Leveraging Parallel Data Processing Frameworks with Verified Lifting

Maaz Bin Safeer Ahmad
Computer Science & Engineering
University of Washington
maazsaf@cs.washington.edu

Alvin Cheung
Computer Science & Engineering
University of Washington
akcheung@cs.washington.edu

http://casper.uwplse.org

Many parallel data frameworks have been proposed in recent years that let sequential programs access parallel processing. To capitalize on the benefits of such frameworks, existing code must often be rewritten to the domain-specific languages that each framework supports. This rewriting—tedious and error-prone—also requires developers to choose the framework that best optimizes performance given a specific workload.

This paper describes CASPER, a novel compiler that automatically retargets sequential Java code for execution on Hadoop, a parallel data processing framework that implements the MapReduce paradigm. Given a sequential code fragment, CASPER uses *verified lifting* to infer a high-level summary expressed in our program specification language that is then compiled for execution on Hadoop. We demonstrate that CASPER automatically translates Java benchmarks into Hadoop. The translated results execute on average 3.3× faster than the sequential implementations and scale better, as well, to larger datasets.

1 Introduction

As computing becomes increasingly ubiquitous, storage cheaper, and data collection tools more sophisticated, more data is being collected today than ever before. Data-driven advances are increasingly prevalent in various scientific domains. As such, effectively analyzing and processing huge datasets poses a grand computational challenge.

Many parallel data processing frameworks have been developed to handle very large datasets [2, 5, 6, 9, 12], and new ones continue to be frequently released [1, 12, 23]. Most parallel data processing frameworks come with domain-specific optimizations that are exposed either via library APIs [1, 2, 5, 6, 9, 23] or high-level, domain-specific languages (DSLs) for users to express their computations [12, 16]. Computations expressible using such API calls or DSLs are more efficient thanks to the frameworks' domain-specific optimizations [3, 16, 20, 22].

However, the many issues with this approach often make domain-specific frameworks inaccessible to non-experts such as researchers studying physical or social sciences. First, domain-specific optimizations for different workloads require an expert to decide up front the most appropriate framework for a given piece of code. Second, end users must often learn new APIs or DSLs [1, 2, 5, 6, 9, 23] and rewrite existing code to leverage the benefits provided by these frameworks. Doing so requires not only significant time and resource but also risks introducing new bugs into the application. Moreover, even users willing to rewrite their applications must first understand the intent of the code which might have been written by others. And manually written, low-level optimizations in the code often obscure high-level intent. Finally, even after learning new APIs and rewriting code, newly emerging frameworks often turn freshly rewritten code into legacy applications. Users must then repeat this process to keep pace with

Dimitrova, Piskac (Eds.): Fifth Workshop on
Synthesis (SYNT 2016)
EPTCS 229, 2016, pp. 67–83, doi:10.4204/EPTCS.229.7

new advances, requiring significant time investments that could be better spent in advancing scientific discovery.

One way to improve the accessibility of these parallel data processing frameworks involves building compilers that automatically convert applications written in common general-purpose languages (such as Java or Python) to high-performance parallel processing frameworks, such as Hadoop or Spark. Such compilers let users write their applications in familiar general-purpose languages and let the compiler retarget portions of their code to high-performance DSLs [7, 11, 15]. The applications can then leverage the performance of these specialized frameworks without the overhead of learning how to program individual DSLs. But such compilers don't always exist, and building one can prove highly complex.

This paper demonstrates the application of *verified lifting* to automatically convert sequential Java code fragments to MapReduce. As input, verified lifting takes program fragments written in a general-purpose language and uses program synthesis to *automatically find* provably correct code summaries. These summaries—expressed in our program specification language—encode the semantics of the input code fragment. The found summaries are then used to translate the original input code to the target high-performance DSL.

The concept of verified lifting has been previously applied to database applications [7] and stencil computations [11]. This paper applies verified lifting to the conversion of sequential data processing Java code to leverage the parallel data processing frameworks Apache Hadoop. The problem statement remains familiar and was first proposed in the MOLD compiler [15], which translates sequential Java code for execution on Apache Spark. MOLD uses pre-defined rewrite rules to search the space of equivalent Apache Spark implementations. It scans the input code for patterns that trigger such rewrite rules, an approach fraught with many limitations. For instance, it requires the a priori definition of complicated rewrite rules, which can be extremely brittle to code pattern changes. In comparison, our approach analyzes program *semantics* rather than program *syntax*, making it robust to code pattern changes. We also do not rely on pre-defined translation rules and can thus discover new solutions and optimizations that the user never knew existed.

We implemented our approach described above in a compiler called CASPER. By converting sequential code fragments to Hadoop, CASPER parallelizes computation at crucial program points where input data collections are being processed. We used CASPER to convert five benchmark programs with encouraging results. This paper thus makes the following contributions:

- We describe the use of *verified lifting* to retarget sequential Java applications to Hadoop by converting code fragments within the application to Hadoop MapReduce tasks.
- We design a *new program specification language* to express the intent of Java code fragments using the MapReduce paradigm.
- We employ *static program analysis techniques* to intelligently restrict the search space of all possible summaries that can be expressed in our specification language and use *inductive synthesis* to find provably correct summaries for each input code fragment.
- We present encouraging preliminary results from using CASPER to identify and optimize code fragments written in sequential Java. To show the potential of our approach, we evaluate our system on five MapReduce benchmarks used in prior work [17] to demonstrate its capabilities and limitations.

In the following, we describe CASPER's design and illustrate its use to convert sequential Java programs into Hadoop tasks in §2. In §3 we explain verified lifting and describe how we implemented each of its steps in CASPER. §4 evaluates how CASPER performs using varied benchmarks and shares our preliminary results. §5 describes related work, and we conclude in §6.

