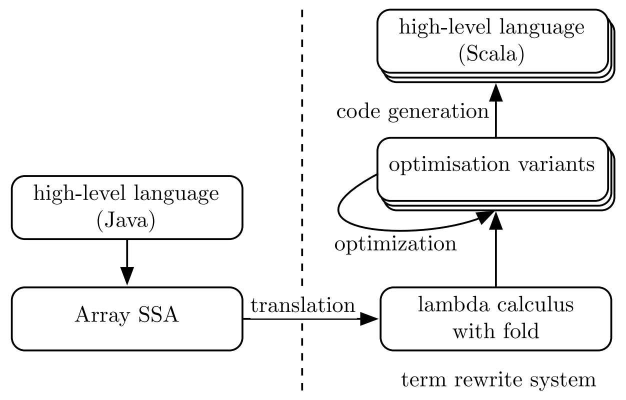

Figure 1. Overview of our translation system.

work on converting from SSA to functional code [6, 15], our translation uses a `fold` operator to maintain the structure of loops, which is essential for later transformations.

The initial "lambda plus `fold`" representation of a program is still far from an effective MapReduce program. While `fold` operations could be implemented using reducers, the lack of mappers hinders parallelism, and the code still operates on shared data structures. To address these problems, we employ a rewriting system to generate a large space of MapReduce programs. The rewrite rules govern where mapper constructs can be introduced in a semantics-preserving manner. Critically, in more complex cases where loop iterations access overlapping locations, we exploit the MapReduce *shuffle* feature to group operations by accessed locations, exposing much more fine-grained parallelism than previous approaches. Given the rewriting rules, our system performs a heuristic search to discover a final output program, using a customizable cost function to rank programs.

We have implemented our techniques in a tool named MOLD, which transforms input Java programs (i.e., Java methods) into Scala programs that can be executed either on a single computing node via parallel Scala collections, or in a distributed manner using Spark, a popular MapReduce framework [2, 31]. MOLD leverages the WALA analysis framework [5] to generate Array SSA and implements a custom rewriting engine using the Kiama [27] library. In an experimental evaluation, we tested our tool on a number of input kernels taken from real-world Java and MapReduce benchmarks. In most cases, MOLD successfully generated the desired MapReduce code, even for codes with complex indirect array accesses that cannot be handled by previous techniques. To our knowledge, MOLD is the first system that can automatically translate sequential implementations of canonical MapReduce programs like "word count"(see Section 2) into effective MapReduce programs.

This paper makes the following contributions:

- We give an automatic translation from imperative array-based code to a functional intermediate representation, amenable to generating MapReduce programs. Critically, the translation represents loops as `fold` operations, preserving their structure for further optimization.
- We present a rewriting system to generate a broad space of equivalent MapReduce programs for a given imperative program from our functional IR. The space is explored via heuristic search based on a cost function that can be customized for different backends.
- We describe a rewrite rule in our system that introduces `groupBy` operations to effectively handle complex indirect array accesses. This novel technique is critical for handling basic MapReduce examples like "word count."
- We present an implementation of our techniques in a tool MOLD, and an experimental evaluation showing its ability to handle complex input programs beyond those in the previous work.

2. Motivating Example

We assume the reader is familiar with the MapReduce model. Here, we briefly review MapReduce details as presented by Dean and Ghemawat [9] before presenting an overview of our approach using their `wordcount` example.

MapReduce background. In the MapReduce framework, the programmer defines a *map* function and a *reduce* function. The functions have the following types:

$$\texttt{map} : \langle k1, v1\rangle \rightarrow \texttt{List}[k2, v2]$$
$$\texttt{reduce} : \langle k2, \texttt{List}[v2]\rangle \rightarrow \texttt{List}[v2]$$

The MapReduce framework relies on a built-in, implicit *shuffle* function to route the output of the mappers to the input of the reducers. Logically, the shuffle function performs a *group-by* operation over the map outputs. That is, for each distinct key k of type $k2$ output by a mapper function, the shuffle function collects all the values of type $v2$ associated with k, forms a list l of these values, and sends the resulting pair (k, l) to a reducer.

Dean and Ghemawat present `wordcount` as an example to define mappers and reducers. In their `wordcount` example, the *map* function takes as input a document name and a String which holds the document contents. For each word w in the contents, the mapper emits a pair $(w, 1)$. The shuffle operation will create a list of the form $[1, 1, \ldots, 1]$ for each word w, grouping the map output values (all ones) by word. Then the reducers simply sum up the number of ones present in the list associated with each word. The resulting sums represent the frequency count for each word.

The built-in *shuffle* or *group-by* operation plays a central role, for at least two reasons. Firstly, the shuffle operation encapsulates all communication between nodes. In traditional distributed memory parallel computing models, the programmer must explicitly pass messages or manage remote memory access in order to express communication. Instead, in MapReduce, the programmer simply defines functions which

```
1  Map<String,Integer> wordCount(List<String> docs) {
2    Map<String,Integer> m = new HashMap<>();
3    for (int i = 0; i < docs.size(); i++) {
4      // simplified word split for clarity
5      String[] split = docs.get(i).split(" ");
6      for (int j = 0; j < split.length; j++) {
7        String w = split[j];
8        Integer prev = m.get(w);
9        if (prev == null) prev = 0;
10       m.put(w, prev + 1);
11     }
12   }
13   return m;
14 }
```

Figure 2. Java word count program.

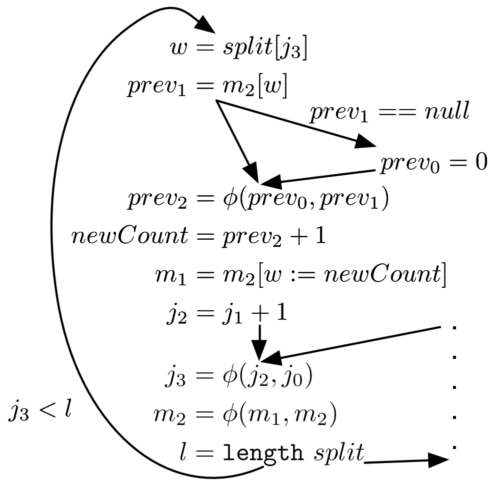

Figure 3. Array SSA form for the inner loop of Figure 2.

produce and consume tuples, and the framework transparently implements the necessary communication. MapReduce cannot express every possible parallel algorithm and communication pattern – but when MapReduce does apply, it relieves the programmer from the burden of managing communication explicitly, resulting in much simpler parallel programming. Secondly, we note that the shuffle operation can be extremely expensive, and can limit performance in many use cases if not managed carefully. Naïve use of MapReduce can result in all-to-all communication patterns whose overhead can overwhelm any speedups from parallel computation.

In the remainder of this paper, we focus on MapReduce primarily as a convenient model for expressing parallel computation. In particular, we consider the challenge of automatically translating sequential code into a MapReduce parallel programming model. We will not address issues specific to large-scale distributed Map-Reduce deployments, such as fault-tolerance, elasticity, and distributed file systems.

Overview of our approach. Consider the challenge of automatically generating effective MapReduce code for `wordcount`. Figure 2 shows the sequential Java code, our starting point. The program iterates through a list of documents `docs`, accumulating the word counts into the `m` map.

Parallelizing the Figure 2 example is difficult because of the updates to the shared `m` map in different loop iterations—naïvely running loop iterations in parallel would cause a race conditions on `m` if two iterations try to simultaneously update a word's count. Some parallelism might be achieved by splitting the `docs` list of documents into chunks, and computing word counts for each chunk simultaneously.

However, this transformation still leaves the sequential work of combining the word counts from the different chunks into final, global word counts. In contrast, the standard MapReduce word count program, which MOLD can generate, enables parallel accumulation of final word counts, by assigning the accumulation task for different ranges of words to different reducers.

We first consider generating a MapReduce program for the inner loop of Figure 2 (lines 6–11), which computes word counts for an array of words `split`. The first step of our technique is to translate the input program into a functional representation via Array SSA form [17]. Figure 3 gives the Array SSA form for our inner loop. Note that here, the `m` map is the "array" being updated by the loop. Our implementation treats Java arrays, `Lists`, and `Maps` in a unified manner as mappings from keys to values. This form can be seen as functional if every write to a location of `m` is treated as creating a new copy of the map with a new value for the appropriate word.

After constructing Array SSA form, MOLD translates the program to a more explicitly functional form. Unlike previous translations from SSA to functional code [6, 15], our technique preserves the structure of loops by translating them using the standard `fold` operation from functional programming. MOLD transforms each non-ϕ SSA assignment into a `let` statement. E.g., $w = split[j_3]$ is transformed to $let\ w = split[j_3]\ in \ldots$.

Each loop (branching and ϕ instructions) is transformed into a fold going over the domain of the initial loop, with its combining operation taking the tuple of ϕ values as one argument and the loop's index as the other, and returning a new tuple of values. In our example, the fold goes over the range of integers from 0 to the length of the `split` array, with a combining operation that takes as arguments m_2 and j_3 and returns m_1, i.e., the in-loop branch of $\phi(m_1, m_2)$.

Each remaining ϕ value with its corresponding `branch` instruction is rewritten into into a predicated `if` assignment. E.g. $\phi(prev_0, prev_1)$ with the corresponding branch condition $prev_1 == null$ is transformed to:

$$\texttt{if}\ prev1 == null\ \texttt{then}\ prev0\ \texttt{else}\ prev1$$

Thus, the SSA-form code in Figure 3 is converted to:

```
let $updatedCount = \lambda\, m_2\, j_3$ .
  let $w = split[j3]$ in
    let $prev1 = m_2[w]$ in
      let $prev2 =$ if $prev1 == null$ then $0$ else $prev1$ in
        let $newCount = prev2 + 1$ in
          let $m_1 = m_2[w := newCount]$ in
            $m_1$
in fold $m_0$ $updatedCount$ $(0...($length $split))$
```

Next, MOLD explores the space of possible optimizing transformations that can be applied to the code above. The transformations are expressed as rewrite rules and are detailed in Section 5. For now, we will focus on the particular transformations that take the functional, yet sequential, code above and turn it into MapReduce form.

