Leveraging Parallel Data Processing Frameworks with Verified Lifting

Maaz Bin Safeer Ahmad
University of Washington
maazsaf@cs.washington.edu

Alvin Cheung
University of Washington
akcheung@cs.washington.edu

Many parallel data frameworks have been proposed in recent years to enable sequential programs leverage parallel processing. Unfortunately, to utilize the benefits of such frameworks, existing code often needs to be rewritten to the domain-specific languages that are supported by different frameworks. This rewriting process is tedious and error-prone, and even if developers are willing to invest resources in rewriting, they still face the problem of choosing which framework to use, as each framework delivers different amounts of performance improvement depending on the workload.

In this paper, we describe CASPER, a new compiler that automatically retargets sequential code written in Java to be executed on Hadoop, a parallel data framework that implements the Map-Reduce paradigm. Given a sequential code fragment, CASPER uses verified lifting to infer a high-level summary expressed using a logic specification language. The inferred summary is then compiled to be executed on Hadoop. The entire process is completely automated. We demonstrate that CASPER can automatically translate a set of Java benchmarks into Hadoop, and the translated results can execute up to 6.4× faster when compared to the sequential implementation.

1 Introduction

More data is being collected today than ever before. Computing has become more ubiquitous, storage is cheaper and better data collection tools are available. Both of these phenomena have become evident in various scientific domains where advances are increasingly data driven. As such, effectively analyzing and processing huge data sets is one of the major computational challenges that we currently face.

Over the past decade, many parallel data processing frameworks have been developed to handle very large datasets [2, 5, 6, 8, 11] and new ones continue to be released every few months [1, 11, 21]. Most parallel data processing frameworks come with domain-specific optimizations that are exposed either via library APIs [1, 2, 5, 6, 8, 21], or via high-level domain specific languages (DSLs) for users to express their computations [11, 15]. The idea is that if the computation can be expressed using such API calls or DSLs, then the resulting computation will be made efficient, thanks to the specialized optimization offered by the frameworks [3, 15, 19, 20].

Unfortunately, there are a number of issues with this approach, making many of these domain-specific frameworks inaccessible to non-expert users such as domain scientists and researchers. First, with so many frameworks available, each offering domain-specific optimizations for different workloads, it requires an expert to decide up front which framework is the most appropriate given a piece of code. To use these frameworks, end users often need to learn new APIs or DSLs [1, 2, 5, 6, 8, 11, 21] and rewrite their existing code. Doing so not only requires significant time and resources, but can also introduce new bugs into the application.

In addition, existing applications need to be rewritten to take advantage of such DSLs and frameworks. Rewriting applications requires understanding the intent of the original programmer, and manually written, low level optimizations often obscure high level intent. Finally, even after spending re-

Submitted to:
SYNT 2016

© M.B.S. Ahmad, A. Cheung

sources in learning new APIs and rewriting code, with newly emerging frameworks freshly rewritten code quickly becomes legacy applications. Users will have to repeatedly go through this process to keep up with new advances, and this requires significant time investment that could have been spent in doing scientific discovery instead.

One of the ways to make these parallel data processing frameworks more accessible is to build compilers that can automatically convert applications written in common general-purpose languages such as Java or Python to high performance processing frameworks such as Hadoop or Spark. Such compilers enable users to write their application in general-purpose languages that they are familiar with and then use the compiler to re-target portions of their code to a high performance DSLs [7, 10, 14]. Users would then be able to leverage the performance of these specialized frameworks without having to learn how to program individual DSLs. The resources required to convert legacy code would also be minimized and make the it easy to re-target legacy code even when documentation is not available. Finally, if the users decided to migrate on to a newer framework in the future, they just have to re-compile the same application that they wrote using a different compiler which targets the desired new framework.

This paper demonstrates the application of verified lifting that can be used to convert Java code fragments to MapReduce tasks. Verified lifting takes as input program fragments written in a general-purpose language, and uses program synthesis to find provably correct summaries of the code. These summaries are expressed in a logical specification language that encodes the semantics of the input code fragment. Once a summary is found, it (along with the original input code) can be translated to the target high performance DSL. The idea of verified lifting has been previously applied to database applications (QBS) [7], and stencil computations (STNG) [10]. In this paper we apply verified lifting to convert sequential data processing code to leverage parallel data processing frameworks. The problem statement is not new: it was first proposed in the MOLD compiler [14] that translates sequential Java code into targeting Apache Spark runtime. It uses pre-defined re-write rules to search the space of equivalent MapReduce implementations. MOLD scans the input code for code patterns that trigger the re-write rules. This approach has a number of limitations. It requires defining very complicated re-write rules a-priori, which is difficult to do. In addition, rules are extremely brittle to code pattern changes. In comparison, our approach deals with program semantics rather than program syntax, making it robust against code pattern changes. It also does not rely on pre-defined translation rules and can thus discover new solutions and optimizations that we did not even know existed.