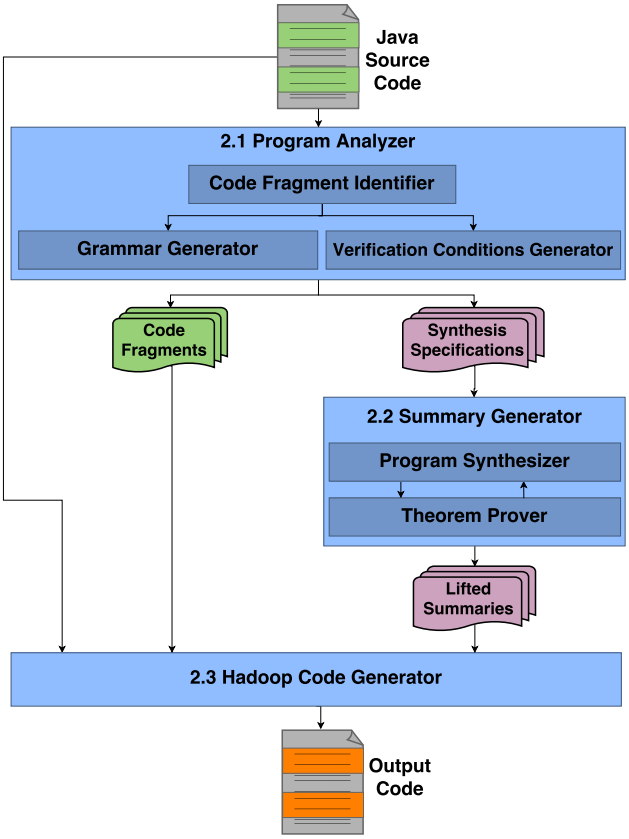


Figure 1: CASPER system architecture diagram. Sequential code fragments (highlighted green) in the input source file are translated to equivalent Hadoop tasks (highlighted orange).

2 System Overview

This section describes the architecture of the CASPER compiler. CASPER automatically identifies and converts sequential Java source code fragments into semantically equivalent MapReduce tasks implemented using Hadoop. It generates new, optimized version of the input source code where original code fragments are replaced by invocations to the generated MapReduce tasks. Figure 1 shows CASPER's different components and how they interact in the compilation pipeline.

Before explaining each component in detail, we should generally note that by statically analyzing Java input source code, CASPER extracts code fragments that can potentially be translated. It then generates a high-level summary of each extracted code fragment. Expressed in our high level program specification language (see §3.1), the summary is inferred by a program synthesizer. To quickly traverse the large space of possible summaries, CASPER bounds the search space considered by the synthesizer and uses a bounded model-checking procedure to locate any candidate candidate summaries for a given code fragment.[1] The Hadoop code generator module uses the verified summary to produce code for Hadoop MapReduce tasks. Lastly, the code generator module prepares a new version of the input source

[1]While not yet implemented in the current CASPER prototype, any candidate summary that passes the bounded model checking phase will be forwarded to a theorem prover, which verifies that the synthesizer-generated summary is semantically equivalent to the original code.

```
1  int[][] histogram(int[] data) {
2    int[] hR = new int[256];
3    int[] hG = new int[256];
4    int[] hB = new int[256];
5    for (int i = 0; i < data.length; i += 3){
6      int r = data[i];
7      int g = data[i + 1];
8      int b = data[i + 2];
9      hR[r]++;
10     hG[g]++;
11     hB[b]++;
12   }
13   int[][] result = new int[3][];
14   result[0] = hR;
15   result[1] = hG;
16   result[2] = hB;
17   return result;
18 }
```

(a) Input source code

```
Map kvPairs = HistogramHadoop.execute();
hR = kvPairs.get(0);
hG = kvPairs.get(1);
hB = kvPairs.get(2);
```

(b) CASPER wrapper to replace the loop in (a)

```
1  public class HistogramHadoop{
2    class HistogramMapper extends Mapper {
3      void map(int key, int[] value){
4        for (int i=0; i<value.length; i+=1) {
5          if(i%3==0) emit((0, value[i]), 1);
6          if(i%3==1) emit((1, value[i]), 1);
7          if(i%3==2) emit((2, value[i]), 1);
8    }}}
9    class HistogramReducer extends Reducer {
10     void reduce(Tuple key, int[] values) {
11       int value = 0;
12       for (int val:values) {value=value+val;}
13       emit(key, value);
14   }}
15   static Map execute() {
16     Job = Job.getInstance();
17     job.setMapper(HistogramMapper);
18     job.setReducer(HistogramReducer);
19     return job.execute();
20   }
21 }
```

(c) CASPER-generated Hadoop task

Figure 2: CASPER translation of the 3D Histogram benchmark.

code by replacing the original code fragments with invocations to the translated Hadoop tasks.

Throughout the remainder of this paper, we use, as a running example, a benchmark from the Phoenix suite of benchmarks [17] that generates 3D histograms from image data stored in a file. Figure 2 shows CASPER's translation of this 3D Histogram benchmark. In Figure 2a, the original program sequentially iterates over an array of integers representing the intensity values of colors red, green, and blue for each pixel. It then counts the number of times each value occurs for each color from Lines 9 to 11. The parallel program CASPER generates, on the other hand, emits a key-value pair with the tuple (color, intensity) as key and 1 as value (lines 5 to 7 in Figure 2c). The generated pairs are then grouped by key and the frequency of each key is calculated by adding all the 1's in the reducer phase (line 12). Figure 2b shows the final code generated by CASPER that replaces the original loop, where it invokes the Hadoop task shown in Figure 2c and uses the output from the Hadoop task to update program state accordingly. In our evaluation, the translated version of the benchmark performed over 1.5× faster for a dataset of 50GB.

We now discuss the three essential modules that make up CASPER's compilation pipeline.

2.1 Program Analyzer

The program analyzer, the first component in CASPER's compilation pipeline, has two goals. First, it identifies all code fragments that are candidates for conversion. Second, it generates synthesis specifications for each identified candidate code fragment. The program analyzer operations are grouped into three sub-components: the code fragment identifier, the verification conditions generator, and the grammar generator.