After inlining some of the `let` expressions and renaming variables for readability, we get to:

```
let $updatedCount = \lambda\, m\, j$ .
  let $w = split[j]$ in
    let $prev = m[w]$ in
      $m[w :=($if $prev == null$ then $0$ else $prev) + 1]$
in fold $m$ $updatedCount$ $(0...($length $split))$
```

One initial observation is that the fold traverses a range of integers instead of the `split` collection itself. Thus, MOLD transforms the code such that the fold traverses the collection without indirection:

```
let $updatedCount = \lambda\, m\, w$ .
  let $prev = m[w]$ in
    $m[w :=($if $prev == null$ then $0$ else $prev) + 1]$
in fold $m$ $updatedCount$ $split$
```

Next, MOLD identifies common update idioms and lifts them to more general functional operations. In our example, the `m` map is lifted to return `zero` for non-existent keys. Thus, the `if` condition returning either zero or the previous value in the map becomes unnecessary, and MOLD replaces it with just the map access. Thus, MOLD transforms the program to:

```
let $updatedCount = \lambda\, m\, w\, .\, m[w := m[w] + 1]$ in
  fold $m$ $updatedCount$ $split$
```

The `fold` call takes the initial `m` map, the `updatedCount` function, and the `split String` array, and computes a new map with updated word counts. The `updateCount` accumulator function takes a map `m` and a word `w` as arguments, and returns a new map that is identical to `m` except that the count for `w` is incremented.

The functional form above is semantically equivalent to the original imperative code, but unfortunately exposes no parallelism, since the `fold` operation is sequential. Furthermore, trying to parallelize the fold directly would not work as the accesses to the `m` collection do not follow a regular pattern, i.e. `w` may have any value, independent of the underlying induction variable of the loop.

A common way to parallelize such code is to take advantage of the commutativity of the updating operation and tile the fold, namely the loop [19]. While this solution does expose parallelism, it is coarse-grained, may not be applicable in the presence of indirect references, and does not match the MapReduce model of computation.

MOLD generates this tiled solution, but it also explores a different parallelization avenue: instead of avoiding the non-linear `w` value, a program can inspect it [8] to reveal parallelism. Parts of computation operating on distinct `w` values are independent so they can be executed in parallel. Thus, our example is also equivalent to:

```
let $grouped = ($groupBy $id\ split)$ in
  map$(\lambda\, k\, v\, .\,$fold $m[k](\lambda\, y\, x\, .\, y + 1)\ v)\ grouped$
```

The inner fold is only computing the size of the `v` list of words, and adding it to the previous value in m. Assuming an initially empty map `m`, in the end, the rewrite system produces the following program as part of its output:

```
let $grouped = ($groupBy $id\ split)$ in
  map$(\lambda\, k\, v\, .\, v.size)\ grouped$
```

The sequential `fold` operation has been completely eliminated. Instead, we are left with the equivalent of the canonical MapReduce implementation of word counting. The `groupBy` operation yields a map from each word to a list of copies of the word, one for each occurrence; this corresponds to the standard mapper. Then, the `map` operation takes the grouped data and outputs the final word count map, corresponding to the standard reducer.[1] Given a large number of documents spread across several servers, the `groupBy` task can be run on each server separately, and the `map` task in the reducer can also be parallelized across servers, given the standard "shuffle" operation to connect the mappers and reducers. The standard MapReduce implementation does not construct explicit lists with a `groupBy`, but instead sends individual word instances to the reducers using "shuffle." MOLD further optimizes the above program to generate this more efficient implementation.

In this section we discussed the inner loop of the input `wordcount` program. In Section 5 we discuss how this integrates with the `fold` for the outer loop, and how MOLD brings the entire program to an optimized MapReduce form.

[1] The value of the `map` operation is itself a map, with the same keys as the input map; see Section 5.1 for details.

Finally, MOLD takes what it considers the best generated code versions and translates them to Scala. The best generated version, exactly as it is output by MOLD, is:

```
docs
  .flatMap({ case (i, v9) => v9.split(" ") })
  .map({ case (j, v8) => (v8, 1) })
  .reduceByKey({ case (v2, v3) => v2 + v3 })
```

The generated code's computation structure is similar to classic MapReduce solutions for the WordCount problem. Each document is split into words, which are then mapped to $(word, 1)$ pairs. The `reduceByKey` groups the "1" values by their associated *word*, and then reduces each group by $+$, effectively counting the number of words. The code above uses custom collection classes which implement interfaces that provide higher-order functional operators (*e.g.*, `map`). The system provides three implementations of these operators. The code can execute using Scala sequential collections, Scala parallel collections, or on the Spark MapReduce framework [31].

3. Generating Functional IR

In this section, we describe the initial stages of our translation system, which convert an input program into a functional intermediate representation via Array SSA form.

3.1 Array SSA

The standard SSA representation enables straightforward tracking of def-use relationships by ensuring each variable has a single static definition which reaches all uses. However, standard SSA does not reflect modification to the stack or heap, such as the effects of array writes. Array SSA form [17] extends traditional SSA form with constructs that represent modifications to array contents.

Array SSA form generates a new name for an array variable on each write. Any subsequent reads from the array use the new name, simplifying tracking of data flow though the array. In contrast to the original work on Array SSA, our system treats arrays as being immutable data structures, as we aim to translate the program into a functional representation. Hence, each array write is modeled as creating a fresh array with a new SSA name. With this functional model, our Array SSA form is simplified compared to the previous work [17]; we do not require a ϕ statement after each array write, and we do not require a special semantics for ϕ statements over array variables.

MOLD generalizes the collections covered by Array SSA to include maps and lists. Thus, new SSA names are also introduced when putting an element into a map, or adding or putting an element into a list. This allows MOLD to treat arrays, lists, and maps in a unified manner as mappings from keys to values.

The Array SSA form guarantees that, as long as no aliases are introduced through callee or caller functions, and no pointer to an array is retrieved/stored through another object or array, the arrays (and the lists and maps) can be treated as a values. Note that our translation to Array SSA form does not check for aliasing of array variables introduced in unobserved code, e.g., via caller or callee functions. Our implementation currently assumes that unobserved code does not introduce such aliasing.

$$
\begin{array}{rl}
\text{Var} & a|b|c|\dots \\
 & |\langle \text{Var}, \text{Var}, \dots \rangle \\
\text{Exp} & \lambda\, \text{Var}[: \text{Type}]\,.\, \text{Exp} \\
 & |\, \texttt{let}\ \text{Var} = \text{Exp}\ \texttt{in}\ \text{Exp} \\
 & |\ \text{Exp}\ \text{Exp} \\
 & |\langle \text{Exp}, \text{Exp}, \dots \rangle \\
 & |\, \text{Exp}[\text{Exp}] \\
 & |\, \text{Exp}[\text{Exp} := \text{Exp}] \\
\text{Type} & A \mid B \mid C \mid \dots \\
 & |\ \text{Type} \to \text{Type} \\
 & |\ \langle \text{Type}, \text{Type}, \dots \rangle \\
 & |\ \text{Type}[\text{Type}]
\end{array}
$$

Figure 4. Lambda calculus IR

3.2 Functional IR

Previous work by Appel [6] and Kelsey [15] observed that a program in SSA form is essentially a functional program. This observation can be extended in a straightforward manner to the Array SSA form described above, yielding a method for translating to a functional intermediate form. However, the aforementioned translation techniques are not suitable for our purposes, as they do not preserve the structure of loops (which are translated to general recursive function calls). Transformation of loop computations is critical to introducing parallelism, so our system relies on knowledge of loop structure. Here, we give a translation from our Array SSA form to a functional IR that includes a built-in `fold` construct, used to preserve the structure of loops.

MOLD's intermediate representation, shown in Fig. 4, is a typed lambda calculus. For brevity, throughout the rest of the paper, we omit the types when they are clear from the context. The IR is enriched with a tuple constructor, e.g., $\langle e_1, e_2, \dots \rangle$. Tuples behave like functions (e.g., $\langle f, g \rangle\ \langle a, b \rangle = \langle f\ a, g\ b \rangle$), and are unwrapped on application (e.g., $(\lambda\, x\, y\,.\, y)\ \langle a, b \rangle = b$). We sometimes use a tuple notation in the parameter part of lambda expression for highlighting a particular grouping of the parameters, but it can be interpreted in a curried fashion, e.g., $\lambda \langle x, y \rangle z\,.\, e\ =\ \lambda\, x\ y\ z\,.\, e$. The IR has `let` expressions which allow simple pattern-matching on tuples (e.g., $\texttt{let} \langle x_1, x_2 \rangle = \langle e_2, e_2 \rangle\ \texttt{in}\ x_1 + x_2$). $a[b]$ and $a[b := c]$ are read and write accesses at index b of map (array) a.

(data structures)

$$\begin{aligned}
\mathtt{M}[A] &: \text{bag (multiset) with values of type } A \\
\mathtt{M}[K,V] &: \text{map with keys of type } K \text{ and values of type } V \\
&\mathtt{M}[K,V] \text{ is an } \mathtt{M}[\langle K,V\rangle]
\end{aligned}$$

(higher order functions)

$$\begin{aligned}
\mathtt{fold} &: B \to (\langle B, A\rangle \to B) \to (\mathtt{M}[A] \to B) \\
\mathtt{map} &: (A \to B) \to (\mathtt{M}[A] \to \mathtt{M}[B]) \\
\mathtt{map} &: (\langle K,V\rangle \to W) \to (\mathtt{M}[K,V] \to \mathtt{M}[K,W]) \\
\mathtt{groupBy} &: (A \to K) \to (\mathtt{M}[A] \to \mathtt{M}[K, \mathtt{M}[A]])
\end{aligned}$$

(functions and operations)

$$\begin{aligned}
\mathtt{zip} &: \langle \mathtt{M}[K,A], \mathtt{M}[K,B]\rangle \to \mathtt{M}[K, \langle A,B\rangle] \\
\mathtt{zip} &: \langle \mathtt{M}[A], \mathtt{M}[B]\rangle \to \mathtt{M}[\langle A,B\rangle] \\
\mathtt{zip} &: \langle \mathtt{M}[K], \mathtt{M}[K,B]\rangle \to \mathtt{M}[K,B] \\
\mathtt{++} &: \langle \mathtt{M}[K,A], \mathtt{M}[K,A]\rangle \to \mathtt{M}[K,A] \\
&\text{addition with replacement of matching keys} \\
\oplus \;\; \boxplus \quad &\text{monoid plus operators} \\
\oplus &: \langle \mathtt{M}[K,A], \mathtt{M}[K,A]\rangle \to \mathtt{M}[K,A] \\
a \oplus b &= (\mathtt{map}\, \lambda\, k \langle x\ y\rangle \,.\, x \boxplus y)(a\, \mathtt{zip}\, b)
\end{aligned}$$

Figure 5. Builtin data structures and operators

Figure 5 shows the built-in data structures along with signatures for the functions and operations operating over them. These constructs are mostly well-known, and we will describe them as needed throughout the paper. The translation from Array SSA relies on left-`fold`, a higher-order function that takes an initial element (a zero) of type B and a combining operation $B \to A \to B$, and returns a function which reduces the elements of a collection of A elements into a value of type B by applying the operation from left to right.