We have implemented verified lifting in a system called CASPER that finds and converts code fragments written in sequential Java to Apache Hadoop. By converting sequential code fragments to Hadoop, CASPER parallelizes computation at crucial points throughout the program where most of the processing is concentrated. We have used CASPER to convert a number of benchmark programs with encouraging results. Overall, this paper has the following contributions:

- We describe how verified lifting can be used to re-target sequential Java applications to Hadoop MapReduce by converting code fragments within the application to Hadoop MapReduce tasks.
- We describe a new logical specification language that we have designed to express the intent of code fragments that can be converted to Hadoop.
- We describe how static program analysis techniques can be used to intelligently restrict the search space of all possible summaries that can be generated by our specification language, and how inductive synthesis can be used to find provably correct summaries for each input code fragment.
- We present preliminary results from using CASPER to identify and optimize code fragments written in Java. To show the potential of our approach, we evaluate our system on a number of benchmarks to demonstrate both its capabilities and limitations.

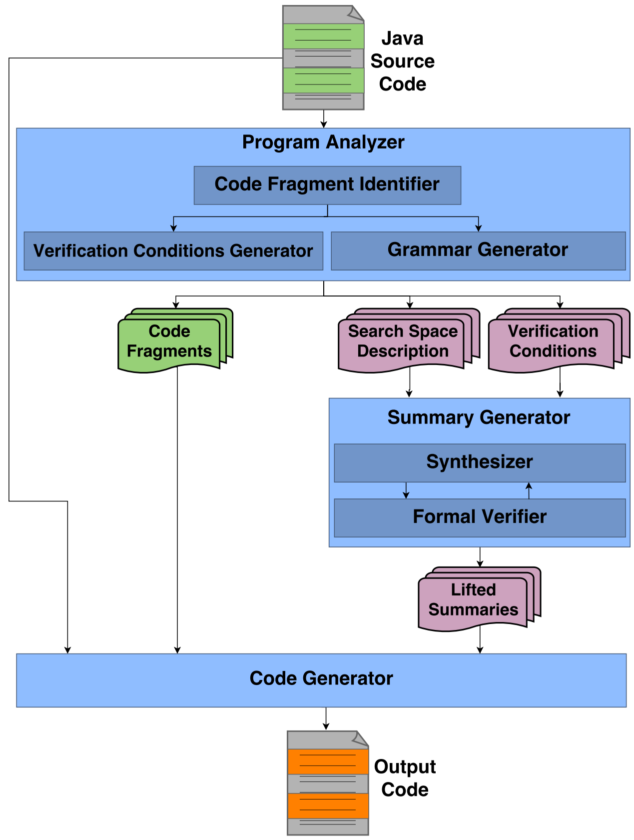


Figure 1: CASPER system architecture diagram

The rest of this paper is organized as follows. In §2, we describe the overall design of CASPER and illustrate how it can be used to convert sequential Java programs into Hadoop tasks. Then, in §3 we explain verified lifting and how each step of the process is implemented in CASPER. Finally, in §4, we evaluate how CASPER performs using a number of benchmarks and share our preliminary results, and conclude in §6.

2 System Overview

In this section we describe the architecture of CASPER. CASPER takes Java source code and automatically identifies and converts fragments to semantically equivalent MapReduce tasks implemented using Hadoop. A new optimized version of the original source code is generated where the original code fragments are replaced by invocations to the generated MapReduce tasks. Figure 1 shows the different components that make up CASPER and how they interact with each other in the compilation pipeline.

