Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications

casper.uwplse.org

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

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

ABSTRACT

MapReduce is a popular programming paradigm for developing large-scale, data-intensive computation. Many frameworks that implement this paradigm have recently been developed. To leverage these frameworks, however, developers must become familiar with their APIs and rewrite existing code. We present Casper, a new tool that automatically translates sequential Java programs into the MapReduce paradigm. Casper identifies potential code fragments to rewrite and translates them in two steps: (1) Casper uses *program synthesis* to search for a program summary (i.e., a functional specification) of each code fragment. The summary is expressed using a high-level intermediate language resembling the MapReduce paradigm and verified to be semantically equivalent to the original using a theorem prover. (2) Casper generates executable code from the summary, using either the Hadoop, Spark, or Flink API. We evaluated Casper by automatically converting real-world, sequential Java benchmarks to MapReduce. The resulting benchmarks perform up to 48.2× faster compared to the original.

ACM Reference Format:
Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications: casper.uwplse.org. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196891

1 INTRODUCTION

MapReduce [21], a popular paradigm for developing data-intensive applications, has varied and highly efficient implementations [4, 5, 8, 36]. All these implementations expose an application programming interface (API) to developers. While the concrete syntax differs slightly across the different APIs, they all require developers to organize their computation into *map* and *reduce* stages in order to leverage their optimizations.

While exposing optimization via an API shields application developers from the complexities of distributed computing, this approach contains a major drawback: for legacy applications to leverage MapReduce frameworks, developers must first understand the existing code's function and subsequently re-organize the computation using mappers and reducers. Similarly, novice programmers, unfamiliar with the MapReduce paradigm, must first learn the different APIs in order to express their computation accordingly. Both require a significant expenditure of time and effort. Further, each code rewrite or algorithm reformulation opens another opportunity to introduce bugs.

One way to alleviate these issues is to build a compiler that translates code written in another paradigm (e.g., imperative code) into MapReduce. Classical compilers, like logical to physical query plan compilers [29], use pattern matching rules, i.e., the compilers contain a number of rules that recognize different input code patterns (e.g., a sequential loop over lists) and translate the matched code fragment into the target (e.g., a single-stage map and reduce). As in query compilers, designing the rules is challenging: they must be both *correct*, i.e., the translated code should have the same semantics as the input, and sufficiently *expressive* to capture the wide variety of coding patterns that developers use to express their computations. We are aware of only one such compiler that translates imperative Java programs into MapReduce [38], and the number of rules involved in that compiler makes it difficult to maintain and modify.

This paper describes a new tool, Casper, that translates sequential Java code into semantically equivalent MapReduce programs. Rather than relying on rules to translate different code patterns, Casper is inspired by prior work on *cost-based query optimization* [41], which considers compilation to be a dynamic search problem. However, given that the inputs are general-purpose programs, the space of possible target programs is much larger than it is for query optimization. To address this issue, Casper leverages recent advances in program synthesis [11, 25] to search for MapReduce programs into which it can rewrite a given input sequential Java code fragment. To reduce the search space, Casper searches over the space of *program summaries*, which are expressed using a *high-level intermediate language (IR)* that we designed. As we discuss in §3.1, the IR's design succinctly expresses computations in the MapReduce paradigm yet remains sufficiently easy to translate into the concrete syntax of the target API.

To search for summaries, Casper first performs lightweight program analysis to generate a description of the space of MapReduce programs that a given input code fragment *might* be equivalent to. The search space is also described using our high-level IR. Casper then uses an off-the-shelf *program synthesizer* to perform the search, but it is guided by an *incremental search algorithm* and our *domain-specific cost model* to speed the process. A *theorem prover* is used to check whether the found program summary is indeed semantically

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196891

equivalent to the input. Once proved, the summary is translated into the concrete syntax of the target MapReduce API. Since the performance of the translated program often depends on input data characteristics (e.g., skewness), Casper generates multiple semantically equivalent MapReduce programs for a given input and produces a monitor module that switches among them based on runtime statistics; the monitor and switcher are automatically generated during compilation.

Compared to prior approaches, Casper does not require compiler developers to design or maintain any pattern matching rules. Furthermore, the entire translation process is completely automatic. We evaluated Casper using a number of benchmarks and real-world Java applications and demonstrated both Casper's ability to translate an input program into MapReduce equivalents and the significant performance improvements that result.

In summary, our paper makes the following contributions:

- We propose a new high-level intermediate representation (IR) to express the semantics of sequential Java programs in the MapReduce paradigm. The language is succinct to be easily translated into multiple MapReduce APIs, yet expressive to describe the semantics of many real-world benchmarks written in a general-purpose language. Furthermore, programs written in our IR can be automatically checked for correctness using a theorem prover (§4.1). The IR, being a high-level language, also lets us perform various *semantic optimizations* using our cost model (§5).
- We describe an efficient search technique for program summaries expressed in the IR without requiring any pattern matching rules. Our technique is both *sound* and *complete* with respect to the input search space. Unlike classical compilers, which rely on pattern matching to drive translation, our technique leverages program synthesis to dynamically search for summaries. Our technique is novel in that it incrementally searches for summaries based on cost. It also uses verification failures to systematically prune the search space and a hierarchy of search grammars to speed the summary search. This lets us translate benchmarks that have not been translated in any prior work (§4.1).
- There are often multiple ways to express the same input as MapReduce programs. Therefore, our technique can generate multiple semantically equivalent MapReduce versions of the input. It also automatically inserts code that collects statistics during program execution to adaptively switch among the different generated versions (§5.2).
- We implemented our methodology in Casper, a tool that converts sequential Java programs into three MapReduce implementations: Spark, Hadoop, and Flink. We evaluated the feasibility and effectiveness of Casper by translating real-world benchmarks from 7 different suites from multiple domains. Across 55 benchmarks, Casper translated 82 of 101 code fragments. The translated benchmarks performed up to 48.2× faster compared to the original ones and were competitive even with other distributed implementations, including manual ones (§7).