MOLD transforms Array SSA form to the functional IR by applying the rules in Figure 6. The instructions in the CFG are visited in topological order. SCCs are considered nodes in the order, and are visited in turn in topological order. We use the $\prec$ operator in Figure 6 to reflect this ordering: $s \prec R$ matches a statement s followed by remaining statements R in the topological ordering.

We first discuss the non-loop rules, which are mostly straightforward. SSA assignments $x = E$ are transformed to `let`s. Any branch instruction is skipped, left to be handled when reaching its associated ϕ. The `return` instruction is replaced with the returned variable, which eventually sits at the innermost level of the `let` nest.

An `if` statement in the original code corresponds a branching statement followed by a set of ϕ instructions in SSA form. MOLD transforms each of the ϕ instructions corresponding to an `if` into a functional `if` with the condition coming from the branching instruction, and the branches being the arguments of the ϕ instruction. As the instructions are visited in a topological order, the variables holding the result for each of the two branches are already available in scope. Computing the results for the `if` before the instruction is not an issue from a semantic perspective because in our representation we have no side effects and no recursion (except for structured `fold` recursion). Also, performance is not hurt as the two branches are inlined by the rewrite system in a latter step.

The more complex rule translates loops to applications of the `fold` operator. In Figure 6, a loop is specified in terms of its ϕ variables, which include the index variable i and other variables $r_1, r_2, \ldots$ updated in the loop body. These variables characterize all possible effects of the loop visible to subsequent code. For each ϕ variable r_k, we use r'_k to refer to the value coming from outside the loop, and r''_k for the new value produced by the loop. The loop gets translated to a `fold` over the domain of values for the index variable, from i' to l, the loop bound, in Figure 6. The combining function takes as arguments the current r_k values and loop index, runs the body of the loop E once for those values, and returns the new r_k values. The initial value for the `fold` is a tuple of the r'_k values coming from outside the loop. Any loop with a loop-invariant domain (i.e., the domain does not depend on the loop's ϕ functions) can be translated to a `fold`. Our current implementation only handles indexed collection iteration with a range of numbers with stride 1.

All other SSA instructions (function calls, operations, etc.) are transformed to lambda calculus in the intuitive straightforward manner. Function calls are not inlined.

4. Translation system

Exploration and refinement. The transformation described in the previous section generates a lambda calculus representation of the original program but it is still far from MapReduce form. The loops in the original program are now sequential `fold`s that do not expose any parallelism. In order to get to MapReduce form, MOLD explores the space of semantically equivalent programs obtained by applying a set of program transformation rules.

MOLD distinguishes between *refinement* and *exploration* rewrite rules. *Refinement* rules make definite improvements to the input term, e.g., eliminating a redundant operation such as $\mathtt{fold}\, r\, \lambda \langle r, \langle k, v\rangle\rangle \,.\, r[k := v]$. *Exploration* rules may either improve the original code or bring it to a form that allows other rules to apply, e.g., loop fission is not necessarily an optimization but may allow another rule to eliminate part of the original loop.

Exploration rules are treated as transitions between states, i.e., applying a transition rule generates a new state in the system. Refinement rules do not generate new states but are applied exhaustively to the output of an exploration

$$\mathcal{L}(x = E \prec R) \rightarrow \texttt{let}\, x = \mathcal{L}(E)\,\texttt{in}\,\mathcal{L}(R)$$

$$\mathcal{L}(a[x := y]) \rightarrow a[x := y]$$

$$\mathcal{L}(\texttt{return}\ x) \rightarrow x$$

$$\mathcal{L}(\text{branch instruction} \prec R) \rightarrow \mathcal{L}(R)\ \text{(handled when reaching its } \phi)$$

$$\mathcal{L}\begin{pmatrix} \texttt{for}\ \ i = \phi(i', i''), \\ \quad r_1 = \phi(r_1', r_1''), \ldots, r_n = \phi(r_n', r_n'') \\ \quad i < l \\ \{\, E \,\} \prec R \end{pmatrix} \rightarrow \mathcal{L}\begin{pmatrix} \texttt{let}\, f = \lambda\, r_1\, r_2 \ldots i \,.\, \mathcal{L}(E \prec \langle r_1'', \ldots, r_n''\rangle)\,\texttt{in} \\ \texttt{let}\, r = \texttt{fold}\langle r_1', \ldots, r_n'\rangle\ f\ \texttt{Range}(i', l)\,\texttt{in} \\ \texttt{let}\, r_1, \ldots, r_n = r\,\texttt{in}\,\mathcal{L}(R) \end{pmatrix}$$

$$\mathcal{L}\begin{pmatrix} x = \phi(x_0, x_1) & \\ \text{generated by the } \texttt{if} & \prec R \\ \text{with branch condition } C & \end{pmatrix} \rightarrow \texttt{let}\, x = \text{if } C \text{ then } x_0 \text{ else } x_1\ \texttt{in}\,\mathcal{L}(R)$$

$$\mathcal{L}(\ldots) \rightarrow \ldots$$

Figure 6. Array SSA to Lambda Calculus with `fold`. $\mathcal{L}$ is the translation function, and $\prec$ reflects a topological ordering of statements in the CFG.

rule. One transition in our rewrite system is comprised of one application of an exploration rule followed by a complete reduction using the set of refinement rules. The set of refinement rules can be seen as a separate confluent rewrite system.

The set of transformation rules is not complete, *i.e.*, they do not generate all possible semantically equivalent programs. The rules are intended to be sound, *i.e.*, they are intended to preserve the semantics of the original program, but we have not formally proven this. More formal characterizations of soundness and completeness are planned for future work.

Optimized exploration. MOLD's rewrite system implements optimization mechanisms which can be applied according to a number of policies, guided by estimates of code quality. The system is not confluent nor terminating – so, the rewrite engine explores the space of possible rewrites guided by a heuristic driven by a cost function. The optimization problem reduces to searching through this system for a good solution. The number of states is kept in check by having a single state represent all alpha-equivalent programs that have the same beta-reduced form.

MOLD searches through the state space guided by a cost approximation function over program variants. The cost function approximates the runtime performance of the code on a particular platform. Thus, MOLD allows optimization for different platforms by adopting appropriate cost functions.

Figure 7 shows the cost estimation function for generating MapReduce programs. The estimated cost is computed recursively over a given term. The cost of function composition/application and tuples is the sum of the cost of their subexpressions. The cost for collection accesses has an extra weight ($C^{collection}_{get}$ and $C^{collection}_{set}$) to encourage access localization. `map` and `fold` operators have an initial cost meant to model the start of a distributed operation ($C^{\texttt{map}}_{init}$, $C^{\texttt{fold}}_{init}$, and $C^{\texttt{groupBy}}_{init}$), and have their operation cost multiplied by a constant ($C^{\texttt{map}}_{op}$, $C^{\texttt{fold}}_{op}$, and $C^{\texttt{groupBy}}_{op}$) representing an approximation for the size of the array. Binary operations are curried, and their function has a constant cost. All other functions have a predefined, constant, cost.

$$\mathcal{C}(F \circ G) = \mathcal{C}(F) + \mathcal{C}(G)$$
$$\mathcal{C}(F(G)) = \mathcal{C}(F) + \mathcal{C}(G)$$
$$\mathcal{C}(\langle F, G, ...\rangle) = \mathcal{C}(F) + \mathcal{C}(G) + ...$$
$$\mathcal{C}(A[I]) = C^{collection}_{get} + \mathcal{C}(A) + \mathcal{C}(I)$$
$$\mathcal{C}(A[K := V]) = C^{collection}_{set} + \mathcal{C}(A) + \mathcal{C}(K) + \mathcal{C}(V)$$
$$\mathcal{C}(\texttt{map}\, F) = C^{\texttt{map}}_{init} + C^{\texttt{map}}_{op} * \mathcal{C}(F)$$
$$\mathcal{C}(\texttt{fold}\, I\ F) = \mathcal{C}(I) + C^{\texttt{fold}}_{init} + C^{\texttt{fold}}_{op} * \mathcal{C}(F)$$
$$\mathcal{C}(\texttt{groupBy}\, F) = C^{\texttt{groupBy}}_{init} + C^{\texttt{groupBy}}_{op} * \mathcal{C}(F)$$

Figure 7. Cost estimation function

A unique set of constants is used for generating all programs in our evaluation, i.e., the constants are not program-specific. We determined good values by manually running experiments and refining the constants. This process could be automated to find a more precise and possibly platform-specific set of constants. Furthermore, our rough cost estimation function could be made more precise by applying techniques such as those proposed by Klonatos et al.[16], but the current approach has proved sufficient for optimizing most programs in our evaluation suite.

5. Optimization rules

In this section we present the main rules of MOLD's rewrite system. We show the rules in a simplified form. The actual

E	letters in uppercase are pattern variables
x	letters in lowercase are program variables or patterns matching program variables
$E_1 \subset E_2$	E_1 is a subexpression of E_2
$\mathrm{free}(E)$	is the set of free variables in E
$x \in E$	is shorthand for $x \in \mathrm{free}(E)$
$E[E_1/E_0]$	substitute E_1 for E_0 in E
K	is only used for denoting pattern matches on the parameters binding to the keys of the operator domain

Figure 8. Legend for following figures

rules have additional complexity for updating types, for handing idiosyncrasies of our source and target languages, for piggy-backing arity information useful for code generation, and for optimizing the exploration. Furthermore, the actual rule set has additional variants and guards for correctly handling non-pure functions like `random`.

Figure 8 summarizes the notation and functions we use in the following figures.

5.1 Extract `map` from `fold`

The transformation for revealing parallelism which is most commonly applied is the "extract map from fold" rule in Figure 9. It transforms a fold by identifying *independent* computations in its combining function f, i.e., operations that do not depend on results from other f invocations during the fold. These independent computations are extracted into a (parallelizable) map operation. For example, $\texttt{fold}\,0\,\lambda r\ k\ v\,.\,r + (\texttt{f}\ k\ v)$ is transformed to $(\texttt{fold}\,0\,\lambda r k\ v_f\,.\,r + v_f) \circ (\texttt{map}\,\lambda k\ v\,.\,\texttt{f}\ k\ v)$, as $(\texttt{f}\ k\ v)$ is independent (we shall explain `map` shortly). After the transformation, the purely-functional `map` can be easily parallelized, while the `fold` with the commutative $+$ operation can also be executed very efficiently.