CASPER begins compilation by statically analyzing the input source code. Using static analysis, CASPER extracts code fragments that can potentially be translated. Next, CASPER generates a high-level summary of each extracted code fragment. The summary is expressed in a high-level logic specification language (to be discussed in §3.1) and is inferred by a program synthesizer. To quickly traverse the large search space of possible summaries, CASPER bounds the search space that the synthesizer considers during the search, and uses a bounded model-checking procedure to find if any candidate sum-

```
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.

mary exists for a given code fragment within the bounded search space. 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 summary generated by the synthesizer is semantically equivalent to the original code. After formal verification, the summary is used by the code generator module to produce code for Hadoop MapReduce tasks. The generated tasks are then inserted back into the original program, and a new version of the input code that leverages the Hadoop MapReduce framework is produced.

We demonstrate CASPER's compilation process using an example from the Phoenix benchmarks [16] that generates 3D histograms from image data read stored in a file. As shown 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, and counts the number of times that each value occurs for each color from Lines 9 to 11 in Figure 2a. The parallel program generated by CASPER, on the other hand, emits a key-value pair with the tuple (color, intensity) as key and 1 as value from Lines 5 to 7 in Figure 2c. The generated pairs are then grouped by the key and the frequency of each pair is calculated by adding all the 1's together in the reducer phase in Line 12. The input code and the output produced by CASPER is shown in Figure 2.

2.1 Program Analyzer

The program analyzer is the first component in CASPER's compilation pipeline. The goal of the program analyzer is two-fold. First, it identifies all code fragments that are candidates for conversion and secondly, it automatically prepares formal synthesis specifications for every candidate code fragment identified. The operations of the program analyzer are grouped into three sub-components as show in Figure 1, namely the code fragment identifier, verification conditions generator, and grammar generator.

The first sub-component of the program analyzer is the code fragment identifier. In the current prototype, CASPER identifies loops and extracts them as candidates for conversion. CASPER currently does not consider any non-looping code fragments as candidates for conversion, as many of them are difficult to express in MapReduce, such as recursive functions. MapReduce works best when the work can be divided into several independent parts, such as the case of loops without any carried dependencies. Recursive functions on the other hand often rely on the output of the recursive calls causing a dependency. On the other hand, CASPER currently ignores loops containing calls to external library methods that are unrecognized by the CASPER compiler. In §3.4, we discuss in detail the criteria for a code fragment to be extracted as candidate and consequently highlight the limitations of CASPER's current implementation.

The second sub-component is the grammar generator. The goal for the grammar generator is to confine the space of summaries that can be synthesized. This is needed as the space of all possible summaries that can be expressed in our logic specification language is too large. The grammar generator takes as input the code fragments extracted by the code fragment identifier, and statically analyzes each code fragment to extract semantic information. It then uses the extracted information to generate a formal grammar for every code fragment. The challenge is to generate a grammar expressive enough such that the correct summary can be formed by it, but not too expressive that the problem of searching for the summary becomes 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 component of the program analyzer is the verification conditions generator. This component uses Hoare logic [9] and static program analysis to generate verification conditions for each code fragment. Verification conditions are logical statements that describe what needs to be true for a given summary to be semantically equivalent to the original code fragment. While the grammar generated by the grammar generator defines the space of summaries to be searched, the verification conditions define the correctness specification that the synthesized summary must satisfy. We explain how CASPER uses Hoare style logic to verify program equivalence in §3.2.

The output of the verification conditions generator is a search template for the summary, with the search space specified by the grammar generator, and the verification conditions generator producing the logical assertions that need to be satisfied given a candidate summary. The template is used by the summary generator to search for a valid summary for the input code fragment.

2.2 Summary Generator