2 OVERVIEW

This section describes how we model the MapReduce programming paradigm and demonstrates by example how Casper translates sequential code into MapReduce programs.

```
@Summary(
  m = map(reduce(map(mat, λm1), λr), λm2)
  λm1 : (i, j, v) → {(i, v)}
  λr : (v1, v2) → v1 + v2
  λm2 : (k, v) → {(k, v/cols)} )
int[] rwm(int[][] mat, int rows, int cols) {
    int[] m = new int[rows];
    for (int i = 0; i < rows; i++) {
        int sum = 0;
        for (int j = 0; j < cols; j++)
            sum += mat[i][j];
        m[i] = sum / cols;
    }
    return m;
}
```

(a) Input: Sequential Java code

```
RDD rwm(RDD mat, int rows, int cols) {
  RDD m = mat.mapToPair(e -> new Tuple(e.i, e.v));
  m = m.reduceByKey((v1, v2) -> (v1 + v2));
  m = m.mapValues(v -> (v / cols));
  return m;
}
```

(b) Output: Apache Spark code

Figure 1: Using Casper to translate the row-wise mean benchmark to MapReduce (Spark).

2.1 MapReduce Operators

MapReduce organizes computation using two operators: *map* and *reduce*. The map operator has the following type signature:

$$\begin{aligned} \boldsymbol{map} &: (mset[\tau],\ \lambda_m) \longrightarrow mset[(\kappa, v)] \\ \boldsymbol{\lambda_m} &: \tau \longrightarrow mset[(\kappa, v)] \end{aligned}$$

Input into *map* is a multiset (i.e., bag) of type τ and a unary transformer function λ_m, which converts a value of type τ into a multiset of key-value pairs of types κ and v. The map operator then concurrently applies λ_m to every element in the multiset and returns the union of all multisets generated by λ_m.

$$\begin{aligned} \boldsymbol{reduce} &: (mset[(\kappa, v)],\ \lambda_r) \longrightarrow mset[(\kappa, v)] \\ \boldsymbol{\lambda_r} &: (v, v) \longrightarrow v \end{aligned}$$

Input into *reduce* is a multiset of key-value pairs and a binary transformer function λ_r, which combines two values of type v to produce a final value. The reduce operator first groups all key-value pairs by key (also known as shuffling) and then uses λ_r to combine, in parallel, the bag of values for each key-group into a single value. The output of *reduce* is another multiset of key-value pairs, where each pair holds a unique key. If the transformer function λ_r is commutative and associative, then *reduce* can be further optimized by concurrently applying λ_r to pairs of values in a key-group.

Casper's goal is to translate a sequential code fragment into a MapReduce program that is expressed using the *map* and *reduce* operators. The challenges in doing so are: (1) identify the correct sequence of operators to apply, and (2) implement the corresponding transformer functions. We next discuss how Casper overcomes these challenges.

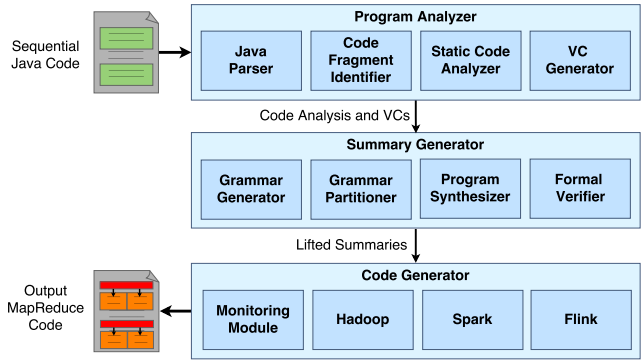


Figure 2: Casper's system architecture. Sequential code fragments (green) are translated into MapReduce tasks (orange).

2.2 Translating Imperative Code to MapReduce

Casper takes in Java code with loop nests that sequentially iterate over data and translates the code into a semantically equivalent MapReduce program to be executed by the target framework. To demonstrate, we show how Casper translates a real-word benchmark from the Phoenix suite [39].

As shown in Figure 1(a), the benchmark takes as input a matrix (`mat`) and computes, using nested loops, the column vector (`m`) containing the mean value of each row in the matrix. Assume the code is annotated with a *program summary* that helps with the translation into MapReduce. The program summary, written using a high-level intermediate representation (IR), describes how the output of the code fragment (i.e., `m`) can be computed using a series of *map* and *reduce* stages from the input data (i.e., `mat`), as shown in lines 1 to 5 in Figure 1(a). While the summary is not executable, translating from that into the concrete syntax of a MapReduce framework (say, Spark) would be much easier than translating from the original input code. This is shown in Figure 1(b) where the *map* and *reduce* primitives from our summary are translated into the corresponding Spark API calls.

Unfortunately, the input code does not have such a summary, which must therefore be inferred. Casper does this via program synthesis and verification, as we explain in §3.

2.3 System Architecture

Figure 2 shows Casper's overall design. We now discuss the three primary modules that make up Casper's compilation pipeline.

First, the *program analyzer* parses the input code into an Abstract Syntax Tree (AST) and uses static program analysis to identify code fragments for translation (§6.1). In addition, for each identified code fragment, it prepares: (1) a *search space description* encoded using our high-level IR that lets the synthesizer search for a valid program summary (§3.1), and (2) *verification conditions* (VCs) (§3.3) to automatically ascertain that the induced program summary is semantically equivalent to the input.

Next, the *summary generator* synthesizes and verifies program summaries (§3.4 and §4.1). To speed up the search, it partitions the search space so that it can be efficiently traversed using our incremental synthesis algorithm (§4.2).

Once a summary is inferred, the *code generator* translates it into executable code. Casper currently supports three MapReduce frameworks: Spark, Hadoop, and Flink. Additionally, this component also generates code that collects data statistics to adaptively choose among different implementations during runtime (§5.2).

$$
\begin{array}{rcl}
PS & := & \forall v.\ v = MR \mid \forall v.\ v = MR[v_{id}] \\
MR & := & map(MR, \lambda_m) \mid reduce(MR, \lambda_r) \mid join(MR, MR) \mid data \\
\lambda_m & := & f : (val) \rightarrow \{Emit\} \\
\lambda_r & := & f : (val_1, val_2) \rightarrow Expr \\
Emit & := & emit(Expr, Expr) \mid if(Expr)\ emit(Expr, Expr) \mid \\
 & & if(Expr)\ emit(Expr, Expr)\ else\ Emit \\
Expr & := & Expr\ op\ Expr \mid op\ Expr \mid f(Expr, Expr, \ldots) \mid \\
 & & n \mid var \mid (Expr, Expr)
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
v & \in & Output\ Variables & v_{id} & \in & Variable\ ID, \\
op & \in & Operators & f & \in & Library\ Methods
\end{array}
$$

Figure 3: Excerpt of the IR for program summaries (PSs), a full description of which is provided in Appendix B.