The signatures for our data structures and mapping operators relevant to this transformation are shown in Figure 5. Data structures are either a bag of values of type A, or *indexed collections* (e.g., arrays or maps) with key type K and value type V. We often view an indexed collection as a list of key-value pairs. The first `map` version takes a collection of elements of type A into a collection of elements of type B, as is standard. The second `map` version is similar but only applies to indexed collections; it generates a new indexed collection with the same keys as the original and the mapped values. We assume that a mapping function $A \to B$ is implicitly lifted to $\langle K, A\rangle \to B$ if necessary.

The "extract `map` from `fold`" rule, shown in Figure 9, matches on any `fold` taking $\langle r_0^0, \ldots, r_n^0\rangle$ as the initial value and a function combining each tuple of keys K and values V of a collection to a tuple $\langle r_0, \ldots, r_n\rangle$. The `fold` operation E is split into the composition of functions $(\lambda\langle v_0^f, \ldots, v_m^f\rangle\,.\,F) \circ G$, such that G is the most expensive computation (according to the cost function $\mathcal{C}$; see Section 4) that is *independent* of other "iterations" of the `fold`'s execution. If we see the `fold` as a loop, G does not have any loop carried-dependencies.

How do we reason that G is independent? For a functional `fold` operation, a dependence on other fold "iterations" must be manifest as an access of an *accumulator* parameter r_i, i.e., a parameter holding the "result thus far" of the fold. Hence, if G makes no reference to any parameter r_i, it is trivially independent. Unfortunately, this simple level of reasoning is insufficient for providing independence for important cases like the following:

$$\texttt{fold}\,r^0\,\lambda\,r\,k\,.\,r[k := f(r[k])]$$

This `fold` updates each entry in a collection to a function of its previous value. We would like to extract the computation $f(r[k])$ into a parallel map operation, but it accesses accumulator parameter r and hence is not trivially independent.

To handle cases like the above, we use this more sophisticated independence check for G:

$$\nexists i \in [0..n]\,.\,r_i \in G \wedge r_i \in E[r^0_\_/r_\_]$$

As shown in Figure 9, the $E[r^0_\_/r_\_]$ expression substitutes an access to the initial collection $r_i^0[k]$ for $r_i[k]$ in E, for all possible r_i and k. (We shall discuss the reason for this particular substitution shortly.) So, in essence, the formula ensures that for any $r_i \in G$, *all* appearances of r_i in the enclosing expression E are of the form $r_i[k]$, i.e., they are accesses to a *current* key k. (Any non-conforming access like $r_i[k+1]$ will not be removed by the $r^0_\_/r_\_$ substitution.) Checking that all collection accesses in E use a current key ensures that G remains independent in spite of its access of the accumulator collection.

If a non-trivial (i.e., contains computation with a non-zero cost) G is found, it is pulled out into a `map` which is then composed with a `fold` executing F, the remaining computation in E. The signature of the `fold`'s operation is adjusted to account for the change: $v_0^f, \ldots, v_m^f$, the variables linking G to F, are now parameters, and any previous parameters (V) which are still needed by F are propagated (i.e., $V_{\cap\,\mathrm{free}(F)}$). As the extracted G no longer has access to the r_i parameters, we place $G[r^0_\_/r_\_]$ in the map instead, so its accesses are performed on the initial collection r^0.

The rule does not specify how E is decomposed. E is in many cases a tuple of expressions. Our current implementation selects the subexpression with the largest cost for each expression in the tuple E. It uses a recursive function computing the largest subexpression considering name binding constraints and the cost function.

(extract `map` from `fold`)

$$\frac{\texttt{fold}\langle r_0^0, \ldots, r_n^0\rangle\ \lambda\langle r_0, \ldots, r_n\rangle\ K\ V\,.\,E}{\begin{array}{c}(\texttt{fold}\langle r_0^0, \ldots, r_n^0\rangle\ \lambda\langle r_0, \ldots, r_n\rangle\ K\ \langle v_0^f, \ldots, v_m^f\rangle V_{\cap\,\mathrm{free}(F)}\,.\,F) \\ \circ\ (\texttt{map}\ \lambda\, K\ V\,.\langle G[r_\_^0/r_\_], V_{\cap\,\mathrm{free}(F)}\rangle)\end{array}}$$

$E = (\lambda\langle v_0^f, \ldots, v_m^f\rangle\,.\,F) \circ G$
F is $\arg\max \mathcal{C}(G)$ with the condition:
$\nexists i \in [0..n]\ .r_i \in G \wedge r_i \in E[r_\_^0/r_\_]$ where
$r_\_^0/r_\_ = r_i^0[k]/r_i[k]$ applied for all $i \in [1..n]\ k \in K$

(fold to group by)

$$\frac{\texttt{fold}\, r_0\ \lambda\, r\ V\,.\,r[E := B]}{(\texttt{map}\ \lambda\, k\ l\,.(\texttt{fold}\, r_0[k]\ \lambda\, g\ V\,.\,C)\ l)\ \circ\ (\texttt{groupBy}\ \lambda\, V\,.\,E)}$$

$C = B[g/r[E]]$
$r \notin C \wedge r \notin E \wedge \exists\, v \in V.v \in E$
we cannot prove E is distinct across the folding

Figure 9. Rules revealing parallelism in `fold` operators

The "extract `map` from `fold`" rule rewrites the example above that updates all collection values to:

$$(\texttt{fold}\, r^0\ \lambda\, r\langle k, v\rangle\,.\,r[k := v]) \circ (\texttt{map}\ \lambda\, k\,.\,f(r_0[k]))$$

The "extract `map` from `fold`" transformation is somewhat analogous to parallelizing a loop with no loop-carried dependencies in imperative code. A key difference is that here, we leverage our functional IR to extract and parallelize sub-computations of the `fold` without worrying about side effects; similar transformations for imperative loops would likely require greater sophistication.

5.2 `fold` to `groupBy`

While the "extract `map` from `fold`" rule exposes significant parallelism, it cannot handle cases where distinct loop iterations can update the same array / map location. MapReduce applications like `wordcount` from Section 2 often work around such issues by using a *shuffle* operation to group inputs by some key and then process each group in parallel. Here we present a "`fold` to `groupBy`" rule that enables our system to automatically introduce such *shuffle* operations where appropriate, dramatically increasing parallelism for cases like `wordcount`. We are unaware of any similar automatic transformation in previous work.

The transformation we used for grouping by word is an application of the "`fold` to `groupBy`" rule shown in Figure 9. As shown in Figure 5, `groupBy` clusters the elements of a collection of type $\texttt{M}[A]$ according to the result of the function $A \rightarrow K$. It returns a map from keys K to lists $\texttt{M}[A]$ of elements in the original collection that map to a specific key. The rule matches any `fold` with a body which is an update of a collection at an index E that we cannot prove as distinct for each execution of the `fold`'s body. (If the index is obviously distinct, MOLD applies the "extract `map` from `fold`" rule instead; see Section 5.1.)

The output code first groups the elements of the collection by the index expression ($\texttt{groupBy}\ \lambda\, V\,.\,E$), and then it folds each of the groups using the update expression B from original body of the loop. *groupBy*'s output is a Map from each distinct value of E to the corresponding subset of the input collection. The `map` operation's parameters are k, which binds to the keys of the grouped collection (i.e., evaluations of E), and l which contains a subset of the input collection. The fold starts from the k value of r_0, and folds l using the operation C, which is original expression B with accesses to index E of the old reducer replaced with with g, the new parameter corresponding only to the k-index of r.

The side condition requires that r does not appear in either the new expression C or the index expression E. Otherwise, the result of the of computation could depend on `fold`'s evaluation order, so the transformation would not be correct. To avoid grouping by an invariant expression, resulting in a single group, the side condition also requires that E is dependent on some parameter in V.

Revisiting the original example, the expression below is the program before applying the rule (with a beta reduction applied to highlight to make the match clear):

$$\texttt{fold}\, m\ \ \lambda\, m\ w\,.\,m[w := m[w] + 1]$$

The outer m matches r_0, the inner m matches r, w matches E, and `m[w] + 1` matches B. The side conditions are satisfied, so the expression is rewritten to:

$$\texttt{map}(\lambda\, k\ l\,.\,\texttt{fold}\, m[k](\lambda\, g\ w\,.\,g + 1)\ l) \circ (\texttt{groupBy}\ \lambda\, w\,.\,w)$$

5.3 Localizing accesses

MapReduce platforms often require computations to be *local*, i.e., to be free of accesses of global data structures. Our system contains rules to *localize* computations that do not meet this condition. Consider the following computation, based on an intermediate transformation of the `wordcount` example:

$$(\texttt{map}\ \lambda\, k\ v\,.\,m[k] + \texttt{size}\ v)\ \mathit{grouped}$$

(localize-map-accesses)

$$\frac{\texttt{map}\,\lambda\, KV\,.\,E}{\lambda\, a\,.((\texttt{map}\,\lambda\, K\, Vv\,.\,E[v/c[i]]) \circ(\texttt{zip}\, a\, c))} \qquad \begin{array}{l} c \in \{c \subset E \mid \\ (\exists! i \in K.c[i] \subset E) \wedge \\ (\text{free}(c) \setminus \text{free}(E) = \emptyset) \wedge \\ (\nexists v \in K \cup V\,.\,v \in c)\} \end{array}$$

(localize-group-by-accesses)

$$\frac{\texttt{groupBy}(\lambda\, k\,.\,E)\; D}{\texttt{groupBy}(\lambda\, kv\,.\,E[v/c[k]])c} \qquad \begin{array}{l} D \text{ is the domain (set of} \\ \text{keys) of the } c \text{ Map} \end{array}$$

Figure 10. Rules for localizing accesses

Here *grouped* maps each word to a list of occurrences, and the `map` is summing the size of each list with existing counts in some map m. This code cannot be executed in MapReduce because it accesses the collection m, unless it is localized.

Localization is achieved by explicitly passing global data as a parameter to the relevant operations, using the built-in `zip` operation from Figure 5. `zip` is overloaded to allow various input collection types. Its first version takes two maps with the same key type into a map from keys to pairs of values. If one of the maps is missing a value for a certain key, the zero value for the map's value type is used instead. For example: $\texttt{zip}(M(1 \to 8), M(2 \to 9)) = M(1 \to \langle 8, 0\rangle, 2 \to \langle 0, 9\rangle)$. `zip`'s second version takes a bag S and a map M and returns a map that retains only the entries from M with keys from S, e.g., $\texttt{zip}(S(3), M(3 \to 8, 1 \to 9, 2 \to 7)) = M(3 \to 8)$.