Taking the specifications generated by the program analyzer, the summary generator traverses the search space to find a summary that satisfies the verification conditions. The summary generator comprises of two modules: the program synthesizer and the formal verifier. The synthesizer takes the search space description and verification conditions generated earlier, and searches for a code summary that satisfies the verification conditions. To make the search problem tractable, the synthesizer employs a bounded model checking procedure. Under bounded model checking, the synthesizer only checks for correctness over a small sub-domain. When a promising candidate for the summary is found that satisfies the verification conditions in the sub-domain, it is passed onto the formal verifier which checks it for correctness over the entire unbounded domain. If the solution fails the formal verification step, it is eliminated from the search space and the search is restarted for a new candidate solution. Using this two-step verification process allows CASPER to quickly discard bad candidates. The more computationally expensive process of formal verification is reserved only for promising candidate solutions. At the end, the summary generator emits a verified summary for each of the code fragments that can then be translated to Hadoop. It is possible that the summary generator may exhaust the entire search space and not find a solution that verifies, in such cases CASPER must decide to either give up on the code fragment or expand the search space by appending more options to the grammar. In the current prototype CASPER achieves that via a preset timeout.

2.3 Hadoop Code Generator

The summaries found by the summary generator are expressed in the high-level logic specifications language. As a final step, CASPER use these summaries to generate Hadoop tasks. This is done using syntax-driven rules that translates each construct in the specification language into an equivalent Hadoop construct. A new modified version of the original input program is then constructed by replacing all of the code fragments that were successfully converted with equivalent Hadoop tasks. Each loop in the program 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

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

3.1 Verified Lifting

Verified lifting [10] is a general technique that infers semantics of code written in a general-purpose language by "lifting" it to summaries that are expressed using a high-level logic language. The summaries are specified using the logic language in the form of postconditions that describe the effects of the code on the *output variables*, i.e., variables that are modified within the code fragment. The goals for our logic specification language are as follows:

- To generate postconditions that CASPER can translate to the target platform. Any valid postcondition that cannot be translated is not useful. Therefore, the language should not include constructs that cannot be translated easily to the target.
- To generate non-trivial postconditions that exhibits parallel data processing. Obviously, a postcondition that executes the computation sequentially is undesirable. §4.2.2 discusses the sources of parallelism in MapReduce and how CASPER generates solutions that leverages them.

With that in mind, CASPER imposes the inferred summaries to be of the following form:

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

where *data* is the iterable input data collection and v' is the value of the output variable before the code fragment starts executing. The semantics of the functions involved in Equation 1 are as follows. The *map* function iterates over the input data while calling the f_m function for every index. f_m takes as input an index and the data collection to generate key-value pairs. Key-value pairs returned by invocations of f_m are collected and returned by *map*. The *reduce* function takes these key-value pairs, groups them by key, and calls the f_r function for each key and all values that correspond to that key. Function f_r aggregates all the values for the given key, and emits a single key-value pair. Like *map*, *reduce* collects all the aggregated key-value pairs and returns an associate array that maps each variable's ID to its final value. The variable identifier is a unique identifier assigned to every output variable. CASPER imposes the summaries (i.e., postconditions) to be of the form described in Eqn. (1) as they can be easily translated to Hadoop tasks.

In practice, CASPER focuses on loops since those can most likely be converted to Hadoop, and in that context input variables correspond to variables that are declared outside of the loop and are read inside the loop body Similarly, output variables are those that are modified inside the loop body but are not declared inside the loop body.

Note that in the above discussion, f_m and f_r remains unspecified. The goal of verified lifting is to find the definition of f_m and f_r that makes the inferred summary a valid one that preserves the semantics of the input code fragment. In CASPER, this is done by the synthesizer (to be discussed in §3.3) generating the implementation of these two functions, using the verification conditions computed by the program analyzer (to be discussed in §3.4) on each code fragment.

3.2 Verifying Equivalence

The summaries inferred by CASPER need to be semantically equivalent to the input code fragment. CASPER establishes the validity of the inferred postconditions using Hoare-style verification conditions [9]. Verification conditions of a code fragment represent the weakest preconditions 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 easy. For example, consider the imperative program statement `x := y + 3`. To show that the a candidate postcondition `x > 10` is a valid postcondition, we must prove that `y + 3 > 10` is true before the statement is executed. Therefore, `y + 3 > 10` is our verification condition. Computing this verification condition is as easy as doing backwards assignment in the post condition, i.e., replacing each instance of `x` in the postcondition with `y+3`. Computing verification conditions for a loop, however, is much more difficult as a loop invariant is needed. The loop invariant is an inductive hypothesis that asserts that the postcondition is true regardless of how many times the loop iterates. Hoare logic states that the following logic 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. $\forall \sigma.\ loopInvariant(\sigma) \wedge \neg loopCondition(\sigma) \rightarrow postCondition(\sigma)$