3 SYNTHESIZING PROGRAM SUMMARIES

As discussed, Casper discovers a program summary for each code fragment before translation. Technically, a program summary is a *postcondition* [26] of the input code that describes the program state after the code fragment is executed. In this section, we explain: (1) the IR Casper uses to express summaries, (2) how Casper verifies a summary's validity, and (3) the search algorithm Casper uses to find valid summaries given a search space description.

3.1 A High-level IR for Program Summaries

One approach to synthesize summaries directly searches in programs written in the target framework's API. This does not scale well; Spark alone offers over 80 high-level operators, even though many of them have similar semantics and differ only in their implementation or syntax (e.g., map, flatMap, filter). To speed up synthesis, we instead search in programs written in a high-level IR that abstracts away syntactical differences and describes only the functionality of a few essential operators. The goals of the IR are: (1) to express summaries that are translatable into the target API, and (2) to let the synthesizer efficiently search for summaries that are equivalent to the input program. To address these goals, Casper's IR models two MapReduce primitives that are similar to the *map* and *fold* operators in Haskell (see §2.1). In addition, our IR models the *join* primitive, which takes as input two multisets of key-value pairs and returns all pairs of elements with matching keys. The IR does not currently model the full range of operators across different MapReduce implementations; however, it already lets Casper capture a wide array of computations expressible using the paradigm and is sufficiently general to be translatable into different MapReduce APIs while keeping the search problem tractable, as we demonstrate in §7.

Figure 3 shows a subset of Casper's IR, used to express both program summaries and the search space. The IR assumes that program summaries are expressed in the stylized form shown in Figure 3 as *PS*, which states that each output variable v (i.e., a variables updated in the code fragment), must be computed using a sequence of *map*, *reduce* and *join* operations over the inputs (e.g., the arrays or collections being iterated). While doing so ensures that the summary is translatable into the target API, the implementations of λ_m and

λ_r for the *map* and *reduce* operators depend on the code fragment being translated. We leave these functions to be synthesized and restrict the body of λ_m to a sequence of *emit* statements, where each *emit* statement produces a single key-value pair, and the body of λ_r is an expression that evaluates to a single value of the required type. Besides *emit*, the bodies of λ_m and λ_r's can consist of conditionals and other operations on tuples, as shown in Figure 3. The output of the MapReduce expression is an associative array of key-value pairs; the unique key v_{id} for each variable is used to access the computed value of that variable. Appendix B lists the full set of types and operators that our IR supports.

3.2 Defining the Search Space

In addition to program summaries, Casper also uses the IR to describe the search space of summaries for the synthesizer. It does so by generating a *grammar* for each input code fragment, like the one shown in Figure 3. The synthesizer traverses the grammar by expanding on each production rule and checks whether any generated candidate constitutes a valid summary (as explained in §3.3).

To generate the search space grammar, Casper analyzes the input code to extract the following properties and their type information:

(1) Variables in scope at the beginning of the input code
(2) Variables that are modified within the input code
(3) The operators and library methods used

The code analyzer extracts these properties using standard program analyses. It computes (1) and (2) using live variable and dataflow analysis [1], and it computes (3) by scanning functions that are invoked in the input code. We currently assume that input variables are not aliased to each other and put guards on the translated code to ensure that is the case.[1] Appendix D shows the analysis results for the TPC-H Q6 benchmark, and we discuss the limitations of our program analyzer module implementation in §6.1.

Given this information, the summary generator builds a search space grammar that is specialized to the code fragment being translated. Figure 6 shows sample grammars that are generated for the code shown in Figure 1(a).[2] The input code uses addition and division; hence, the grammar includes addition and division in its production rules for λ_m and λ_r. Furthermore, Casper also uses type information of variables to prune invalid production rules in the grammar. For instance, if the output variable v is of type *int*, the final operation in the synthesized MapReduce expression must evaluate to a value of type *int*. Since the output type of a reduce operation is inferred from the type of its input, we can propagate this information to restrict the type of values the reduce operation accepts. To make synthesis tractable and the search space finite, Casper imposes recursive bounds on the production rules. For instance, it limits the number of MapReduce operations a program summary can use and the number of *emit* statements in a single transformer function. In §4.2, we discuss how Casper further specializes the search space by changing the set of production rules available in the grammar or specifying different recursive bounds.

[1]Thus, if variable handles v1 and v2 are both inputs into the same code fragment, Casper wraps the translated code as: `if (v1 != v2) { [Casper translated code] } else { [original code] }`. Computing precise alias information requires more engineering [43] and does not impact our approach.
[2]Refer to Appendix D to see how a grammar can be encoded in our IR.

$$invariant(m, i) \equiv \quad 0 \leq i \leq rows \ \wedge$$
$$m = map(reduce(map(mat[0..i], \lambda_{m1}), \lambda_r), \lambda_{m2})$$

(a) Outer loop invariant

Initiation	$(i = 0) \rightarrow Inv(m, i)$
Continuation	$Inv(m, i) \wedge (i < rows) \rightarrow$ $Inv(m[i \mapsto sum(mat[i])/cols], i + 1)$
Termination	$Inv(m, i) \wedge \neg(i < rows) \rightarrow PS(m, i)$

(b) Verification conditions to ascertain the correctness of the program summary PS given loop invariant Inv

Figure 4: Proof of soundness for the row-wise mean benchmark.

3.3 Verifying Program Summaries

To search for a valid summary within the search space, Casper requires a way to check whether a candidate summary is semantically equivalent to the input code. It does so using standard techniques in program verification, namely, by creating *verification conditions* based on Hoare logic [26]. Verification conditions are Boolean predicates that, given a program statement S and a postcondition (i.e., program summary) P, state what must be true *before* S is executed in order for P to be a valid postcondition of S. Verification conditions can be systematically generated for imperative program statements, including those processed by Casper [33, 47]. However, each loop statement requires an extra *loop invariant* to construct an inductive proof. Loop invariants are Boolean predicates that are true before and after every execution of the loop body regardless of how many times the loop executes.

The general problem of inferring the strongest loop invariants or postconditions is undecidable [33, 47]. Unlike prior work, however, two factors make our problem solvable: first, our summaries are restricted to only those expressible using the IR described in §3.1, which lacks many problematic features (e.g., pointers) that a general-purpose language would have. Moreover, we are interested only in finding loop invariants that are *strong enough* to establish the validity of the synthesized program summaries.