In the current prototype, CASPER's code fragment identifier finds loops from the input program and extracts them as candidates for conversion. CASPER currently does not consider non-looping code fragments (such as recursive functions) as candidates for conversion. In addition, CASPER also ignores loops containing calls to external library methods that are unrecognized by the CASPER compiler. In §3.4, we formally list the criteria needed for a code fragment to be extracted as a candidate for conversion, highlighting some current limitations of CASPER's implementation.

The second program analyzer sub-component is the grammar generator, which aims to confine the space of synthesizable summaries. Without doing so, the space of all possible summaries expressible in our program specification language would be too large. The grammar generator takes as input the code fragments extracted by the code fragment identifier and statically analyzes each one to extract semantic information. It then uses the extracted information to generate a program grammar for every code fragment. The challenge here is to generate a grammar expressive enough to express the correct summary, but not so expressive as to make the problem of summary search intractable. In §3.3.2, we explain how the grammar generator leverages static program analysis to construct a grammar for each code fragment.

The third program analyzer sub-component is the verification conditions generator. This component uses Hoare logic [10] and static program analysis to generate verification conditions for each code fragment. *Verification conditions* are logical statements that describe what must be true for a given summary to be semantically equivalent to the original code fragment. We explain how CASPER uses Hoare style program verification to verify program equivalence in §3.2.

The output of the verification conditions generator is a search template for the summary, with: (i) the search space specified by the grammar generator, and (ii) the verification conditions generator producing the logical assertions that must be satisfied given a candidate summary. The summary generator uses this template to search for a valid summary of the input code fragment, as we next explore.

2.2 Summary Generator

Using the program analyzer's specifications, the summary generator traverses the search space to find a summary that satisfies the verification conditions. It consists of two modules: the program synthesizer and the theorem prover. The synthesizer takes the search space description and verification conditions previously generated and searches for a code summary that satisfies the verification conditions. To make the search problem tractable, it uses a bounded model checking procedure: the synthesizer checks for correctness only over a small sub-domain. When a promising candidate for the summary is found, viz., one that satisfies the verification conditions in the sub-domain, it is passed onto the formal theorem prover [2], which checks it for correctness over the entire domain of inputs. Candidate solutions that fail the formal verification step are eliminated from the search space, and the search restarts for a new candidate solution. Using this two-step verification process helps CASPER quickly discard bad candidates. The more computationally expensive process of formal verification is reserved only for promising candidate solutions. As output, the summary generator emits a verifiably correct summary for each code fragment that can then be translated to Hadoop. Note that the summary generator may not always find a solution that can be proven correct: some fragments are impossible to translate to MapReduce while others might have complex solutions that CASPER currently can not generate. In such cases, CASPER gives up on translating the code fragment.

[2] See footnote 1.

2.3 Hadoop Code Generator

The summaries synthesized by the summary generator are expressed in our high-level program specification language. Generating Hadoop implementations from these high level program specifications is straightforward and is achieved in CASPER through syntax-directed translation rules. The code generator module also outputs the code required to embed these Hadoop tasks into the original program. Essentially, CASPER generates a new (source) version of the input code, where each code fragment that was successfully translated is replaced by code that first invokes the corresponding Hadoop task and then uses the output generated by the Hadoop task to update the state of the program. Figure 2b shows such generated wrapper code for the 3D Histogram example. We present more details about CASPER's code generation module in §3.5.

3 Converting Code Fragments

This section explains how CASPER uses verified lifting to convert sequential Java code fragments to MapReduce tasks. We review the concept of verified lifting in §3.1 and describe the program specification language CASPER uses to express program summaries. In §3.2, we explain how CASPER verifies that the identified summaries preserve program semantics of the original code fragment. §3.3 discusses the search process CASPER uses to find program summaries, while §3.4 explains how CASPER selects suitable code fragments for translation. Finally, §3.5 explains code generation after the program summary has been inferred.

3.1 Verified Lifting

Verified lifting [7,11] is a general technique that infers the semantics of code written in a general-purpose language by "lifting" it to summaries expressed using a high-level language. CASPER specifies code fragment summaries in our program specification language in the form of postconditions that describe the effects of the code fragment on its *output variables*, i.e., variables that are modified within the code fragment. The goals of our program specification language are:

- To generate summaries that CASPER can translate to the target platform DSL. This excludes valid summaries that cannot be translated. Therefore, the language should omit constructs that cannot be translated easily to the target.
- To generate non-trivial summaries that exhibit parallel data processing. Obviously, this excludes summaries that execute the computation sequentially. §4.2.2 discusses the sources of parallelism in MapReduce and how CASPER generates solutions that exploits them.

With these goals in mind, CASPER's inferred summaries must be of the form:

$$\forall\, v \in outputVariables\ .\ v = reduce(map(data, f_m), f_r)[id_v] \tag{1}$$

where $data$ is the iterable input data collection. The map function iterates over the $data$ while calling the f_m function on each element. f_m takes as input an element from $data$ and generates potentially multiple key-value pairs. map then collects and returns key-value pairs generated by invocations of f_m. The $reduce$ function takes these key-value pairs, groups them by key, and calls f_r for each key and all values that correspond to that key. Function f_r aggregates all values for the given key and emits a single key-value pair. Like map, $reduce$ collects all aggregated key-value pairs and returns an associative array

that maps each variable's ID to its final value. The variable ID is a unique identifier that CASPER assigns to every output variable. CASPER requires that summaries (i.e., postconditions) be of the form described in Eqn. (1) for easy translation to Hadoop tasks.

In the preceding discussion, f_m and f_r remain unspecified. Verified lifting seeks a definition of f_m and f_r that makes a valid inferred summary, viz., one that preserves the semantics of the input code fragment. To do this in CASPER, the synthesizer generates the implementation of these two functions (see §3.3) using the verification conditions computed by the program analyzer for each code fragment (see §3.4).