Using the `zip` operation, the `map` from the example above can be transformed to:

$$\texttt{map}\ (\lambda\, k\ v\ v_m\,.\,v_m + \texttt{size}\ v)\ \texttt{zip}(\mathit{grouped}, m)$$

The "localize `map` accesses" rule in Figure 10 achieves this transformation. In this form, the `map`'s operation only refers to data provided through its parameters, making it amenable to MapReduce execution. The "localize `groupBy` accesses" (Figure 10) and "localize `fold` accesses" (not shown) achieve the same purpose for their respective operators.

5.4 Loop optimizations and code motion

Our rewrite system has many rules that are inspired from classic loop optimizations and code motion that permits the application of more rules during rewriting. These are detailed in Appendix B.

5.5 Monoid-based transformations

Various transformation rules rely on viewing the map data structure as a *monoid*, i.e., a set with an associative binary operator and an identity element. Its identity is an empty map, while its "plus" operation (denoted by $\oplus$) is based on the "plus" of its value type parameters.

(eliminate null check)

$$\frac{\begin{array}{c}\texttt{if}\ \ a[k] == \mathit{null}\ \ \texttt{then}\ \ 0 \\ \texttt{else}\ \ a[k]\end{array}}{a[k]} \qquad \begin{array}{l} a \text{ is a monoid with } 0 \\ \text{as identity} \end{array}$$

(identify map monoid plus)

$$\frac{\begin{array}{c}\texttt{map}(\lambda\, i\, x\, y\,.\,x \oplus y) \\ (\texttt{zip}\ A\ B)\end{array}}{A \boxplus B} \qquad \begin{array}{l} A \text{ and } B \text{ are } \texttt{M}[T] \text{ monoids} \\ \oplus \text{ is the plus for } T \\ \boxplus \text{ is the plus for } \texttt{M}[T] \end{array}$$

(swap map and fold)

$$\frac{(\texttt{fold}\, r_0\, \oplus) \circ (\texttt{map}\, f)}{\lambda\, c\,.(r_0 \oplus f(\texttt{fold}\ 0_\boxplus\ \boxplus)\ c))} \qquad \begin{array}{l} \texttt{map}\ \text{ over monoid } \boxplus, 0_\boxplus \\ \forall ab. f(a \boxplus b) = f(a) \oplus f(b) \end{array}$$

(flatMap)

$$\frac{(\texttt{fold}\, r_0\, \oplus) \circ (\texttt{map}\, f)}{r_0 \oplus \text{flatMap}\, f} \qquad \texttt{fold}\text{ over the monoid } \oplus, 0_\oplus$$

Figure 11. Monoid-based rules

The sum of two maps, i.e., $a \oplus b$, is another map with the same keys and the values the sum of their values. If a value is missing in either map, it is replaced by the value type's *zero*. At the bottom of Figure 5 we give a possible implementation for the $\oplus$ based on `zip`.

Identifying computation that can be seen as operating over monoid structures allows further optimizations, since we can exploit associativity to expose more parallelism. Figure 11 shows our set of monoid-based transformation rules. The first two rules are "enabling" rules that make code more amenable to other optimizations, while the final rule is itself an optimization.

To illustrate the "eliminate `null` check" rule in Figure 11, let us revisit an intermediate expression from the motivating example:

$$\begin{array}{l}\texttt{let}\, \mathit{prev} = m[w]\ \texttt{in} \\ \quad m[w :\!= (\texttt{if}\ \mathit{prev} == \mathit{null}\ \texttt{then}\ 0\ \texttt{else}\ \mathit{prev}) + 1]\end{array}$$

Here, the conditional block can be eliminated by considering m a monoid with 0 as the identity element. Applying the rule yields:

$$\texttt{let}\, \mathit{prev} = m[w]\ \texttt{in}\ m[w :\!= \mathit{prev} + 1]$$

This transformation enables other optimizations by giving the code a more uniform structure.

In Section 2 we showed how the the inner loop of the `wordcount` code of Figure 2 is transformed to a MapReduce form. We now explain the last two rules of Figure 11 by showing how they are used to optimize the full loop nest. The inner loop of Figure 2 iterates over each line in the input.

After beta reduction, and without assuming m is initially empty as we did in Section 2, its optimized form is:

$$(\texttt{map}\,\lambda\,k\;v\,.\,m[k] + \texttt{size}\;v) \circ (\texttt{groupBy}\,id)$$

Placing this code in the context of the IR for the outer loop yields (after applying some non-monoid rules):

```
fold $m$ $\lambda\, m\ \langle i, doc\rangle$ .
  let $do\_count = (\texttt{map}\,\lambda\,k\;v\,.\,\texttt{size}\;v) \circ (\texttt{groupBy}\,id)$ in
    $m$ ++ $((\texttt{map}\,\lambda\,k\langle v_1, v_2\rangle\,.\,v_1 + v_2)$
          $(\texttt{zip}\;\;m\;(do\_count\;(\texttt{split}\;\;doc))))$
```

We can simplify this program using the "identify map monoid plus" rule in Figure 11. The m map and the value of $(do\_count\;(\texttt{split}\;\;doc))$ are both maps from strings to numbers. Integer numbers are monoids with arithmetic addition as plus, so our maps can be seen as monoids with the $\oplus$ operator defined in Figure 5. Zipping two monoids and adding up their corresponding values, as done above, is simply an implementation of the $\oplus$ operator. Thus, applying the rule rewrites the code to:

```
fold $m$ $\lambda\, m\ \langle i, doc\rangle$ .
  let $do\_count = (\texttt{map}\,\lambda\,k\;v\,.\,\texttt{size}\;v) \circ (\texttt{groupBy}\,id)$ in
    $m \oplus (do\_count\;(\texttt{split}\;\;doc))$
```

$do\_count\;(\texttt{split}\;\;doc)$ does not depend on the m parameter, so can be extracted to a `map` using the "extract `map` from `fold` rule (see Section 5.1). Furthermore, the resulting `map` is split into a composition of maps through fission. The computation reaches this form:

```
let $foldDocCount$ = fold $m$ $\lambda\, m\ \langle i, docCount\rangle$ .
  $m \oplus docCount$ in
    $foldDocCount \circ$
    $(\texttt{map}\,\lambda\,i\;groups\,.(\texttt{map}\,\lambda\,k\;v\,.\,\texttt{size}\;v)\;groups) \circ$
    $(\texttt{map}\,\lambda\,i\;split\,.\,\texttt{groupBy}\,id\;split) \circ$
    $(\texttt{map}\,\lambda\,i\;doc\,.\,\texttt{split}\;\;doc)$
```

While the above computation reveals significant parallelism, the final `fold`, which merges the word count map for each document, is inefficient: it repeats the work of grouping results by word and summing counts. Notice that instead of doing all the operations for each *doc* and merging the results at the end, the program could start by merging all the *doc*s and then computing word counts on the result. The "swap `map` with `fold`" shown in Figure 11 achieves this transformation.

"swap `map` with `fold`" rewrites a composition of a `fold` over a monoid of $\texttt{M}[B]$ using the monoid's plus ($\oplus$) with a `map` using function $f : A \to B$ into an application of f to the result of `fold`ing over `map`s input using monoid A's plus ($\boxplus$). The original value r_0 is also $\oplus$-added. Notice that the transformation eliminates the `map` operation, replacing it with a single application of f. Depending on the cost of f, this may result in significant speedups. The operation is correct as long as f distributes over the monoid pluses, i.e., $\forall\; a\; b\;.\; f(a \boxplus b) = f(a) \oplus f(b)$.

All three `map` functions in the above programs have distributive operations. Guided by our cost function, MOLD applies the "swap `map` with `fold`" rule two times and does two reductions of operations with identity. After some restructuring for readability, we reach the following program:

```
let $foldBagPlus$ = fold $0_{\texttt{Bag}}\;\lambda\; allWords\;\langle i, words\rangle$ .
    $allWords \boxplus_{\texttt{Bag}} words$ in
let $mapToSize = \texttt{map}\,\lambda\,k\;v\,.\,\texttt{size}\;v$
in $\lambda\,input\,.\,m \oplus (mapToSize \circ (\texttt{groupBy}\,id) \circ$
    $foldBagPlus\;\circ(\texttt{map}\,\lambda\,i\;doc\,.\,\texttt{split}\;\;doc))\;input$
```

$foldBagPlus$ is the counterpart of $foldDocCount$ from the previous code version but, instead of merging count maps, it now merges `Bag`s of words. A `map` followed by a folding of the `Bag` is equivalent to a Scala `flatMap` operation, as expressed by the "flatMap" rule in Fig. 11. Applying the rule bring the above program to:

```
let $mapToSize = \texttt{map}\,\lambda\,k\;v\,.\,\texttt{size}\;v$
in $\lambda\,input\,.\,m \oplus (mapToSize \circ (\texttt{groupBy}\,id) \circ$
    $(\texttt{flatMap}\,\lambda\,i\;doc\,.\,\texttt{split}\;\;doc))\;input$
```

The `groupBy` generates a `Map` from words to `Bag`s of words, which can be expensive in terms of I/O. As we are only interested in the size of the `Bag`, the program is transformed (by the "reduce by key" rule in 14 from Appendix B) to:

```
let $mapToSize = \texttt{map}\,\lambda\,k\;v\,.\,\texttt{size}\;v$
in $\lambda\,input\,.\,m \oplus (\text{reduceByKey}\;+) \circ$
    $(\texttt{map}\,\lambda\,i\;word\,.(word, 1)) \circ$
    $(\texttt{flatMap}\,\lambda\,i\;doc\,.\,\texttt{split}\;\;doc))\;input$
```

The `Bag` of words is mapped to a `Bag` of $(word, 1)$ pairs. `reduceByKey` groups the "1" values by key (*i.e.*, word) and reduces each group using $+$ operation, which is equivalent to the previous counting but does not generate a `Bag` of identical words for each key(*i.e.*, word).