As an example, Figure 4(a) shows an outer loop invariant *Inv*, which can be used to prove the validty of the program summary shown in Figure 1(a). Figure 4(b) shows the verification conditions Casper constructs to state what the program summary and invariant must satisfy. We can check that this loop invariant and program summary are indeed valid based on Hoare logic as follows. First, the *initiation* clause asserts that the invariant holds before the loop, i.e., when `i` is zero. This is true because the invariant asserts that the MapReduce expression is true only for the first i rows of the input matrix. Hence, when `i` is zero, the MapReduce expression is executed on an empty dataset, and the output value for each row is 0. Next, the *continuation* clause asserts that after one more execution of the loop body, the i^{th} index of output vector `m` should hold the mean for the i^{th} row of the matrix `mat`. This is true since the value of `i` is incremented inside the loop body, which implies that the mean for the i^{th} row has been computed. Finally, the *termination* condition completes the proof by asserting that if the invariant is true, and `i` has reached the end of the matrix, then the program

summary *PS* must now hold as well. This is true since `i` now equals the number of rows in the matrix, and the loop invariant asserts that `m` equals the MapReduce expression executed over the entire matrix, which is the same assertion as our program summary.

Casper formulates the search problem for finding program summaries by constructing the verification conditions for the given code fragment and leaving the body of the summary (and any necessary invariants for loops) to be synthesized. For the program summary and invariants, the search space is expressed using the same IR as discussed in §3.1. Formally, the synthesis problem is:

$$\exists\, ps,\ inv_1, \ldots, inv_n.\ \ \forall \sigma.\ VC(P, ps, inv_1, \ldots, inv_n, \sigma) \quad (1)$$

In other words, Casper's goal is to find a program summary ps and any required invariants $inv_1, \ldots, inv_n$ such that for all possible program states σ, the verification conditions for the input code fragment P are true. After the synthesizer has identified a candidate summary and invariants, Casper sends them and the verification conditions to a theorem prover (see §4.1), and to the code generator to generate executable MapReduce code if the program summary is proven to be correct.

3.4 Search Strategy

Casper uses an off-the-shelf program synthesizer, Sketch [42], to infer program summaries and loop invariants. Sketch takes as input: (1) a set of candidate summaries and invariants encoded as a grammar (e.g., Figure 3), and (2) the correctness specification for the summary in the form of verification conditions. It then attempts to find a program summary (and any invariants needed) using the provided grammar such that the verification conditions hold true.

The universal quantifier in Eq.1 make the synthesis problem challenging. Therefore, Casper uses a two-step process to ensure that the found summary is valid. First, it leverages Sketch's *bounded model checking* to verify the candidate program summary over a finite (i.e., "bounded") subset of all possible program states. For example, Casper restricts the maximum size of the input dataset and the range of values for integer inputs. Finding a solution for this weakened specification can be done very efficiently by the synthesizer. Once a candidate program summary can be verified for the bounded domain, Casper passes the summary to a theorem prover to determine its soundness over the entire domain, which is more expensive computationally. Casper currently translates the summary along with an automatically generated proof script to Dafny [31] for full verification. This two-step verification makes Casper's synthesis algorithm sound, without compromising efficiency.

3.4.1 Synthesis Algorithm. Figure 5 (lines 1 to 8) shows the core CEGIS [45] algorithm Casper's synthesizer uses. The algorithm is an iterative interaction between two modules: a candidate program summary generator and a bounded model checker. The candidate summary generator takes as input the IR grammar G, the verification conditions for the input code fragment VC, and a set of concrete program states Φ. To start the process, the synthesizer populates Φ with a few randomly chosen states, and generates program summary candidate ps and any needed invariants $inv_1, \ldots, inv_n$ from G such that $\forall \sigma \in \Phi\ .\ VC(ps, inv_1, \ldots, inv_n, \sigma)$ is true. Next, the bounded model checker verifies whether the candidate program summary holds over the bounded domain. If it does, the algorithm returns ps as the solution. Otherwise, the model checker returns a counter-example state ϕ such that $VC(ps, inv_1, \ldots, inv_n, \phi)$ is false. The algorithm adds ϕ to Φ and restarts the program summary generator. This continues until either a program summary is found that passes bounded model checking or the search space is exhausted.

```
1  function synthesize (G, VC):
2    Φ = {} // set of random program states
3    while true do
4      ps, inv_1..n = generateCandidate(G, VC, Φ)
5      if ps is null then return null // search space exhausted
6      φ = boundedVerify(ps, inv_1..n, VC)
7      if φ is null then return (ps, inv_1..n) // summary found
8      else Φ = Φ ∪ φ // counter-example found
9
10 function findSummary (A, VC):
11   G = generateGrammar(A)
12   Γ = generateClasses(G)
13   Ω = {} // summaries that failed verification
14   Δ = {} // summaries that passed verification
15   for γ ∈ Γ do
16     while true do
17       c = synthesize(γ - Ω - Δ, VC)
18       if c is null and Δ is null then
19         break // move to next grammar class
20       else if c is null then
21         return Δ // search complete
22       else if fullVerify(c, VC) then Δ = Δ ∪ c
23       else Ω = Ω ∪ c
24   return null // no solution found
```

Figure 5: Casper's search algorithm.

A limitation of the CEGIS algorithm is that, while efficient, the found program summary might be true only for the finite domain and thus will be rejected by the theorem prover when checking for validity over the entire domain. In this case, Casper dynamically changes the search space grammar to exclude the candidate program summary that does not verify and restarts the synthesizer to generate a new candidate summary using the preceding algorithm. We discuss this process in detail in §4.1.

4 IMPROVING SUMMARY SEARCH

We now discuss the techniques Casper uses to make the search for program summaries more robust and efficient.

4.1 Leveraging Verifier Failures

As mentioned, the program summary that the synthesizer returns can fail theorem prover validation due to the bounded domain used during search. For instance, assume we bound the integer inputs to have a maximum value of 4 in the synthesizer. In this bounded domain, the expressions `v` and `Math.min(4,v)` (where `v` is an input integer) are deemed to be equivalent even though they are not equal in practice. While prior work [17, 28] simply fails to translate such benchmarks if the theorem prover rejects the candidate summary, Casper uses a two-phase verification technique to eliminate such candidates. This ensures that Casper's search is complete with respect to the search space defined by the grammar.