3.2 Verifying Equivalence

The summaries CASPER infers must be semantically equivalent to the input code fragment. CASPER establishes the validity of the inferred postconditions using Hoare-style verification conditions [10], which represent the weakest preconditions of a code fragment that must be true to establish the postcondition of the same code fragment under all possible executions. Generating verification conditions for simple assignment statements and conditionals is straightforward. For example, consider the imperative program statement `x := y + 3`. To show that the candidate postcondition `x > 10` is a valid postcondition, we must prove that `y + 3 > 10` is true before the statement is executes. In this case, `y + 3 > 10` is called the *verification condition* for this postcondition. Computing verification condition is easy for simple statements. For a loop, however, computing verification conditions becomes more difficult since a loop invariant is needed. The *loop invariant* is a hypothesis that asserts that the postcondition is true regardless of how many times the loop iterates. Hoare logic states that the following three statements must hold for the loop invariant (and postcondition) to be valid:

1. $\forall \sigma.\ preCondition(\sigma) \rightarrow loopInvariant(\sigma)$
2. $\forall \sigma.\ loopInvariant(\sigma) \wedge loopCondition(\sigma) \rightarrow loopInvariant(body(\sigma))$
3. $\forall \sigma.\ loopInvariant(\sigma) \wedge \neg loopCondition(\sigma) \rightarrow postCondition(\sigma)$

Statement 1 asserts that the loop invariant must be true when the precondition is true for all program states (σ), i.e., the loop invariant must be true before entering the loop. Statement 2 asserts that for all possible program states σ—assuming that the loop invariant is true and that the loop continues—the loop invariant remains true after one more execution of the loop body; (here, $body(\sigma)$ returns a new program state after executing the loop body at σ). Statement 3 asserts that if the loop invariant is true and if loop terminates, then the postcondition must be true for all possible program states.

Two challenges affect the identification of postconditions (and hence summaries) for code fragments that involve loops. First, *both* the loop invariants and postcondition must be synthesized. Unlike prior work on searching for invariants [8, 21], however, CASPER needs to find loop invariants that are only logically strong enough to establish the soundness of the postcondition, i.e., those that satisfy statement 3. This is made easier thanks to the specific form of the postcondition that CASPER looks for. In addition, establishing the validity of the found invariants and postconditions requires checking *all* possible program states, complicating the synthesis problem. We discuss how CASPER makes the search problem manageable in §3.3.3.

3.3 Searching for summaries

CASPER seeks to infer a summary for each code fragment, where each summary is a postcondition of the form explained in §3.1. This section describes how CASPER uses synthesis to search for postconditions *and* the loop invariants they require to prove the postconditions correct.

$$
\begin{aligned}
&preCondition(hR, hG, hB, i) \equiv \\
&\quad hR = [0..0] \ \wedge \ hG = [0..0] \ \wedge \ hB = [0..0] \ \wedge \ i = 0 \\
&postCondition(data, hR, hG, hB) \equiv \\
&\quad \forall \ \ 0 \leq j < hR.length. \ hR[j] = reduce(map(data, f_m), f_r)[(0, j)] \ \wedge \\
&\quad \forall \ \ 0 \leq j < hG.length. \ hG[j] = reduce(map(data, f_m), f_r)[(1, j)] \ \wedge \\
&\quad \forall \ \ 0 \leq j < hB.length. \ hB[j] = reduce(map(data, f_m), f_r)[(2, j)] \\
&loopInvariant(data, hR, hG, hB, i) \equiv \\
&\quad LoopCounterExp \ \wedge \\
&\quad \forall \ \ 0 \leq j < hR.length. \ hR[j] = reduce(map(data[0:i], f_m), f_r)[(0, j)] \ \wedge \\
&\quad \forall \ \ 0 \leq j < hG.length. \ hG[j] = reduce(map(data[0:i], f_m), f_r)[(1, j)] \ \wedge \\
&\quad \forall \ \ 0 \leq j < hB.length. \ hB[j] = reduce(map(data[0:i], f_m), f_r)[(2, j)]
\end{aligned}
$$

Figure 3: Definitions of precondition, postcondition and loop invariant for the 3D Histogram example.

3.3.1 Generating Verification Conditions

In §3.2, we explained the three verification conditions that must be satisfied by the synthesized summary. These verification conditions involve a precondition, postcondition, and loop invariant for the code fragment. Preconditions are generated by extracting, through static program analysis, the program state (values of input and output variables) just before the loop starts executing. When the value of a variable before the loop starts cannot be determined, CASPER generates a new variable to represent the initial value. The loop invariant has a form similar to the postcondition (see §3.1); unlike the postcondition, however, which calls *map* and *reduce* on the entire data collection, the loop invariant calls *map* and *reduce* only on the subset of the collection that has so far been traversed by the loop. Also, the loop invariant includes an expression that describes the behavior of the loop counters.

Figure 3 shows the precondition, postcondition and loop invariant generated for the 3D Histogram benchmark. The postcondition and loop invariant functions describe the behavior that must be true for the bodies of f_m and f_r to be correct. For example, the postcondition states that for each index j of hR, the value of $hR[j]$ must equal to the output of map and reduce functions for key $(0, j)$.

3.3.2 Specifying Search Space

This section describes how CASPER generates the grammar that the synthesizer uses to construct bodies of f_m and f_r. By dynamically generating a grammar for each code fragment, CASPER restricts the space of summaries through which the synthesizer must search.

Recall that the function f_m takes as parameters the input data collection and an index into the collection and returns a set of key-value pairs. CASPER constructs the body of f_m using emit statements and conditionals. The current CASPER prototype does not generate implementations of f_m that involve loops. Based on our experiments, we found that using the same number of emit statements as output variables in the code fragment works well as a starting point. The number of emit statements can then be increased if a solution cannot be found. In general, however, CASPER takes a conservative approach to avoid implementations with redundant emit statements since they generate unnecessary shuffle data, consequently hurting performance. Each emit statement produces a key-value pair; the key and value can be any expression generated by one of our expression grammars or tuples of such expressions.