6. Implementation

We present some details regarding MOLD's implementation by following through the transformation phases in Figure 1. The translation from Java to Array SSA is an extension of the SSA implementation in WALA [5] to handle arrays and collections. The translation from Array SSA to the Lambda IR is implemented in Scala. For the rewrite system we extended Kiama [27], a strategy-based term rewriting library for Scala. We added support for state exploration, name-bindings aware operations (e.g., "subexpression of"), and cost-guided exploration modulo $\alpha\beta$-conversion. MOLD renames variables where necessary to solve name conflicts, and flattens `let` expressions to improve performance by the following rule:

$$\frac{\texttt{let}\, x = (\texttt{let}\, y = E_y \,\texttt{in}\, E_x)\,\texttt{in} \ldots}{\texttt{let}\, y = E_y \,\texttt{in}\,(\texttt{let}\, x = E_x \,\texttt{in} \ldots)}$$

The flattened `let` constructs translate to cleaner Scala code with less block nesting. MOLD generates Scala code from the lambda calculus IR by syntactic pretty-printing rules like ($\mathcal{S}$ is a function translating to Scala):

$$\mathcal{S}(\lambda V \,.\, F) \rightarrow \texttt{\{}\,\mathcal{S}(V)\texttt{=>}\mathcal{S}(F)\texttt{\}}$$

The `let` expressions are transformed to value declarations:

$$\texttt{let}\, X = Y \,\texttt{in}\, Z \rightarrow \texttt{val}\ \ \mathcal{S}(X)\ \texttt{=}\ \mathcal{S}(Y)\ \texttt{;}\ \mathcal{S}(Z)$$

The Scala code is emitted using Kiama's pretty printing library [29] and Ramsey's algorithm [23] for minimal parenthesization.

The built-in data structures (Fig. 5) roughly follow Scala collection library's conventions. Code generated by our tool can be executed as traditional Scala code by simple implicit conversions [22]. The same code can be executed either sequentially or using the Scala parallel collections [4]. In a similar manner, we also provide a backend based on the Spark MapReduce framework [31]. This allows programs generated by MOLD to be executed on Hadoop YARN clusters [30].

7. Evaluation

We now present and experimental evaluation designed to answer the following research questions:

1. **Can MOLD generate *effective* MapReduce code?** We define *effective* code as satisfying three conditions: 1) it should not do redundant computation, 2) it should reveal a high level of parallelism, and 3) accesses to large data structures should be localized, to enable execution on a distributed memory MapReduce platform.
2. **Is MOLD *efficient*?** We measure how long it takes for MOLD to find an effective solution.
3. **Is the proposed approach *general*?** We show that the core rewrite rules are general and match many code scenarios. Also, we discuss how our cost optimization approach can generate effective solutions for different execution platforms.

To answer these questions, we study the results of using MOLD to translate several sequential implementations of the Phoenix benchmarks [24], a well-established MapReduce benchmark suite which provides both MapReduce and corresponding sequential implementations. The benchmark suite provides C implementations while our system expects Java code as input. We do a manual, straightforward, syntactic, translation of the C sequential implementations to Java. We transform C `struct` to simple Java data classes, functions to static methods, and arrays to Java arrays. Since Java does not have native multi-dimensional arrays, we use a separate Java class implementing two-dimensional array behavior. Also, because MOLD takes a single Java method as input, we inline methods that contain significant parts of the computation.

Furthermore, we manually implemented and tuned each of the benchmarks in Scala to provide a baseline against which we compare the performance of the MOLD-generated code variants (see Section 6). We derived three hand-written implementions: the first uses sequential Scala collections, the second uses parallel Scala collections, and the third use Spark [31]. We did not directly use third-party implementations since in some cases they do not exist, and in other cases they introduce optimizations orthogonal to the translation to MapReduce (*e.g.*, they use the Breeze [3] linear algebra library).

To gauge the quality of the code generated by MOLD we pass each program through our tool, we execute the resulting code and hand-written implementations, and we measure and evaluate the results. For each program:

- we measure MOLD's execution time and log the sequence of applied rules to reach the optimal solution,
- we check that the transformations preserved the semantics of the original program by executing both the original program and the generated one on the same input data set and asserting that the results are the same,
- we manually inspect the code to check whether the operators going over large input data sets have been parallelized, and whether data accesses have been localized,
- we execute the generated code with the three backends described at the end of Section 6, and then we execute the hand-written implementations on the same data sets.

When comparing with hand-written implementations, we execute each version of a benchmark on the same dataset five times within the same JVM instance. We report the average of the measurements after dropping the highest and lowest time values. We run the experiments on a quad-core Intel Core i7 at 2.6 GHz (3720QM) with 16 GB of RAM. MOLD's rewrite system state exploration is parallelized.

7.1 Can MOLD generate *effective* MapReduce code?

We discuss how MOLD generates MapReduce code for each of the subject programs, and how the generated code compares to hand-written implementations. Table 1 shows the number of loops and loop nests in the original programs, and gives the translation time and a summary of the transformations applied in order to reach the optimal version.

WordCount Figure 2 shows the original sequential code, with the outer loop iterating over documents, and the inner loop iterating over each word in each document and updating

Algorithm	# loop nests	# loops	Translation time (s)	$\frac{\texttt{fold}}{\texttt{fold} \circ \texttt{map}}$	$\frac{\texttt{fold}}{\texttt{groupBy}}$	Localization	Optimization & Enabling	Monoid	**Total**
WordCount	1	1	11	1	1	3	7	3	15
Histogram	1	1	233	0	3	1	11	3	18
Linear Regression	1	1	28	1	0	1	0	0	2
String Match	1	1	68	1	0	1	0	0	2
Matrix Product	1	3	40	4	0	2	14	0	20
PCA	2	5	66	10	0	1	4	0	15
KMeans	2	6	340	5	0	1	4	0	10

Table 1. Evaluation programs and applied transformations

a shared map (`counts`). One of the solutions found by MOLD is a map over all documents followed by a fold. The `map` contains a `groupBy` operation which computes a word count for each document (facilitated by the "`fold` to `groupBy`" rule). The `fold` merges the word counts across documents. While this is a good solution, merging word count maps may be expensive. Using the "swap `map` with `fold`" rule, MOLD moves the `fold` operation up the computation chain. In this way it reaches a form similar to the traditional MapReduce solution for the WordCount problem. The documents are split into words, which are then shuffled (`groupBy`) and the numbers of elements in each word bucket is counted.

The generated code exposes the maximal amount of parallelism for this case, and all accesses are localized so the code is distributable. Appendix A lists the transformation steps taken by the tool to reach the solutions discussed above.

Histogram This benchmark poses a similar challenge to WordCount. It computes the frequency with which each RGB color component occurs within a large input data set. MOLD generates a solution similar to WordCount. It first groups each color component by its values, and then maps to the size of the groups. Given a cost function which puts a higher weight on `groupBy` operations, MOLD can also generate a solution where the input data set is tiled and a `map` goes over each tile to compute its histogram, and the resulting histograms are merged by a final `fold`. This is similar to the approach taken by the Phoenix MapReduce solution.

The generated code is parallel and accesses are localized. The Phoenix implementation assumes the input is a single array encoding the RGB channels. The "`fold` vertical fission" rules split the computation by channel but cannot eliminate the direct stride accesses to the input array. To localize the accesses, the tool assumes the MapReduce implementation has a function `selectEveryWithOffset(k,o)` which takes every `k`th element of the input starting with offset `o`.

Linear Regression and String Match These two benchmarks are less challenging. MOLD brings them both to a parallelized and localized form by an application of "extract `map` from `fold`" followed by an application of "localize `map` accesses".

Matrix Product Matrix multiplication is a classic problem with many possible parallel and distributed implementations. The solution reached by MOLD under the cost function targeting MapReduce is a nesting of three `map` operators. Considering the input arrays `a` and `b`, the outer `map` goes over `a`'s rows, the middle `map` goes over `b`-transposed's rows (i.e., goes over `b`'s columns), and the inner operation `zips` `a`'s row with `b`'s column, then it `maps` with the multiplication operation, and finally it sums up the products. The generated code allows fine grained parallelism and it has good localization of accesses.

PCA and KMeans Generally, for these benchmark the generated code exposes good parallelism and access localization, but the generated code is not optimal. In PCA, there are three redundant `maps` and one redundant `fold`, out of a total of twelve operators. They are leftover due to some limitations of our code motion rules in handling complex tuple structures. In both cases the transformation leaves some additional no-op restructuring of the results at the end of the computation pipeline. Considering that the transformation is source-to-source, a programmer could identify and fix these issues, while still benefiting from having most of the algorithm well parallelized.

Backends For five of the benchmarks, the Scala generated code type-checks. For the remaining two, we had to add types where MOLD could not infer them due to type erasure in the input Java bytecode. Using the backends described in the Section 6, we were able to execute all benchmarks using the Scala collections, and we were able to execute five of the benchmarks on Spark. The remaining two benchmarks, KMeans and PCA, were not executable on Spark due to current limitations of the framework (it does not allow nested distributed data structures).

Comparison with hand-written implementations Table 2 compares the execution time of the generated code and hand-written implementations. Columns 2 and 3 show the execution time of the generated code using sequential and parallel Scala collections, respectively. Similarly, columns 4 and 5 show the hand-written implementations using sequential and parallel Scala collections, respectively. Columns 6 and 7 show the

program	Scala				Spark			
	generated		hand-written		generated		hand-written	
	sequential	parallel	sequential	parallel	1-thread	8-thread	1-thread	8-thread
WordCount	42.31	17.38	35.58	14.94	5.99	2.41	5.93	2.42
Histogram	9.65	7.17	9.98	6.60	8.84	2.42	8.65	2.59
LinearRegression	13.77	11.00	13.08	10.60	1.15	0.62	1.03	0.51
StringMatch	8.65	3.81	2.58	0.68	4.78	1.92	4.48	1.57
MatrixProduct	8.81	2.05	0.41	0.48	9.02	2.29	5.90	1.58
PCA	7.02	14.86	3.53	0.77	-	-	-	-
KMeans	11.90	40.31	0.61	0.89	-	-	-	-

Table 2. Execution results.

results of executing the generated code using Spark with either 1 or 8 hardware threads. Columns 8 and 9 show the corresponding results when executing the hand-written Spark implementations.

The generated versions using parallel Scala collections are between 23% and 80% faster than the generated sequential versions, with the exception of PCA and KMeans which exhibit a slowdown when executed in parallel. The generated code for these two benchmarks was found to contain significant redundant computation that severely impacts the performance of the parallel code.

Comparing the executions of the generated and hand-written versions that use parallel Scala collections, we found the former is 4%-16% slower for WordCount, Histogram, and LinearRegression, 4x-6x slower for StringMatch and MatrixProduct, and at least one order of magnitude slower for PCA and KMeans (for the reason explained earlier).

The 8-thread Spark-based versions of WordCount, LinearRegression, and StringMatch are faster than the Scala-based ones. We believe Spark gains this advantage by caching intermediate results and by using memory mapping for IO. The generated Spark codes for WordCount and Histogram are slightly faster (1% and 7% respectively) than the hand-written Spark versions. The remaining benchmarks are 22%-45% slower. As explained earlier, the generated code for KMeans and PCA relies on nested distributed data structures, which are not currently supported in Spark.