To achieve completeness, Casper must first prevent summaries that failed the theorem prover from being regenerated by the synthesizer. A naive approach would be to restart the synthesizer

Property	G_1	G_2	G_3
Map/Reduce Sequence	m	m → r	m → r → m
# Emits in λ_m	1	2	2
Key/Value Type	int	int	int or Tuple<int,int>

$$G1 := map(mat, \lambda_m) \qquad \lambda_m := \begin{cases} (i,j,v) \rightarrow [(i,j)] \\ (i,j,v) \rightarrow [(i,v)] \\ (i,j,v) \rightarrow [(j, v+i)] \\ (i,j,v) \rightarrow [(i+j, v)] \\ \vdots \end{cases}$$

$$G2 := reduce(map(mat, \lambda_m), \lambda_r) \qquad \lambda_m := \begin{cases} (i,j,v) \rightarrow [(i,v)] \\ (i,j,v) \rightarrow [(j, v+i)] \\ (i,j,v) \rightarrow [(i,j),(v,1)] \\ \vdots \end{cases} \qquad \lambda_r := \begin{cases} (v_1, v_2) \rightarrow v_1 \\ (v_1, v_2) \rightarrow v_2 + 4 \\ (v_1, v_2) \rightarrow v_1 + v_2 \\ \vdots \end{cases}$$

$$G3 := map(reduce(map(mat, \lambda_{m1}), \lambda_r), \lambda_{m2}) \qquad \lambda_{m1} := \begin{cases} (i,j,v) \rightarrow [(i,v)] \\ (i,j,v) \rightarrow \left[(i,(v,i))\right] \\ (i,j,v) \rightarrow [(i+1, j-v), (i,v)] \\ \vdots \end{cases} \qquad \lambda_r := \begin{cases} (v_1, v_2) \rightarrow v_1 \\ (v_1, v_2) \rightarrow v_1 + v_2 \\ (v_1, v_2) \rightarrow (v_1.1, v_2.2) \\ \vdots \end{cases} \qquad \lambda_{m2} := \begin{cases} (k,v) \rightarrow [(k,v),(v,k)] \\ (k,v) \rightarrow \left[((v.1,k), v.2)\right] \\ (k,v) \rightarrow [(k, v/cols)] \\ (k,v) \rightarrow if(v > i)[(k,v)] \\ \vdots \end{cases}$$

Figure 6: Incremental grammar generation. Casper generates a hierarchy of grammars to optimize search.

until a new summary is found, assuming that the algorithm implemented by the synthesizer is non-deterministic. However, this approach is incomplete because the algorithm may never terminate since it can continually return the same incorrect summary. Instead, Casper modifies the search space to ensure forward progress. Recall from §3.4 that the search space for candidate summaries $\{c_1, \ldots, c_n\}$ is specified using an input grammar that is generated by the program analyzer and passed to the synthesizer. Thus, to prevent a candidate c_f that fails the theorem prover from being repeatedly generated from grammar G, Casper simply passes in a new grammar $G - \{c_f\}$ to the synthesizer. This is implemented by passing additional constraints to the synthesizer to block a summary from being regenerated.

Theorem. Casper's algorithm for inferring program summaries is sound and complete with respect to the given search space.
A proof sketch for this theorem is provided in Appendix A.

Figure 5 shows how Casper infers program summaries and invariants. Casper calls the synthesizer to generate a candidate summary c on line 17 and attempts to verify c by passing it to the theorem prover on line 22. If verification fails, c is added to Ω, the set of incorrect summaries, and the synthesizer is restarted with a new grammar $G - \Omega$. We explain the full algorithm in §4.3.

In §7.3.2, we provide experimental results that illustrate how our two-phase verification algorithm effectively finds program summaries even when faced with verification failures.

4.2 Incremental Grammar Generation

Although Casper's search algorithm is complete, the space of possible summaries to consider remains large. To address this, Casper incrementally expands the search space for program summaries to speed up the search. It does this by: (1) adding new production rules to the grammar, and (2) increasing the number of times that each product rule is expanded.

The benefits of this approach are twofold. First, since the search time for a valid summary is proportional to search space size, Casper often finds valid summaries quickly, as our experiments show. Second, since larger grammars are more syntactically expressive, the found summaries are likely to be more expensive computationally. Hence, biasing the search towards smaller grammars likely produces program summaries that run more efficiently. Although this is not sufficient to guarantee optimality of generated summaries, our experiments show that in practice Casper generates efficient solutions (§7.2).

To implement incremental grammar generation, Casper partitions the space of program summaries into different *grammar classes*, where each class is defined based on these syntactical features: (1) the number of MapReduce operations, (2) the number of *emit* statements in each *map* stage, (3) the size of key-value pairs emitted in each stage, as inferred from the types of the key and value, and (4) the length of expressions (e.g., $x + y$ is an expression of length 2, while $x + y + z$ has a length of 3). All of these features are implemented by altering production rules in the search space grammar. A grammar hierarchy is created such that all program summaries expressible in a grammar class G_i are also expressible in a higher level class, i.e., G_j where $j > i$.

4.3 Casper's Search Algorithm for Summaries

Figure 5 shows Casper's algorithm for searching program summaries. The algorithm begins by constructing a grammar G using the results of program analysis A on the input code. First, Casper partitions the grammar G into a hierarchy of grammar classes Γ (line 12). Then, it incrementally searches each grammar class $\gamma \in \Gamma$, invoking the synthesizer to find summaries in γ (line 17). Each summary (and invariants) the synthesizer returns is checked by a theorem prover (line 22); Casper saves the set of correct program summaries in Δ and all summaries that fail verification in Ω. Each synthesized summary (correct or not) is eliminated from the search space, forcing the synthesizer to generate a new summary each time, as explained in §4.1. When the grammar γ is exhausted, i.e., the synthesizer has returned null, Casper returns the set of correct summaries Δ if it is non-empty. Otherwise, no valid solution was found, and the algorithm proceeds to search the next grammar class in Γ. If Δ is empty after exploring every grammar in Γ, i.e., no summary could be found in the entire search space, the algorithm returns null.