The f_r function reduces all values emitted by *map* for a given key into a single value. The body of

$$
\begin{array}{rcl}
\mathsf{f_m} & ::= & \{\mathsf{EmitMap};\ \mathsf{EmitMap};\ \mathsf{EmitMap};\} \\
\mathsf{EmitMap} & ::= & \mathsf{emit(Exp,\ Exp)} \mid \mathsf{if(BoolExp)\{\ emit(Exp,\ Exp)\ \}} \\
\mathsf{Exp} & ::= & \mathsf{IntExp} \mid \mathsf{BoolExp} \mid \mathsf{(Exp, Exp)} \\
\mathsf{IntExp} & ::= & \mathsf{IntTerm} \mid \mathsf{data[IntExp]} \mid \mathsf{IntExp} + \mathsf{IntExp} \mid \mathsf{IntExp} \,\%\, \mathsf{IntExp} \\
\mathsf{IntTerm} & ::= & \mathsf{intLiteral} \mid \mathsf{loopCounter} \\
\mathsf{BoolExp} & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{IntExp} == \mathsf{IntExp} \mid \mathsf{BoolExp} \ \wedge\ \mathsf{BoolExp} \\
 & \mid & \mathsf{BoolExp} \ \vee\ \mathsf{BoolExp} \\
\mathsf{f_r} & ::= & \{\mathsf{value} = \mathsf{IntLiteral};\ \mathsf{for(v\ in\ values)\{\ value} = \mathsf{FoldExp}\ \}\ \mathsf{emit(key, value);}\} \\
\mathsf{FoldExp} & ::= & \mathsf{FoldTerm} \mid \mathsf{FoldExp} + \mathsf{FoldExp} \\
\mathsf{FoldTerm} & ::= & \mathsf{intLiteral} \mid \mathsf{value} \mid \mathsf{v} \\
\mathsf{LoopCounterExp} & ::= & \mathsf{LoopTerm} <= \mathsf{LoopTerm} <= \mathsf{LoopTerm} \\
\mathsf{LoopTerm} & ::= & \mathsf{loopCounter} \mid \mathsf{intLiteral} \mid \mathsf{data.length}
\end{array}
$$

Figure 4: Grammar generated for 3D Histogram example.

f_r implements the folding operation. CASPER uses the synthesizer to generate the folding expression that reduces two values into one. It also generates an expression grammar to synthesize the folding expression.

CASPER generates expression grammars for each primitive data type found in the code fragment. Each grammar can be used to generate expressions that evaluate to a value of its type. The expressions are formulated using the operators and function calls from the original code fragment. Input variables, loop counters, and literals from the code fragment are used as terminals. For arithmetic types, CASPER lets the synthesizer generate new constants as well. Furthermore, CASPER generates an expression grammar to construct the folding expression in f_r and the loop counter expression in the loop invariant.

All expression grammars generated by CASPER are bounded to a set level of recursion which the user can specify. The recursive bound of a grammar controls the amount of times the synthesizer is allowed to expand the non-terminals while formulating an expression. If the synthesizer cannot find a solution, the expression grammars can be incrementally expanded by either introducing new operators and functions that were not found in the code fragment or increasing the recursive bound on the grammar. The order in which new constructs are added to the grammar is guided by priority values that we have encoded into CASPER.

Figure 4 shows the grammar generated for the 3D Histogram benchmark after 2 iterations of grammar expansion. It is easy to see how to the solution presented in Figure 2 can be generated from this grammar.

3.3.3 Search Procedure

Despite all the search space constraints already discussed, the space of possible summaries remains large. Therefore, to accelerate the search, CASPER splits the verification process into two parts: it first uses a bounded-checking procedure to find candidate invariants and postconditions. For candidate invariants and postconditions that pass the bounded-checking procedure, it then uses a theorem prover to establish soundness for all input program states. If the theorem prover fails (via a timeout) or returns unsat, the synthesizer continues to search for a new candidate summary in the same search space. When it finds no more candidate summaries, the synthesizer expands the grammar to increase the search space. It does

this by either adding new non-terminals, increasing the recursive bound for the grammar, or increasing the number of emits made by f_m, as discussed earlier. Configuration parameters specified by the user control this iterative expansion of the search space. Eventually, the synthesizer either finds a verifiably correct summary or halts efforts to convert the code fragment.

CASPER also decouples the synthesis procedure from formal verification and uses off-the-shelf tools for each of the two sub-problems. This methodology works well in practice to reduce the synthesis time.

3.4 Initial Code Extraction

The current CASPER prototype parses the abstract syntax tree (AST) of the input program source code to extract loops as individual fragments. CASPER then analyzes each fragment's AST to ensure it meets the following criteria:

- The code fragment contains no unsupported library function calls. To synthesize summaries, CASPER must identify input and output variables (see §3.4.1), and the lack of library source code makes this impossible unless models that describe library function semantics are encoded into the compiler. CASPER currently supports commonly used library functions, such as methods of the `java.lang.{String,Integer}` and `java.util.{ArrayList,Map}` classes.
- Each loop contains no unstructured control flow. CASPER's current implementation cannot extract necessary semantics from such loops, such as the premature terminations and loop stride.
- The code fragment contains no nested loops. CASPER does not currently process nested loops. If any is found, CASPER attempts to optimize only the innermost loop.
- The code fragment contains no assignment statements that can create an alias. Moreover, CASPER does not currently perform any alias analysis and assumes that none of the input variables in the code fragment is aliased. Thus, user defined objects cannot be assigned. Fields of these objects can be modified as long as they are a primitive type. Similarly, array indexes can be modified—if array is of an immutable type—but not the pointer to an array. Support for assigning common immutable data structures, such as `java.lang.{Integer,String}`, has been built into the compiler.

CASPER overlooks code fragments that do not satisfy these criteria. Once a loop has been marked for conversion, it is normalized to a simpler form before further analysis. The normalization breaks down large instructions into smaller, simpler ones (such as breaking down all expressions into binary ones) and converts all loop constructs into `while(true){...}` loops. All of which are standard compiler transformations.