7.2 Is MOLD efficient?

Table 1 reports the MOLD execution time in column 4. For all but one program, the tool reaches the solution in under 4 minutes, and in some cases, MOLD is fast enough that it could run interactively, with the programmer getting the translated code within tens of seconds. The outlier is KMeans, which is a larger program compared to the others and has separated nested loops and complex control structure.

7.3 Is the proposed approach general?

We say that a rewrite rule set is general if it can be used to reach effective solutions for multiple programs, and each good solution depends on the application of multiple rules. Columns 5-9 of Table 1 show, for each program, the number of applications for each of the rule groups presented in Section 5. The "extract `map` from `fold`" rule (column 5) is required in all but one of the programs. The "`fold` to `groupBy`" rule (column 6) is used for WordCount and Histogram, the two programs with indirect accesses. Upon inspection, we found that the previous two rules parallelized the computationally expensive sections of the programs. Furthermore, the access localization rules are useful in all cases (column 7). With the exception of Linear Regression and String Match, all other programs also require the application of classic loop optimization and code motion transformations.

8. Related Work

Inspector-Executor. MOLD exposes parallelism using the `fold` to `groupBy` rewrite that introduces shuffle operations to move data to computation. This particularly benefits imperative loops that update a global data structure with irregular access patterns, as in `wordcount`.

This mechanism is akin to inspector-executor frameworks where (1) at runtime, an *inspector* curates data access patterns for a loop body and determines an ordering for retrieving data values, and (2) an *executor* fetches the values from remote memory locations in the specified order and executes the loop body. This model is used for irregular reference patterns over sparse data structures to improve spatial locality and hide memory access latency. Initial inspector-executor transformations were applied manually [8]. Numerous advances have automated the process and introduced advanced data reordering that combine static and runtime analysis (*e.g.*, [28] and [25]). In [28], the authors showed that sequences of loop transformations (*e.g.*, data reordering, iteration reordering, tiling) can be legally composed at compile time to yield better performance for indirect memory references.

MOLD differs from inspector-executor models in a number of ways. By rewriting loops into a functional style with high-level operators, we retain a level of abstraction that is not available when loops are decomposed into explicit inspector and executor loops. Further, MOLD operators may be mapped to more general parallel execution frameworks that include MapReduce, as we have shown in this paper.

MapReduce. MapReduce offers a programming model for parallel computing that is convenient and scalable, for applications that fit the paradigm. It frees the programmer from the burden of orchestrating communication in a distributed or parallel system, leaving such details to the MapReduce framework which may offer other benefits as well. For this reason, it is often the target of compilation from annotated sequential codes or domain-specific languages. In our work, we aimed to apply a source-to-source transformation to MapReduce directly from unmodified sequential (Java) code.

The prevalence of general purpose GPUs has catalyzed the interest in source-to-source transformations from sequential codes to parallel orchestration languages (*e.g.*, OpenMP, OpenCL) or GPU languages (*e.g.*, CUDA). In [21] for example, a number of these approaches are evaluated and a new skeleton-based compiler is described. A common theme in these efforts is the reliance on a programmer to identify and annotate their source code to aid the compiler in generating a suitable and correct parallel implementation. In comparison, MOLD automatically discovers if a loop is suitable for translation into a MapReduce style and applies term rewriting rules to enumerate a number of candidate implementations.

In [19], the author describes a compiler analysis for recognizing parallel reductions. This analysis relies on array dataflow analysis [13] to summarize data that is reachable and modified within a loop, and is applicable when memory aliases can be disambiguated. An important differentiator in our work is the use of `groupBy` which affords the ability to resolve data aliases via MapReduce shuffle operations.

The MOLD internal representation is derived from a program in array SSA form, extending previous observations that a program in SSA form is essentially a functional program [6, 15]. This functional representation is the basis for the transformations described in the paper to rewrite imperative loops into a MapReduce style. MOLD leverages the power of functional programming [18] and its algebraic properties [7, 20]. We use many of these properties in the optimization rules described in Section 5.

Program synthesis and refactoring. An extensive body of work concerns the use of program synthesis techniques to generate efficient code. Particularly relevant to our work is superoptimization, where program forms are enumerated and checked against supplied test cases to find a desired code sequence. This may be exhaustive as in the original superoptimizer, goal-oriented [14], or stochastic [26]. In many of these applications, the context is a peephole optimizer that reorders the instructions of a critical inner loop at ISA-level. This is also the case for component based program synthesis [12]. In contrast, our work is a source-to-source transformation that applies a much larger scale refactoring to loops from an imperative code sequence to a functional MapReduce style.

There is also work on refactoring toward parallelism or a more functional form. For example, [10] proposes a refactoring tool to parallelize Java loops, and [11] presents an automated refactoring of Java code to use the Java 8 collection operators. Both approaches transform the original program AST directly and are limited to specific access patterns.

9. Conclusion

We presented MOLD, a tool that automatically translates sequential, imperative code into code suitable for parallel MapReduce execution. MOLD first translates input code into a functional intermediate representation, preserving loop structures using `fold` operations. Then, MOLD searches the space of equivalent programs for an effective MapReduce implementation, based on a set of rewrite rules and a cost function that can be tuned for particular architectures. In contrast to previous work, MOLD can effectively handle irregular array accesses by introducing `groupBy` operations, which translate directly to MapReduce shuffle operations. Our evaluation showed that MOLD generated the desired MapReduce output for several real-world kernels, including codes like `wordcount` that are beyond previous techniques.

References

[1] Apache Hadoop. http://hadoop.apache.org. Accessed on 03/05/2014.

[2] Apache Spark. https://spark.apache.org. Accessed on 03/20/2014.

[3] Breeze. http://www.scalanlp.org. Accessed on 03/20/2014.

[4] Scala Parallel Collections. http://docs.scala-lang.org/overviews/parallel-collections/overview.html. Accessed on 03/20/2014.

[5] T. J. Watson Libraries for Analysis. http://wala.sf.net. Accessed: 2013-05-20.

[6] A. W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, Apr. 1998.

[7] R. S. Bird. Algebraic identities for program calculation. *Comput. J.*, 32(2):122–126, Apr. 1989.

[8] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–478, Sept. 1994.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. OSDI'04, 2004.

[10] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: Refactoring for loop parallelism in java. OOPSLA '09, pp. 793–794, 2009.

[11] L. Franklin, A. Gyori, J. Lahoda, and D. Dig. Lambdaficator: From imperative to functional programming through automated refactoring. ICSE '13, pp. 1287–1290, 2013.

[12] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. PLDI '11, pp. 62–73, 2011.

[13] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. Supercomputing '95, 1995.

[14] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. PLDI '02, pp. 304–314, 2002.

[15] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. IR '95, pp. 13–22, 1995.

[16] Y. Klonatos, A. Nötzli, A. Spielmann, C. Koch, and V. Kuncak. Automatic synthesis of out-of-core algorithms. SIGMOD '13, pp. 133–144, 2013.

[17] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. POPL '98, pp. 107–120, 1998.

[18] R. Lämmel. Google's MapReduce programming model — revisited. *Science of Computer Programming*, 70(1):1 – 30, 2008.

[19] S.-w. Liao. Parallelizing user-defined and implicit reductions globally on multiprocessors. ACSAC'06, pp. 189–202, 2006.

[20] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. FPCA '91, pp. 124–144, 1991.

[21] C. Nugteren and H. Corporaal. Introducing Bones: a parallelizing source-to-source compiler based on algorithmic skeletons. GPGPU-5, pp. 1–10, 2012.

[22] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. OOPSLA '10, pp. 341–360, 2010.

[23] N. Ramsey. Unparsing expressions with prefix and postfix operators. *Software: Practice and Experience*, 28(12):1327–1356, 1998.

[24] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. HPCA '07, pp. 13–24, 2007.

[25] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. SC '12, pp. 72:1–72:11, 2012.

[26] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. ASPLOS '13, pp. 305–316, 2013.

[27] A. M. Sloane. Lightweight language processing in Kiama. GTTSE III, pp. 408–425. Springer, 2011.

[28] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. PLDI '03, pp. 91–102, 2003.

[29] S. d. Swierstra and O. Chitil. Linear, bounded, functional pretty-printing. *J. Funct. Program.*, 19(1):1–16, Jan. 2009.

[30] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. SOCC '13, pp. 5:1–5:16, 2013.

[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud'10, pp. 10–10, 2010.

A. WordCount

```
for (int i = 0; i < docs.length; i++) {
  String[] split = docs[i].split(" ");
  for (int j = 0; j < split.length; j++) {
    String word = split[j];
    Integer prev = m.get(word);
    if (prev == null) prev = 0;
    m.put(word, prev + 1);
  }
}
```

⇓ Java to Lambda

```
Range(0, docs.size).fold(m)({
    case (v34, i) =>
      val split = docs(i).split(" ");
      val v10 = split.size;
      Range(0, v10).fold(v34)({
          case (v33, j) =>
            val v13 = v33(split(j));
            val prev = { if (v13 != null) v13 else 0 };
            v33.updated(split(j), prev + 1)
      })
  })
```

⇓ S:eliminate-null-check

```
Range(0, docs.size).fold(m)({
    case (v34, i) =>
      val v6 = docs(i).split(" ");
      val v4 = Range(0, v6.size).groupBy({
          case (j) => v6(j)
      });
      val v5 = v4.map({
          case (v2, v3) =>
            v3.fold(v34(v2))({
                case (v1, j) => v1 + 1
              })
      });
      v34 ++ v5
  })
```

⇓ S:reduce-monoid-identity-op

```
Range(0, docs.size).fold(m)({
    case (v34, i) =>
      val v6 = docs(i).split(" ");
      val v4 = Range(0, v6.size).groupBy({
          case (j) => v6(j)
      });
      val v5 = v4.map({
          case (v2, v3) =>
            v34(v2) + v3.size
      });
      v34 ++ v5
  })
```

⇓ E:localize-group-by-accesses

```
Range(0, docs.size).fold(m)({
    case (v34, i) =>
      val v4 = docs(i).split(" ").groupBy(ID).__2;
      val v5 = v4.map({
          case (v2, v3) =>
            v34(v2) + v3.size
      });
      v34 ++ v5
  })
```

⇓ E:localize-map-accesses

```
Range(0, docs.size).fold(m)({
    case (v34, i) =>
      val v4 = docs(i).split(" ").groupBy(ID).__2;
      val v5 = v4.zip(v34).map({
          case (v2, (v3, v7)) =>
            v7 + v3.size
      });
      v34 ++ v5
  })
```