4.4 Row-wise Mean Revisited

We now illustrate how `findSummary` searches for program summaries using the row-wise mean benchmark discussed in §2.2. Figure 6 shows three sample (incremental) grammars Casper generated as a result of calling `generateClasses` (Figure 5, line 12) along with their properties. For example, the first class, G_1, consists of program summaries expressed using a single *map* or *reduce* operator, and the transformer functions λ_m and λ_r are restricted to emit only one integer key-value pair. A few candidates for λ_m are shown

in the figure. For instance, the first candidate, $(i, j, v) \rightarrow [(i, j)]$, maps each matrix entry to its row and column as the output.

If `findSummary` fails to find a valid summary in G_1 for the benchmark, it advances to the next grammar class, G_2. G_2 expands upon G_1 by including summaries that consist of two *map* or *reduce* operators, and each λ_m can emit up to 2 key-value pairs. The search next moves to G_3, where G_3 expands upon G_2 with summaries that include up to three *map* or *reduce* operators, and the transformers can emit either integers or tuples. As shown in Figure 1(a), a valid summary is finally found in G_3 and added to Δ. Search continues in G_3 for other valid summaries in the same grammar class. The search terminates after all valid summaries in G_3, i.e., those returned by the synthesizer and fully verified, are found. This includes the one shown in Figure 1(a).

5 FINDING EFFICIENT TRANSLATIONS

There often exist many semantically equivalent MapReduce implementations for a given sequential code fragment, with significant performance differences. Many frameworks come with optimizers that perform low-level optimizations (e.g., fusing multiple map operators). However, performing *semantic transformations* is often difficult. For instance, at least three different implementations of the StringMatch benchmark exist in MapReduce, and they differ in the type of key-value pairs the *map* stage emits (see §7.4). Although it is difficult for a low-level optimizer to discover these equivalences by syntax analysis, Casper can perform such optimization because it searches for a high-level program summary expressed using the IR. We now discuss Casper's use of a cost model and runtime monitoring module for this purpose.

5.1 Cost Model

Casper uses a cost model to evaluate different semantically equivalent program summaries that are found for a code fragment. Because Casper aims to translate data-intensive applications, its cost model estimates data transfer costs as opposed to compute costs.

Each synthesized program summary is a sequence of *map*, *reduce* and *join* operations. The semantics of these operations are known, but the transformer functions that they use (λ_m and λ_r) are synthesized and determine the operation's cost. We define the cost functions of the *map*, *reduce* and *join* operations below:

$$cost_m(\lambda_m, N, W_m) = W_m * N * \sum_{i=1}^{|\lambda_m|} sizeOf(emit_i) * p_i \quad (2)$$

$$cost_r(\lambda_r, N, W_r) = W_r * N * sizeOf(\lambda_r) * \epsilon(\lambda_r) \quad (3)$$

$$cost_j(N_1, N_2, W_j) = W_j * N_1 * N_2 * sizeOf(emit_j) * p_j \quad (4)$$

The function $cost_m$ estimates the amount of data generated in the *map* stage. For each *emit* statement in λ_m, the size of the key-value pair emitted is multiplied by the probability that the *emit* statement will execute (p_i). The values are then summed to get the expected size of the output record. The total amount of data emitted during the map stage equals to the product of expected record size and the number of times λ_m is executed (N). The cost function for a *reduce* stage, $cost_r$, is defined similarly, except that λ_r produces only a single value and the cost is adjusted based on whether λ_r is commutative and associative. The function ϵ returns 1 if these properties hold; otherwise, it returns W_{csg}. The cost function for *join* operations, $cost_j$, is defined over: the number of elements in the two input datasets (N_1 and N_2), the selectivity of the join predicate (p_j), and the size of the output record. W_m, W_r and W_j are the weights assigned to the map, reduce and join operations. W_{csg} is the penalty for a non-commutative associative reduction. In our experiments, we used the values 1, 2, 2 and 50 for these weights, respectively based on our empirical studies.

To estimate the cost of a program summary, we simply sum the cost of each individual operation. The first operator in the pipeline takes symbolic variables $N_{0..i}$ as the number of records. For each subsequent stage, we use the number of key-value pairs generated by the current stage, expressed as a function over $N_{0..i}$:

$$cost_{mr}([(op_1, \lambda_1), (op_2, \lambda_2), \ldots], N_{0..i}) = cost_{op1}(\lambda_1, N, W) + cost_{mr}([(op_2, \lambda_2), \ldots], count(\lambda_1, N_{0..i}))$$

The function *count* returns the number of key-value pairs generated by a given stage. For *map* stages, this equals $\sum_{i=1}^{|emits|} p_i$; for *reduce* stages, it equals the number of unique key values on which the reducer was executed; for joins, it equals $N_1 * N_2 * p_j$.

5.2 Dynamic Cost Estimation

The cost model computes the cost of a program summary as a function of input data size N. We use this cost model to compare the synthesized summaries both statically and dynamically. First, calling `findSummary` returns a list of verified summaries that were found. Casper then uses the cost model to prune summaries when a less costly one exists in the list. Not all summaries can be compared that way, however, since they could depend on the value distribution of the input data or how frequently a conditional evaluates to true, as shown in the candidates for grammar G_3's λ_{m1} in Figure 6.

In such cases, Casper generates code for all remaining summaries that have been validated, and it uses a runtime monitoring module to evaluate their costs dynamically when the generated program executes. As the program executes, the runtime module samples values from the input dataset (Casper currently uses first-k values sampling, although different sampling method may be used). It then uses the samples to estimate the probabilities of conditionals by counting the number of data elements in the sample for which the conditional will evaluate to true. Similarly, it counts the number of unique data values that are emitted as keys. These estimates are inserted into Eqn 2 and Eqn 3 for each program summary to get comparable cost values. Finally, the summary with the lowest cost is executed at runtime. Hence, if the generated program is executed over different data distributions, it will run different implementations, as illustrated in §7.4.

6 IMPLEMENTATION

We implemented Casper using the Polyglot framework [37] to parse Java code into an abstract syntax tree (AST). Casper traverses the program AST to identify candidate code fragments, performs program analysis, and generates target code. We now describe the Java features supported by our compiler front-end. We also discuss how Casper identifies code fragments for translation and generates executable code from the verified program summary.

6.1 Supported Language Features

To translate a code fragment, Casper must first successfully generate verification conditions for that fragment (as explained in §3.3). Casper can currently do this for basic Java statements, conditionals, functions, user-defined types, and loops.