3.4.1 Extracting Input and Output Variables

CASPER makes additional passes on the normalized AST to extract input and output variables. It examines each assignment statement inside the code fragment in isolation and extracts assignment targets as output variables. Similarly, all variables in the source of an assignment are extracted as input variables (this may also include some output variables). Local variables declared inside a loop body are considered neither input nor output variables. To determine whether a function call parameter is an input or output variable, CASPER must analyze the function's source code. For library functions, this information must be encoded into CASPER beforehand. If a constant index of an array is accessed, then a separate input variable is created for the array element. However, if any dynamic accesses are made, then the entire array is considered an input variable.

For the 3D Histogram example shown in Figure 2, arrays `hR`, `hG` and `hB` are labeled as output variables, and the `data` array is identified as an input variable. Variables `i`, `r`, `g` and `b`—all declared inside the loop body—are considered to be neither input nor output variables.

3.5 Code Generation

After CASPER finds a summary for each input code fragment, the last step is to convert each such summary into a Hadoop task. The class encapsulating the Hadoop task has an `execute` method, which takes as parameters all input variables in the code fragment. This method invokes the Hadoop task and returns an associative array that maps each variable identifier to its final value as computed by the Hadoop task. The associative array is then used to update the output variables before the remaining program is executed. Translation of f_m and f_r to concrete Hadoop syntax is done using syntax-driven translation rules. Since the postcondition is already in the MapReduce form, the rules to translate them into the concrete syntax of Hadoop are straightforward and omitted here for brevity.

Figure 2c shows the final output code for the 3D Histogram example. `HistogramHadoop` is the class generated by CASPER, and the `execute` method invokes the Hadoop runtime with the generated map and reduce classes. The resulting values—`hR`, `hG`, and `hB`—are compiled and returned by `execute` and assigned to the original program's corresponding output variables as shown in Figure 2b. The code that reconstructs the arrays from key-value pairs is not shown for brevity.

4 Evaluation

We now describe our prototype implementation of CASPER and present the results derived from applying CASPER to varied benchmarks.

4.1 Implementation

CASPER's program analysis and code generation modules are implemented by extending the open source Java compiler Polyglot [14]. For synthesis, CASPER uses an off-the-shelf synthesizer called SKETCH [18]. SKETCH uses counter-example guided inductive synthesis as its core algorithm. The program analyzer encodes the verification conditions and search space in the SKETCH language. We implemented the functions and data structures required to model the semantics of MapReduce programs in SKETCH as well. In addition, CASPER automatically models in SKETCH all program-specific user-defined data types. SKETCH performs bounded model-checking to generate a summary, which we then use to generate the Hadoop Code. We have not yet implemented CASPER's formal verification component in CASPER and therefore rely solely on bounded model-checking to verify correctness.

4.1.1 Platform for Evaluation

We used our CASPER prototype to translate sequential Java benchmarks into Hadoop tasks. We measured the performance of both the original and the generated implementations on a 10 node cluster of Amazon AWS m3.xlarge instances. Each m3.xlarge node was equipped with High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) 2.5 GHz processors, 15 GB of memory, and 80 GB of SSD storage. The cluster ran Ubuntu Linux 14.04 LTS, Hadoop 2.7.2 and Spark 1.6.1. We used HDFS for input data storage in both sequential and MapReduce implementations.

4.1.2 Benchmarks

We evaluated the performance of CASPER on the following five benchmarks. These benchmarks were taken from the Phoenix suite of benchmarks [17] and represent traditional problems that can be parallelized by rewriting using the MapReduce paradigm.

- The **Summation** benchmark sums all integer values in a list.
- The **Word Count** benchmark counts the frequency of each word in a body of text by iterating through each word in the input file.
- The **String Match** benchmark determines whether a set of two strings is contained in a body of text. It returns a Boolean value for each string as output. Like Word Count, this benchmark also iterates through each word in the input file.
- The **3D Histogram** benchmark generates a three-dimensional histogram that tallies the frequency of each RGB color component in an image (Figure 2a). The output is an array for each color component that holds the frequency of each intensity value.
- The **Linear Regression** benchmark iterates over a collection of cartesian points (x, y) and computes a number of coefficients for linear regression: namely, x, y, $x*x$, $x*y$, $y*y$.

All benchmarks read input data from a text file saved on HDFS. For the generated Hadoop solutions, class `org.apache.hadoop.mapred.FileInputFormat` is used to read and split data across multiple mappers.

4.2 Compilation Performance

This section reports the time that CASPER takes to generate Hadoop implementations and discusses the quality of these implementations.

4.2.1 Scalability

Table 1 shows the average time (over 5 runs) required to synthesize a summary for each of the five benchmarks. CASPER synthesized Hadoop implementations for all benchmarks within an hour. Simpler benchmarks, such as **Summation** and **Word Count**, were converted in under a minute and required only one iteration of grammar generation. No benchmark required more than two iterations to successfully synthesize an implementation.

Benchmark	Program Analysis	Synthesis and BMC	# of Grammar Iterations
Summation	$<$ 1s	13s	1
Word Count	$<$ 1s	44s	1
String Match	$<$ 1s	1406s	2
3D Histogram	$<$ 1s	2355s	2
Linear Regression	$<$ 1s	1801s	2

Table 1: Average time for CASPER to synthesize each benchmark.

4.2.2 Sources of Parallelism

A MapReduce program has two primary sources of parallelism. First, processing can be parallelized in the *map phase* by partitioning the input data and spawning multiple mappers to process each partition simultaneously. Second, the *reduce phase* can be executed in parallel by grouping data to separate keys and aggregating for each key simultaneously. Hadoop also supports the use of combiners. Before the shuffle phase, *combiners*—if used—aggregate data locally on every node to offer additional parallelism and decrease the amount of data that needs to be shuffled.

We now discuss the implementations CASPER generated and how each leveraged both map and reduce side parallelism.

The **Summation** benchmark produces as output a single integer variable. All data must be aggregated together and cannot be split to multiple keys. The translated solution emits a key-value pair $(0, number)$ for each number in the input dataset during the map phase. These key-value pairs are aggregated locally on each node in parallel before being sent to the reducer. Note that key 0 is the unique ID for the output variable.