The first statement above asserts that for all program state (σ), the loop invariant must be true when the precondition is true, i.e., loop invariant must be true before entering the loop. The second statement asserts that for all possible program states, assuming that the loop invariant is true, and that the loop continues, then 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 given σ). The last statement asserts that for all

$$\begin{array}{l}
\mathsf{preCondition}(\mathsf{hR},\mathsf{hG},\mathsf{hB},\mathsf{i}) \equiv \\
\quad \mathsf{hR} = [0..0] \ \wedge \ \mathsf{hG} = [0..0] \ \wedge \ \mathsf{hB} = [0..0] \ \wedge \ \mathsf{i} = 0 \\[6pt]
\mathsf{postCondition}(\mathsf{data},\mathsf{hR},\mathsf{hG},\mathsf{hB}) \equiv \\
\quad \forall \ \ 0 \le \mathsf{j} < \mathsf{hR.length.} \ \ \mathsf{hR[j]} = \mathsf{reduce(map(data}, \mathsf{f_m}), \mathsf{f_r})[(0,\mathsf{j})] \ \wedge \\
\quad \forall \ \ 0 \le \mathsf{j} < \mathsf{hG.length.} \ \ \mathsf{hG[j]} = \mathsf{reduce(map(data}, \mathsf{f_m}), \mathsf{f_r})[(1,\mathsf{j})] \ \wedge \\
\quad \forall \ \ 0 \le \mathsf{j} < \mathsf{hB.length.} \ \ \mathsf{hB[j]} = \mathsf{reduce(map(data}, \mathsf{f_m}), \mathsf{f_r})[(2,\mathsf{j})] \\[6pt]
\mathsf{loopInvariant}(\mathsf{data},\mathsf{hR},\mathsf{hG},\mathsf{hB},\mathsf{i}) \equiv \\
\quad \mathsf{LoopCounterExp} \wedge \\
\quad \forall \ \ 0 \le \mathsf{j} < \mathsf{hR.length.} \ \ \mathsf{hR[j]} = \mathsf{reduce(map(data}[0:\mathsf{i}], \mathsf{f_m}), \mathsf{f_r})[(0,\mathsf{j})] \ \wedge \\
\quad \forall \ \ 0 \le \mathsf{j} < \mathsf{hG.length.} \ \ \mathsf{hG[j]} = \mathsf{reduce(map(data}[0:\mathsf{i}], \mathsf{f_m}), \mathsf{f_r})[(1,\mathsf{j})] \ \wedge \\
\quad \forall \ \ 0 \le \mathsf{j} < \mathsf{hB.length.} \ \ \mathsf{hB[j]} = \mathsf{reduce(map(data}[0:\mathsf{i}], \mathsf{f_m}), \mathsf{f_r})[(2,\mathsf{j})]
\end{array}$$

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

possible program states, if the loop invariant is true and the loop has terminated, then the postcondition must be true.

There are two challenges associated with finding the postconditions (and hence summaries) for code fragments that involve loops. First, *both* the loop invariants and postcondition need to be synthesized. However, CASPER only needs to find loop invariants that are logically strong enough to establish the soundness of the postcondition, i.e., those that satisfy 3. listed above. In addition, establishing the validity of the found invariants and postconditions require checking *all* possible program states, making the synthesis problem extremely challenging. We discuss how CASPER makes the search problem manageable in §3.3.3.

3.3 Searching for summaries

The goal of CASPER is to infer a summary for each code fragment, where each summary is expressed as a postcondition of the form explained in §3.1. In this section we discuss how CASPER uses synthesis to search for such a postcondition *and* the loop invariant it requires to prove the postcondition correct.

3.3.1 Generating Verification Conditions

In §3.2, we presented the general form of the summary used by CASPER. The program analyzer must generate the precondition, postcondition and loop invariant used in the verification conditions for every code fragment. Preconditions are generated by extracting the state of the program (the values of input and output variables) just before the loop starts executing. Whenever the value of a variable before the loop cannot be extracted, CASPER generates a new variable to represent the initial value. The postcondition is defined in the form explained in §3.1. The loop invariant has similar form as the postcondition, except that unlike the postcondition, which calls *map* and *reduce* on the entire data collection, the loop invariant only calls *map* and *reduce* on the subset of the collection up until the current index. Additionally, the loop invariant involves an expression describing the behavior of the loop counters.

Figure 3 shows the precondition, postcondition and the 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 post condition states that for each index j of hR, the value of $hR[j]$ should equal the output of map and reduce functions for key $(0, j)$.

3.3.2 Specifying Search Space

In this section, we discuss the generation of a grammar that is used by the synthesizer to generate bodies of f_m, f_r and the loop counter expression. By specifying a grammar to generate the missing components, CASPER restricts the space of possible function bodies that can be synthesized.

The body of the f_m function is a set number of emits. Initially, CASPER use the same number of emits as the number of output variables in the code fragment. If a solution cannot be found, the number of emits are incremented until the maximum limit (specified by the user) is reached. Each emit can be embedded within a conditional statement. Currently, the grammar does not synthesize loops in the f_m function. Expressions are synthesized for key, value and loop conditions using input variables and operators found in the original code fragment. Again, if synthesis fails to find a solution, other arithmetic and comparison operators are added to the grammar. Note that only variables and operators that are compatible with the type of the key and value are used.