Basic Types. Casper supports all basic Java arithmetic, logical, and bit-wise operators. It can also process reads and writes into primitive arrays and common Java Collection interfaces, such as `java.util.{List, Set,Map}`. Casper can be extended to support other data structures, such as `Stack` or `Queue`.

User-defined Types. Casper traverses the program AST to find declarations of all types that were used in the code fragment being translated. It then dynamically translates and adds these types to the IR as `structs`, as shown in Appendix B.

Loops. Casper computes VCs for different types of loops (`for`, `while`, `do`), including those with loop-carried dependencies [1], after applying classical transformations [1] to convert loops into the `while(true){...}` format.

Methods. Casper handles methods by inlining their bodies. Polymorphic methods can be supported similarly by inlining different versions with conditionals that check the type of the host object at runtime. Recursive methods and methods with side-effects are not currently supported because they are unlikely to gain any speedup by being translated to MapReduce.

External Library Methods. Casper supports common library methods from standard Java libraries (e.g., `java.lang.Math` methods) by modeling their semantics explicitly using the IR. Users can similarly provide models for other methods that Casper currently does not support. [3]

6.2 Code Fragment Identification

Casper traverses the input AST to identify code fragments that are amenable for translation by searching for loops that iterate one or more data structures (e.g., a list or an array). We target loops since they are most likely to benefit from translation to MapReduce. We have kept our loop selection criteria lenient to avoid false negatives.

6.3 Code Generation

Once an identified code fragment is translated, Casper replaces the original code fragment with the translated MapReduce code. It also generates "glue" code to merge the generated code into the rest of the program. This includes creating a `SparkContext` (or an `ExecutionEnvironment` for Flink), converting data into `RDDs` (or Flink's `DataSets`), broadcasting required variables, etc. Since some API calls (such as Spark's `reduceByKey`) are not defined for non-commutative associative transformer functions, Casper uses these API calls only if the generated code is indeed commutative and associative (otherwise, Casper uses safe, albiet less efficient, transformations, such as `groupByKey`). Finally, Casper also generates code for sampling input data and dynamic switching, as discussed in §5.2. Appendix C presents a subset of code-generation rules for the Spark API.

[3] We provide examples of library function and type models in Appendix B.

Suite	# Translated	Mean Speedup	Max Speedup
Phoenix	7 / 11	14.8x	32x
Ariths	11 / 11	12.6x	18.1x
Stats	18 / 19	18.2x	28.9x
Bigλ	6 / 8	21.5x	32.2x
Fiji	23 / 35	18.1x	24.3x
TPC-H	10 / 10	31.8x	48.2x
Iterative	7 / 7	18.4x	28.8x

Table 1: Number of code fragments translated by Casper and their mean and max speedups compared to sequential implementations.

7 EVALUATION

In this section, we present a comprehensive evaluation of Casper on a number of dimensions, including its ability to: (1) handle diverse and realistic workloads, (2) find efficient translations, (3) compile efficiently, and (4) extend to support other IRs and cost-models in the future. All experiments were conducted on an AWS cluster of 10 m3.2xlarge instances (1 master node, 9 core nodes), where each node contains an Intel Xeon 2.5 GHz processor with 8 vCPUs, 30 GB of memory, and 160 GB of SSD storage. We used the latest versions of all frameworks available on AWS: Spark 2.3.0, Hadoop 2.8.3, and Flink 1.4.0. The data files for all experiments were stored on HDFS.

7.1 Feasibility Analysis

We first assess Casper's ability to handle a variety of data-processing applications. Specifically, we determine whether: (1) Casper can generate verification conditions for a syntactically diverse set of programs, (2) our IR can express summaries for a broad range of data-processing workloads, and (3) Casper's ability to find such summaries. To this end, we used Casper to optimize a set of 55 diverse benchmarks from real-world applications that contained a total of 101 translatable code fragments.

Basic Applications. For benchmarking, we assembled a set of small applications from prior work and online repositories. These applications, summarized below, contain a diverse set of code patterns commonly found in data-processing workloads (e.g., aggregations, selections, grouping, etc), as follows:

- *Bigλ* [44] consists of several data analysis tasks such as *sentiment analysis*, *database operations* (e.g., selection and projection), and *Wikipedia log processing*. Since Bigλ generates code from input-output examples rather than from an actual implementation, we recruited computer science graduate students in our department to implement a representative subset of the benchmarks from their textual descriptions. This resulted in 211 lines of code across 7 files.
- *Stats* is a set of benchmarks Casper automatically extracted from an online repository for the statistical analysis of data [32]. Examples include *Covariance*, *Standard Error* and *Hadamard Product*. The repository contains 1162 lines of code across 12 Java files, mostly consisting of vector and matrix operations.
- *Ariths* is a set of simple mathematical functions and aggregations collected from prior work [14, 19, 23, 40]. Examples include *Min*, *Max*, *Delta*, and *Conditional Sum*. The suite contains 245 lines of code than span 11 files.

Across the 3 suites, Casper identified 38 code fragments, of which 35 were successfully translated. One code-fragment that

Casper failed to translate used a variable-sized kernel to convolve a matrix; two others required broadcasting data values to many reducers during the map stage, but such mappers are currently inexpressible in our IR due to the absence of loops.

Traditional Data-Processing Benchmarks. Next, we used Casper to translate a set of well-known, data-processing benchmarks that resemble real-world workloads:

- We manually implemented Q1, Q6, Q15 and Q17 from the *TPC-H* benchmark using sequential Java and used Casper to translate the Java implementations to MapReduce. The selected queries cover many SQL features, such as aggregations, joins and nested queries.
- *Phoenix* [39] is a collection of standard MapReduce problems—such as *3D Histogram*, *Linear Regression*, *KMeans*, etc.—used in prior work [38]. Since the original sequential implementations were written in C, we used the sequential Java translations of the benchmarks from prior work in our experiments. The suite consists of 440 lines of code across 7 files.
- *Iterative* represents two popular iterative algorithms that we manually implemented into sequential versions: *PageRank* and *Logistic Regression Based Classification*.