The CASPER-generated implementation of the **Word Count** benchmark emits $(word, 1)$ for each word encountered. The reducer then sums the values for each key. All nodes aggregate data locally (using a combiner) to compute word counts for the assigned data partition before the reducer aggregates intermediate results. In addition, CASPER uses the words as keys. Therefore, the aggregation for different words is performed in parallel.

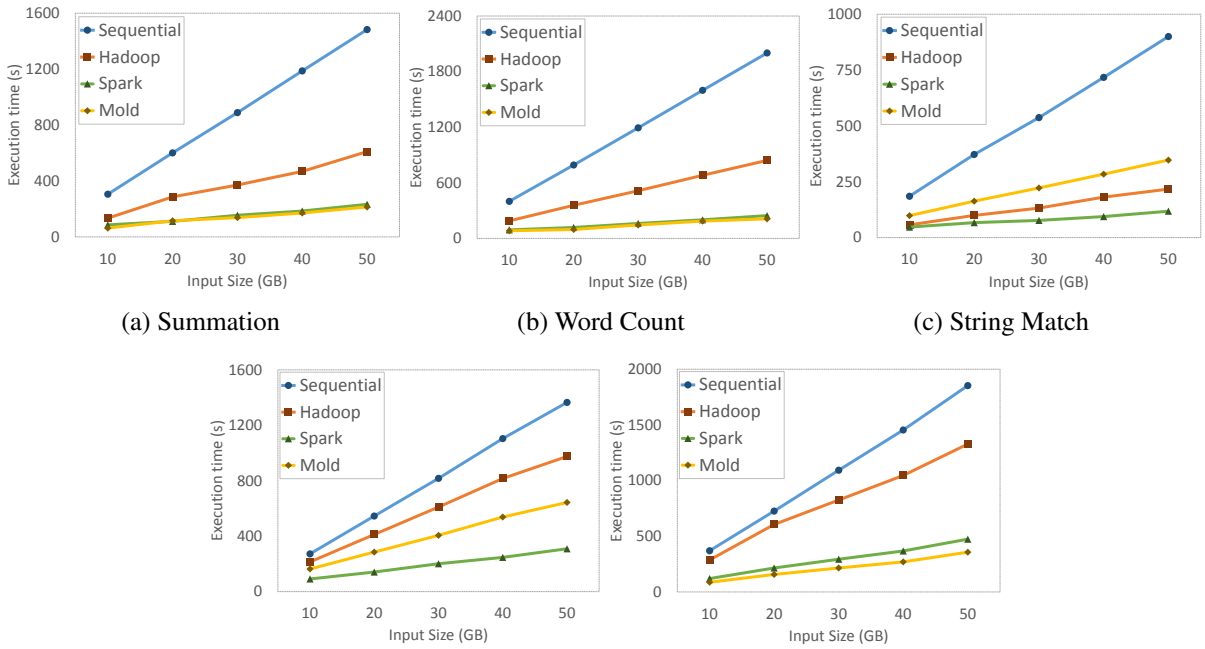
The generated **String Match** benchmark implementation parallelizes the search process. Each mapper iterates its assigned partition of text and emits $(key, true)$ whenever a key being searched is encountered. The data is locally aggregated by doing a disjunction of all values for a given key. Reduce side parallelism is leveraged as each key is aggregated in parallel.

The **3D Histogram** benchmark resembles the word count problem. Hence, the CASPER generated implementation iterates over each pixel and emits $((color, intensity), 1)$, where the key is a tuple of color and the intensity value. Data is aggregated in parallel in the reduce phase for each index of each histogram, for a total of 255×3 keys. As with the preceding benchmarks, data is locally aggregated before shuffling.

Linear Regression resembles the summation benchmark. All coefficients for a given point (x, y, $x*x$, $y*y$, and $x*y$) are calculated and emitted by the mapper, with a different key corresponding to each coefficient. For each key, the values are aggregated (by summation) locally before being globally reduced.

As is evident from all these benchmarks, CASPER generated non-trivial implementations. CASPER leveraged reduce side parallelism, reducing each output variable in parallel by assigning to each variable a unique ID and reducing data for each variable ID in parallel. For arrays, even greater parallelism was achieved by reducing each index of the array in parallel. CASPER also exploited map side parallelism by evaluating expressions before they are emitted by the mapper (e.g., as in Linear Regression). Lastly, CASPER used the reduce class as a combiner to locally aggregate data whenever the reduce input and output key-value pairs were of the same type.

To evaluate the quality of optimization CASPER achieved, we compared the runtime performance of the original sequential implementations to the Hadoop implementations generated. We also examined the performance when synthesized summaries were manually translated to the Spark framework. Finally, to add context, we compared the performance of Spark implementations generated by MOLD. Figure 5 graphs the results of all five benchmarks against different dataset sizes.



(a) Summation (b) Word Count (c) String Match

(d) Histogram (e) Linear Regression

Figure 5: Performance comparison of original implementations (blue) vs CASPER optimized Hadoop (orange) and Spark (Green) implementations. Performance of implementations when optimized using MOLD added for reference (yellow).

4.2.3 Alternate Implementations

As discussed, CASPER generates non-trivial implementations that effectively leverage the parallelism offered by Hadoop MapReduce. However, these implementations may not be the most efficient ones. For the 3D Histogram benchmark, an alternative Hadoop implementation would be to emit for each pixel in the input data key-value pairs of the form $(intensity, color)$. Hadoop would then group the data by the 256 intensity values. Aggregation would involve simply counting the number of times each color (Red, Green, or Blue) appears for a given key. Whether CASPER generates this implementation or the one discussed earlier in the paper depends upon which implementation the synthesizer discovers first. An important opportunity for future work is to enable CASPER to use heuristics to reason about the optimal implementation.

4.3 Performance of the Generated Benchmarks

In all five benchmarks, the generated Hadoop implementations were not only faster than their sequential counterparts but they also scaled better. Even for our smallest dataset (10GB), the Hadoop implementations outperformed the original implementations. The average speed up for Hadoop implementations across all benchmarks was 3.3× with a maximum speedup of 4.5× in the case of String Match.