The body of the f_r function is a loop reducing all values emitted by the map for a given key into one. The expression that aggregates these values is constructed using an expression grammar using the same operators available to f_m.

For the loop counter expression, CASPER only uses comparison operators. Number literals, loop counter variables and length of the data being iterated are used as potential operands.

Since our implementation of map is constant and iterates data with a stride of one, we need some way to handle loops with a stride greater than one. Emitting key value pairs at every index often leads to duplicate emits and incorrect results. A simple work-around for this problem is to introduce conditionals along with the mod and equality operator into the grammar. This allows CASPER to synthesize more complicated solutions which only emit key-value pairs for every n'th index for any arbitrary n.

Another insight is that for code fragments where the output variable is a collection, we want to explore solutions that exploit parallelism by reducing each index of the collection separately. This requires emitting indexes as keys. Therefore, we introduced the tuple construct, which allows emitting indexes as keys while still tagging the key-value pair with the variable ID.

Figure 4 shows the grammar generated for the 3D Histogram example using the above methodology. It is easy to see how the solution presented in Figure 2 may be generated from this.

3.3.3 Search Procedure

Despite all the constraints on the search space that we have already discussed, the space of possible summaries is still very large. To make synthesis tractable, CASPER first bounds the number of times non-terminals are expanded in the generated grammar. In addition, CASPER speeds up the verification process by splitting the problem into two parts: CASPER first uses a bounded-checking procedure to find candidate invariants and postconditions. For candidate invariants and postconditions that pass the bounded-checking procedure, CASPER then uses a theorem prover to establish soundness for all input program states. If the theorem prover fails (via a timeout), then the synthesizer is resumed to search for a new candidate summary in the same search space. When no more candidate summaries exist in the current search space, the synthesizer expands the grammar to increase the search space. This is done by either adding new operators, increasing the unwrap bound for the grammar or increasing the number of emits made by f_m as discussed earlier. This technique of iteratively expanding the search space is controlled through configuration parameters which are specified by the user. Eventually, the synthesizer will either find a verifiably correct summary or give up and not convert the code fragment.

The second insight allows CASPER to decouple the synthesis procedure from formal verification,

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

Figure 4: Grammar generated for 3D Histogram example.

and use off-the-shelf tools for each of the two sub-problems. While we have not implemented formal verification, this methodology works well in practice in terms of reducing the amount of synthesis time, as our experiments in §4.2.1 demonstrate.

3.4 Initial Code Extraction

As discussed in §2.1, the current CASPER prototype extracts loops from the input program as candidate code fragments. It does so by traversing the parsed abstract syntax tree (AST) of the input program source code to identify loops and extract them into individual fragments. Each extracted code fragment is parsed and analyzed to ensure they meet the following criteria:

- Within the code fragment, there are no unsupported library function calls. To synthesize summaries, CASPER identifies input and output variables (see §3.4.1), and the lack of library source code makes this difficult without building models that describe the semantics of the library functions. CASPER currently supports commonly used library functions such as methods of the `java.lang.{String,Integer}` and `java.util.{ArrayList,Map}` classes.
- There is no unstructured control flow in each of the loops. The current implementation of CASPER is unable to extract necessary semantics from such loops, such as the termination condition and loop stride.
- There are no nested loops. CASPER currently does not process nested loops. In case there are nested loops in the program, CASPER will attempt to optimize only the inner most loop.
- There are no assignment statements in the code fragment that can create an alias. Moreover, CASPER currently does not perform any alias analysis and assumes that none of the input variables in the code fragment are aliased. As such, objects of user defined types may not be assigned.

Fields of such objects may be read or modified as long as the fields themselves are a primitive type. Similarly, array indexes may be read or modified but not arrays as a whole. Support for assigning common immutable data structures such as `java.lang.{Integer,String}` has been built into the compiler.

Code fragments that do not satisfy the above criteria are filtered. After a loop has been marked for conversion, it is normalized to a simpler form before further analysis. The normalization process involves breaking down large instructions into smaller simpler ones and converting all loop constructs into `while(true)` loops.

3.4.1 Extracting Input and Ouput Variables