⇓ E:localize-fold-accsses

```
docs.fold(m)({
    case (v34, (i, v8)) =>
      val v4 = v8.split(" ").groupBy(ID).__2;
      val v5 = v4.zip(v34).map({
          case (v2, (v3, v7)) =>
            v7 + v3.size
        });
      v34 ++ v5
  })
```

⇓ E:map-vertical-fission

```
docs.fold(m)({
    case (v34, (i, v8)) =>
      val v4 = v8.split(" ").groupBy(ID).__2;
      val v5 = v4.map({
          case (v2, v3) => v3.size
        }).zip(v34).map({
          case (v2, (v9, v7)) => v7 + v9
        });
      v34 ++ v5
  })
```

⇓ E:identify-map-monoid-plus

```
docs.fold(m)({
    case (v34, (i, v8)) =>
      (v8.split(" ").groupBy(ID).__2).map({
          case (v2, v3) => v3.size
        }) |+| v34
  })
```

⇓ E:pull-map-from-fold-by-subexpression-extract

```
docs.map({
    case (i, v8) =>
      (v8.split(" ").groupBy(ID).__2).map({
          case (v2, v3) => v3.size
        })
  }).fold(m)({
    case (v34, (i, v10)) =>
      v10 |+| v34
  })
```

⇓ E:map-horizontal-fission

```
docs.map({
    case (i, v8) =>
      v8.split(" ").groupBy(ID).__2
  }).map({
    case (i, v11) =>
      v11.map({
          case (v2, v3) => v3.size
        })
  }).fold(m)({
    case (v34, (i, v10)) =>
      v10 |+| v34
  })
```

⇓ E:swap-map-with-fold

```
val v11 = docs.map({
    case (i, v8) =>
      v8.split(" ").groupBy(ID).__2
  }).fold("ZERO–TOKEN")({
    case (v12, (i, v13)) =>
      v12 |+| v13
  });
v11.map({
    case (v2, v3) => v3.size
  })
```

⇓ E:map-horizontal-fission

```
val v11 = docs.map({
    case (i, v8) =>
      v8.split(" ").groupBy(ID)
  }).map({
    case (i, v14) => v14.__2
  }).fold("ZERO–TOKEN")({
    case (v12, (i, v13)) =>
      v12 |+| v13
  });
v11.map({
    case (v2, v3) =>
      v3.size
  })
```

⇓ E:map-horizontal-fission

```
val v11 = docs.map({
    case (i, v8) =>
      v8.split(" ")
  }).map({
    case (i, v15) =>
      v15.groupBy(ID)
  }).map({
    case (i, v14) => v14.__2
  }).fold(Map())({
    case (v12, (i, v13)) =>
      v12 |+| v13
  });
v11.map({
    case (v2, v3) =>
      v3.size
  })
```

⇓ E:swap-map-with-fold

```
val v14 = docs.map({
    case (i, v8) =>
      v8.split(" ")
  }).map({
    case (i, v15) =>
      v15.groupBy(ID)
  }).fold(Map())({
    case (v16, (i, v17)) =>
      v16 |+| v17
  });
(v14.__2).map({
    case (v2, v3) =>
      v3.size
  })
```

⇓ E:swap-map-with-fold

```
val v15 = docs.map({
    case (i, v8) =>
      v8.split(" ")
  }).fold(Map())({
    case (v18, (i, v19)) =>
      v18 |+| v19
  });
val v14 = v15.groupBy(ID);
(v14.__2).map({
    case (v2, v3) =>
      v3.size
  })
```

⇓ E:flatMap

```
val v15 = docs.flatMap({
    case (i, v8) =>
      v8.split(" ")
  });
val v14 = v15.groupBy(ID);
(v14.__2).map({
    case (v2, v3) =>
      v3.size
  })
```

⇓ E:reducebykey

```
docs.flatMap({
    case (i, v8) =>
      v8.split(" ")
  }).map({
    case (j, v6) =>
      (v6, 1)
  }).reduceByKey({
    case (v2, v3) =>
      v2 + v3
  })
```

B. Loop Optimizations and Code Motion

(map fusion)

$$\frac{(\texttt{map}\,\lambda\,k_2 v_2\,.\,E_2)\circ(\texttt{map}\,\lambda\,k_1 v_1\,.\,E_1)}{\texttt{map}\,\lambda\,k_1, v_1\,.\,E_2[k_1/k_2][E_1/v_2]}$$

(fold vertical fission)

$$\frac{\texttt{fold}\langle r_0^0,\dots,r_n^0\rangle\,\lambda\langle\langle r_0,\dots,r_n\rangle V\rangle\,.\langle E_0,\dots,E_n\rangle}{\lambda\,c\,.\langle(\texttt{fold}\,r_0^0\,\lambda\,r_0\,V\,.\,E_0)\;c,\dots,(\texttt{fold}\,r_n^0\,\lambda\,r_n\,V\,.\,E_n)\;c\rangle}\qquad\begin{array}{l}\text{applied }\forall k\in 0\dots n\\ \forall k'\neq k.r_{k'}\notin E_k\end{array}$$

(map vertical fission)

$$\frac{(\texttt{map}\,\lambda\,KV\,.\,E)\circ(\texttt{zip}\,C_0\dots C_n)}{(\texttt{map}\,\lambda\,KV[z/v]\,.\,E[z/F])\circ(\texttt{zip}\,C_0\dots(\texttt{map}\,\lambda\,k\;v\,.\,F\;C_k)\dots C_n)}\qquad\begin{array}{l}F,G=\operatorname*{arg\,max}\limits_{E=F\circ G\wedge cond}\;\mathcal{C}(F)\\ cond:(\exists!k.C_k\in F)\wedge v\in V\text{ goes over }C_k\wedge\\ \quad\forall v'\in V.v\neq v\Rightarrow v'\notin C_k\end{array}$$

(map horizontal fission)

$$\frac{\texttt{map}\;\lambda\,K\;V\,.\,B}{(\texttt{map}\,\lambda\,K(\mathrm{free}(F)\cap V)(\mathrm{free}(G)\setminus\mathrm{free}(B))\,.\,F)\circ(\texttt{map}\,\lambda\,KV\,.\,G)}\qquad\begin{array}{l}F,G=\operatorname*{arg\,max}\limits_{B=F\circ G}\mathcal{C}(F)\\ F\text{ is not trivial}\end{array}$$

Figure 12. Fusion-fission rules for merging and splitting `map` and `fold` operators. As loops in original imperative program often update multiple variables, the initial Array SSA to Lambda phase generates `fold` operators reducing over tuples of those variables. Even after `map` operators are revealed, the operators sometimes still involves large tuples. The vertical fission rules in the figure split the operators operating over tuples into multiple operators going over parts of the original tuples. The `fold` vertical fission rule rewrites a `fold` reducing to a tuple into tuple of `fold` operators reducing to the same tuple. The `map` vertical fission achieves the same purpose for the input domain. The `map` horizontal fission is the counterpart of the traditional loop fission, splitting a `map` with a $F\circ G$ into $(\texttt{map}\,F)\circ(\texttt{map}\,G)$. The `map` fusion rule is its inverse.

(transpose zipped)

$$\frac{(\texttt{map}\,\lambda\,K\langle v_0,\dots,v_n\rangle\,.\,E)(\texttt{zip}\,C_0\dots C_n)}{(\texttt{map}\,\lambda\,K\langle v_0,\dots,v_k',\dots,v_n\rangle\,.\,E[v_k'/v_k[J]])(\texttt{zip}\,C_0\dots(\texttt{t}(C_k))[J]\dots C_n)}\qquad\begin{array}{l}k\in 1\dots n\\ \exists J.v_k[J]\subset E\\ \mathrm{free}(J)\subset\mathrm{free}(ALL)\end{array}$$

(lower-dimension-for-update)

$$\frac{\texttt{fold}\,r^0\,\lambda\,r\;k\;V\,.\,r[\langle c,k\rangle:=E]}{\texttt{fold}(r^0[c])\,\lambda\,r'\;k\;V\,.\,r'[k:=E[r'[k]/r[\langle c,k\rangle]]}\qquad\begin{array}{l}c\text{ is loop invariant}\\ \text{all accesses to }r\text{ in }E\text{ are }r[\langle c,k\rangle]\end{array}$$

$$\frac{\texttt{fold}\,r^0\,\lambda\,r\;k\;V\,.\,r[\langle k,c\rangle:=E]}{\texttt{fold}(\texttt{t}(r^0)[c])\,\lambda\,r'\;k\;V\,.\,r'[k:=E[r'[k]/r[\langle k,c\rangle]]}\qquad\begin{array}{l}c\text{ is loop invariant}\\ \text{all accesses to }r\text{ in }E\text{ are }r[\langle k,c\rangle]\end{array}$$

Figure 13. Code motion rules for arrays and collections with multiple dimensions.

(eliminate empty fold)

$$\frac{\texttt{fold}\, r_0 (\lambda\, r K V\, .\, r[K := V]) d}{r_0 \,\texttt{++}\, d}$$

(eliminate empty map)

$$\frac{\texttt{map}(\lambda\, r V\, .\, V) d}{d}$$

(fold $\rightarrow$ size)

$$\frac{\texttt{fold}\, r_0\, \lambda\, r\, I\, .\, r + E}{\lambda\, x\, .\, E * (\text{size}\, x) + r_0} \quad \forall i \in I . i \notin E$$

(reduce by key)

$$\frac{(\texttt{map}\ \lambda\, K\ V\, .\ \text{size}\, V) \circ (\texttt{groupBy}\, \lambda\, P\, .\, E)}{(\text{reduceByKey}\, +) \circ (\texttt{map}\, \lambda\, P\, .(E, 1))}$$

(transpose of transpose)

$$\frac{(\texttt{t} \circ \texttt{t})\ a}{a}$$

(factor out store)

$$\frac{\texttt{fold}\, r'\, \lambda\, r\, i\, .\, r[E := B]}{r'[E := \texttt{fold}\, r'[E]\, \lambda\, k\, .\, B[k/r[E]]]} \quad i \notin E \wedge r \notin E$$

(apply store)

$$\frac{r[I := E_0][I := E_1]}{r[I := E_1[E0/(r[I := E_0][I])]]}$$

Figure 14. Simplifying and enabling transformations rules used by MOLD to enable the application of other rules as well as to simplify computation. They also generate code which uses ++, an operation shown in Figure 5. ++ takes two maps with the same type into a new map where the entries are the union of the maps, and any matching keys taking their value from the second map.