Translating the summaries synthesized by CASPER into Spark yielded even higher speedups (up to 8.1×) since Spark uses cluster memory much more efficiently and minimizes disk I/O between different MapReduce stages. Extending CASPER to automatically generate Spark code from the synthesized summary is currently a work in progress.

5 Related Work

MapReduce DSLs. MapReduce is a popular programming model. It scales elastically, integrates well with distributed file systems, and abstracts away from the user low-level synchronization details. As such, many systems have been built that compile code down to MapReduce [3–5]. However, these systems provide their own high-level DSLs in which the users must use to express their computation. In contrast, CASPER works with native Java programs and infers rewrites automatically.

Source-to-Source Compilers. Many efforts translate programs directly from low-level languages into high-level DSLs. MOLD [15], a source-to-source compiler, relies on syntax-directed rules to convert native Java programs to Apache Spark. Unlike MOLD, we translate on the basis of program semantics. This eliminates the need for rewrite rules, which are difficult to generate and brittle to code pattern changes. Many source-to-source compilers have been built in other domains for similar purposes. For instance, [13] evaluates numerous tools for C to CUDA transformations. However, these compilers often require manual efforts to annotate the original source code. Our methodology works with code without any user annotation.

Synthesizing Efficient Implementations. Extensive literature describes the use of synthesis to generate efficient implementations and optimizing programs. [19] is the most recent research that attempts to synthesize MapReduce solutions with user-provided input and output examples. QBS [7] and STNG [11] both use verified lifting and synthesis to convert low-level languages to specialized high-level DSLs for database applications and stencil computations respectively.

6 Conclusion

This paper presented CASPER, a compiler that automatically re-targets native Java code to execute on Hadoop. CASPER uses verified lifting to convert code fragments in the original program to a high-level representation that can then be translated to generate equivalent Hadoop tasks for distributed data processing. We implemented a prototype of CASPER and evaluated its performance on several MapReduce benchmarks. Our experiments show that CASPER can translate all input benchmarks, and the generated programs can run on average 3.3× faster compared to their sequential counterparts.

7 Acknowledgment

The authors are grateful for the support of NSF grants CNS-1563788 and IIS-1546083 as well as DARPA award FA8750-16-2-0032, and DOE award DE-SC0016260.

References

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernandez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt & Sam Whittle (2015): *The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing*. Proceedings of the VLDB Endowment 8, pp. 1792–1803, doi:10.14778/2824032.2824076.

[2] *Apache Hadoop*. `http://hadoop.apache.org`. Accessed: 2016-04-19.

[3] *Apache Hive*. http://hive.apache.org. Accessed: 2016-04-20.

[4] *Apache Pig*. http://tensorflow.org/. Accessed: 2016-05-01.

[5] *Apache Spark*. https://spark.apache.org. Accessed: 2016-04-19.

[6] *Apache Storm*. http://storm.apache.org. Accessed: 2016-04-19.

[7] Alvin Cheung, Armando Solar-Lezama & Samuel Madden (2013): *Optimizing Database-backed Applications with Query Synthesis*. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, ACM, New York, NY, USA, pp. 3–14, doi:10.1145/2491956.2462180.

[8] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz & Chen Xiao (2007): *The Daikon System for Dynamic Detection of Likely Invariants*. *Sci. Comput. Program.* 69(1-3), pp. 35–45, doi:10.1016/j.scico.2007.01.015.

[9] *GraphLab Create*. https://dato.com/. Accessed: 2016-04-20.

[10] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. *Communications of the ACM* 12(10), pp. 576–580, doi:10.1145/363235.363259.

[11] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky & Armando Solar-Lezama (2016): *Verified Lifting of Stencil Computations*. *SIGPLAN Not.* 51(6), pp. 711–726, doi:10.1145/2980983.2908117.

[12] *MongoDB 3.2*. https://www.mongodb.org. Accessed: 2016-04-19.

[13] Cedric Nugteren & Henk Corporaal (2012): *Introducing 'Bones': A Parallelizing Source-to-source Compiler Based on Algorithmic Skeletons*. In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, ACM, New York, NY, USA, pp. 1–10, doi:10.1145/2159430.2159431.

[14] *Polyglot*. http://www.cs.cornell.edu/Projects/polyglot/. Accessed: 2016-05-01.

[15] Cosmin Radoi, Stephen J. Fink, Rodric Rabbah & Manu Sridharan (2014): *Translating Imperative Code to MapReduce*. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, ACM, New York, NY, USA, pp. 909–927, doi:10.1145/2660193.2660228.

[16] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand & Saman Amarasinghe (2013): *Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines*. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, ACM, New York, NY, USA, pp. 519–530, doi:10.1145/2491956.2462176.

[17] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski & Christos Kozyrakis (2007): *Evaluating MapReduce for Multi-core and Multiprocessor Systems*. In: *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, IEEE Computer Society, Washington, DC, USA, pp. 13–24, doi:10.1109/HPCA.2007.346181.

[18] *SKETCH*. https://people.csail.mit.edu/asolar/. Accessed: 2016-05-01.

[19] Calvin Smith & Aws Albarghouthi (2016): *MapReduce Program Synthesis*. *SIGPLAN Not.* 51(6), pp. 326–340, doi:10.1145/2980983.2908102.

[20] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat & Sanjit Seshia (2007): *Sketching Stencils*. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, ACM, New York, NY, USA, pp. 167–178, doi:10.1145/1273442.1250754.

[21] Saurabh Srivastava & Sumit Gulwani (2009): *Program Verification Using Templates over Predicate Abstraction*. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, ACM, New York, NY, USA, pp. 223–234, doi:10.1145/1542476.1542501.

[22] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky & Kunle Olukotun (2014): *Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages*. *ACM Trans. Embed. Comput. Syst.* 13(4s), pp. 134:1–134:25, doi:10.1145/2584665.

[23] *TensorFlow*. `http://tensorflow.org/`. Accessed: 2016-04-20.