Additional passes are made on the normalized AST to extract input and output variables. CASPER examines each assignment statement inside the code fragment in isolation and extracts the targets of the assignments as output variables if they were not declared inside the loop body. Similarly, all variables that appear as a source of an assignment are extracted as input variables if they were not declared inside the code fragment and they are not in the output variable set. To determine whether a parameter to a function call is an input or output variable CASPER needs to look at the function source code to determine if it is modified or not. For library functions, this information is encoded into the CASPER. If a constant index of an array is accessed, then a separate input variable is created for the array element. However, if a dynamic access is also made, then the entire array is considered an input variable.

For the 3D Histogram example Figure 2, arrays `hR`, `hG` and `hB` are extracted as output variables and the `data` array is identified as an input variable. Variables `i`, `r`, `g` and `b` are all declared inside the loop body and thus not considered as input or 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. The `execute` method takes as parameters all input variables of the code fragment. It 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. Since the postcondition is already in the MapReduce form, the rules to translate them into the concrete syntax of Haddop are straightforward and omitted due to lack of space.

Figure 2c shows the final output code for the 3D Histogram example Figure 2. `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, namely `hR`, `hG` and `hB`, are compiled and returned by `execute` and are assigned into the original program's corresponding output variables as shown in Figure 2b. Code responsible for reconstructing the arrays from key-value pairs is not shown for brevity.

4 Evaluation

In this section we discuss our current implementation of CASPER and present the results in applying CASPER to a number of benchmarks.

4.1 Implementation

We have implemented a prototype of the approach described earlier called CASPER. The program analyzer in CASPER is implemented by extending the open source Java compiler Polyglot [13]. For synthesis, CASPER utilizes an off-the-shelf synthesizer called SKETCH [17]. SKETCH uses counter-example guided inductive synthesis as its core algorithm. The program analyzer in CASPER defines the verification conditions and search space in the SKETCH language. We have implemented the functions and data structures required to model the semantics of *map* and *reduce* in the SKETCH language as well. In addition, all the program specific user defined data types are automatically modeled in SKETCH. SKETCH performs bounded model checking to generate a summary which we use to generate the Hadoop Code. We have not implemented the the formal verification component in CASPER and relies solely on bounded model checking to verify correctness.

4.1.1 Platform For Evaluation

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

4.1.2 Benchmarks

We have evaluated the performance of CASPER on the following benchmarks. These benchmarks have been taken for the Phoenix suite of benchmarks [16] and represent traditional MapReduce problems.

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

4.2 Compilation Performance

In this section, we report the speed at which Hadoop implementations are generated by CASPER and discuss the quality of those implementations.

4.2.1 Scalability

In our experiments, CASPER was able to synthesize Hadoop implementations for all benchmarks within one hour. Simpler benchmarks such as **Summation** and **Word Count** were converted in under a minute and required only one iteration. No benchmark required more than two iterations to successfully synthesize an implementation for. Table 1 lists the average time required to synthesize a summary over five runs.

Benchmark	Time(s)	# of Iterations
Summation	13	1
Word Count	44	1
String Match	1406	2
3D Histogram	2355	2
Linear Regression	1801	2

Table 1: Time it takes CASPER to synthesize each benchmark.

4.2.2 Sources of Parallelism

In a MapReduce program, there are 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. Secondly, the reduce phase can be executed in parallel by grouping data to separate keys and aggregating for each key in parallel. Hadoop also supports the use of combiners. Combiners aggregate data locally on every node before the shuffle phase to offer additional parallelism and decrease the amount of data that needs to be shuffled. We now discuss the benchmarks processed by CASPER and how each leverages both map and reduce side parallelism.

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

The CASPER generated implementation of the **Word Count** benchmark emits $(word, 1)$ for each word encountered. All nodes aggregate data locally to compute word counts for the assigned data partition before the reducer aggregates the intermediate results. In addition to partially aggregating in parallel, CASPER uses the word as key. Therefore, the aggregation for different words will be performed in parallel.

The **String Search** 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 is similar to the word count problem. For each pixel the implementation emits $((color, index), 1)$, where the key is a tuple of color and index. Data is aggregated in parallel in the reduce phase for each index of each histogram for a total off 255x3 keys. As with the above benchmarks data is locally aggregated before the reduce phase.

Linear Regression is similar to 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 of the coefficients. For each key, the values are aggregated together (by summation) locally before being globally reduced.

As is evident through all these benchmarks, CASPER can generate non-trivial implementations. In particular, CASPER leverages reduce side parallelism by reducing each output variable in parallel by assigning a unique variable ID to each variable and grouping data based on this variable ID. For arrays, even greater parallelism can be achieved by reducing each index of the array in parallel. CASPER also exploits map side parallelism as some expressions may be evaluated before being emitted (e.g., as in Linear Regression). Lastly, CASPER uses the reduce class as a combiner to locally aggregate data whenever the reduce input and output key-value pairs are of the same type.

To evaluate the quality of optimization achieved by CASPER, we compare the runtime of the original sequential implementations against those generated by CASPER. Figure 5 graphs the results for all five benchmarks against different dataset sizes.

4.2.3 Alternate Implementations

As discussed in §4.2.2, CASPER generates non-trivial implementations that effectively leverage the parallelism offered by Hadoop MapReduce. However, these implementations may not be the most efficient one. In this section we use the 3D Histogram benchmark as an example to discuss alternate implementations that exist within the search space but are not considered, as the search process in CASPER stops as soon as it finds a valid summary.

For the 3D Histogram benchmark, an alternative Hadoop implementation is to emit for each value in the input data the pair $(value, color)$. Therefore, we would group the data by key for a total of 255 keys to be processed in parallel. Aggregation would involve simply counting the number of times each color appears for a given key. Which implementation CASPER will generate is currently a matter of which implementation is discovered first by the synthesizer. An important area for future work is to allow CASPER to reason about the best implementation through the use of heuristics.

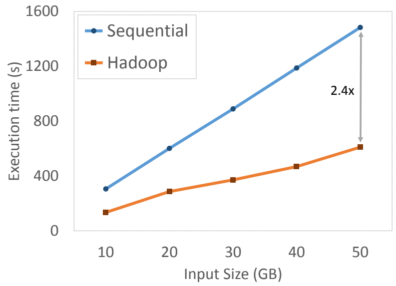
4.3 Performance of the Generated Benchmarks

In all five benchmarks, the generated Hadoop implementations are not only faster than their sequential counterparts but they also scale better as size of the data gets larger. Even for our smallest dataset of size 10GB, the Hadoop implementations outperform the original implementations by up to 5.4×, in the case of the Linear Regression benchmark. In general, the average speedup across all benchmarks is 3.3× as compared to sequential implementations.

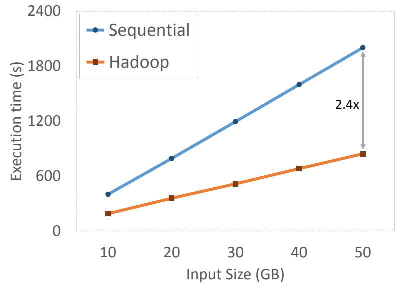
The best performance speedup was observed in Linear Regression. By computing coefficients for each point in the map phase, map side parallelism was achieved and by grouping values by keys reducer side parallelism was exploited. In addition, the generated reducer class was used as the combiner class as well.

5 Related Work

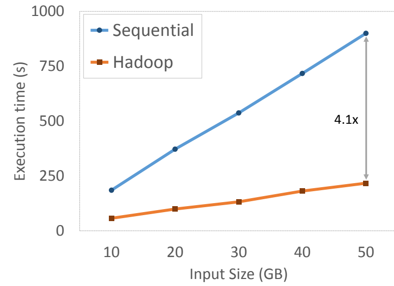
MapReduce DSLs. MapReduce is a popular programming model. It is scales elastically, integrates well with distributed file systems and abstracts away low level synchronization details from the user. As such, many systems haven been built which compile code down into MapReduce [3–5]. However, these systems come with their own high level DSLs that they require the user to write their programs in. We, in contrast, work with native Java programs.



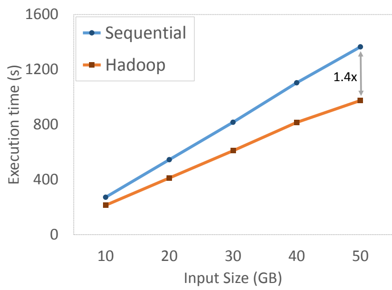
(a) Summation



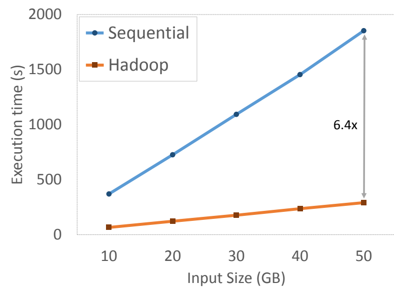
(b) Word Count



(c) String Match



(d) Histogram



(e) Linear Regression

Figure 5: Performance comparison of original implementations vs CASPER optimized implementations.

Source-to-Source Compilers. Efforts have been made to translate programs directly from low level languages into high level DSLs. MOLD [14] is a source-to-source compiler that relies on syntax directed rules to convert native Java programs to Apache Spark runtime. Our work differs from MOLD as 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 to ease programming of GPU processors. [12] evaluates numerous tools for C to CUDA transformations. However, these compilers often require user interference in the form of annotating the original source code. Our methodology works with un-annotated code.

Synthesizing Efficient Implementations. There is extensive literature on using synthesis to generate efficient implementations and optimizing programs. [18] is latest research work that attempts to synthesize MapReduce solutions by using user provided input and output examples. QBS [7] and STNG [10] both utilize verified lifting and synthesis to convert low level languages specialized high level DSLs.

6 Conclusion

In this paper we presented CASPER, a compiler automatically re-targets native Java code to Hadoop runtime. CASPER uses verified lifting to convert code fragments in the original program to a high level representation which can then be translated to generate equivelant Hadoop tasks for distributed data processing. We have implemented a prototype of CASPER and evaluated it's performance on several standard MapReduce benchmarks. Our tests show that programs optimized by CASPER can run upto 6.4× faster.

References

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernndez-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.

[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.

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

[9] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. Commun. ACM 12(10), pp. 576–580.

[10] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky & Armando Solar-Lezama (2016): *Verified Lifting of Stencil Computations*.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] Calvin Smith & Aws Albarghouthi (2016): *MapReduce Program Synthesis*.

[19] 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.

[20] 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.

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