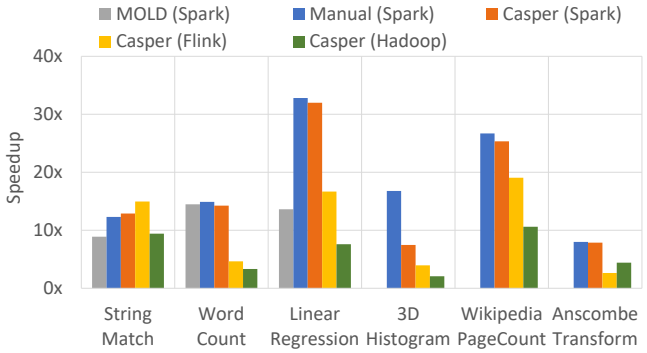
Casper successfully translated all 4 TPC-H queries and both iterative algorithms. It successfully translated 7 of 11 from the Phoenix suite. Three of the 4 failures were due to the IR's lack of support for loops inside transformer functions. One benchmark failed to synthesize within 90 minutes, causing Casper to time out.

Real-World Applications. Fiji [24] is a popular distribution of the ImageJ [27] library for scientific image analysis. We ran Casper on the source code of four Fiji packages (aka plugins). *NL Means* is a plugin for denoising images via the non-local-means algorithm [13] with optimizations [20]. *Red To Magenta* transforms images by changing red pixels to magenta. *Temporal Median* is a probabilistic filter for extracting foreground objects from a sequence of images. *Trails* averages pixel intensities over a time window in an image sequence. These packages, authored by different developers, contain 1411 lines of code that span 5 files. Of the 35 candidate code fragments identified across all 4 packages, Casper successfully optimized 23. Three of the failures were caused by the use of unsupported types or methods from the ImageJ library since we did not model them using the IR, and the search timed out for the remaining 9.

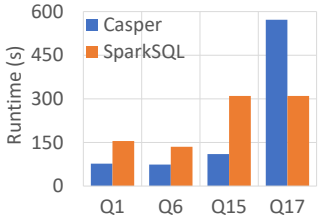
Table 1 summarizes the results of our feasibility analysis. Of the 101 individual code fragments identified by the compiler across all benchmarks, Casper translated 82. We manually inspected all code files to ensure that Casper's code fragment identifier missed no translatable code fragments. Overall, the benchmarks form a syntactically diverse set of applications.[4]

Because MOLD is not publicly available, we obtained the generated code from the MOLD authors for the benchmarks used in its evaluation [38]. Of the 7 Phoenix benchmarks, MOLD could not translate 2 (*PCA* and *KMeans*). Another 2 (*Histogram* and *Matrix Multiplication*) generated semantically correct translations that worked well for multi-core execution but failed to execute on the cluster because they ran out of memory. For the remaining 3 benchmarks (*Word Count*, *String Match* and *Linear Regression*), MOLD generated working implementations. In contrast, Casper translated 4 of the 7 Phoenix benchmarks. For *PCA* and *KMeans*, Casper translated and successfully executed a subset of all the loops found, while translation failed for the other loops and the *Matrix Multiplication benchmark* for reasons explained above.

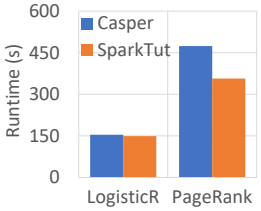
[4]We summarize the syntactic features of the code fragments in Appendix E.1.



(a) Casper achieves speedup competitive with manual translations



(b) TPC-H benchmarks



(c) Iterative algorithms

Figure 7: A runtime comparison of Casper-generated implementations against reference implementations.

7.2 Performance of the Translated Benchmarks

Casper helps an application leverage the optimization and parallelization provided by MapReduce implementations by translating their code. Therefore, in this section, we examine the quality of the translations Casper produced by comparing their performance to that of reference distributed implementations.

We used Casper to translate summaries for these benchmarks to three popular implementations of the MapReduce programming model: Hadoop, Spark, and Flink. The translated Spark implementations, along with their original sequential implementations, were executed on three synthetic datasets of sizes 25GB, 50GB, and 75GB. Overall, the Spark implementations Casper generated are 15.6× faster on average than their sequential counterparts, with a max improvement of up to 48.2×. Table 1 shows the mean and max speedup observed for each benchmark suite using Spark on a 75GB dataset. We also executed the Hadoop and Flink implementations generated by Casper for a subset of 10 benchmarks, some of which are shown in Figure 7(a). The average speedups observed (over the 10 benchmarks) by these implementations are 6.4× and 10.8×, respectively. These results show that Casper can effectively improve the performance of applications by an order of magnitude by retargeting critical code fragments for execution on MapReduce frameworks.

Figure 7(a) plots the speedup achieved by the MOLD-generated implementations for *String Match*, *Word Count*, and *Linear Regression*. The Spark translations MOLD generated for these benchmarks performed 12.3× faster on average than the sequential versions. The solutions generated by Casper for *String Match* and *Linear Regression* were faster than those generated by MOLD by 1.44× and 2.34×, respectively. For *String Match*, Casper found an efficient encoding to reduce the amount of data emitted in the map stage (see §7.4), whereas MOLD emitted a key-value pair for every word in the dataset. Furthermore, MOLD used separate MapReduce operations to compute the result for each keyword; Casper computed the result for all keywords in the same set of operations. For *Linear Regression*, MOLD discovered the same overall algorithm as Casper except its implementation zipped the input RDD with its index as a pre-processing step, almost doubling the size of input data and hence the amount of time spent in data transfers.

For the *Ariths*, *Stats*, *Bigλ*, and *Fiji* benchmarks, we recruited Spark developers through UpWork.com to manually rewrite the benchmarks since reference distributed implementations were not available.[5] Figure 7(a) compares the performance of (a subset of) Casper-generated implementations to handwritten benchmark implementations over the 75GB dataset. Results show that the Casper-generated implementations perform competitively, even with those manually written by developers. In fact, of the 42 hand-translated benchmark implementations, 24 used the same high-level algorithm as the one generated by Casper, and most of the remaining ones differ by using framework-specific methods instead of an explicit map/reduce (e.g., using Spark's built-in filter, sum, and count methods). However, these variations did not cause a noticeable performance difference. One interesting case was the 3D Histogram benchmark, where the developer exploited knowledge about the data to improve runtime performance. Specifically, the developer recognized that since RGB values always range between 0-255, the histogram data structure would never exceed 768 values. Therefore, the developer used Spark's more efficient *aggregate* operator to implement the solution. Casper, not knowing that pixel RGB values are bounded, assumed that the number of keys could grow to be arbitrarily large and that using the aggregate operator could cause out-of-memory errors, hence it generated a single stage map and reduce instead.

For *PageRank* and *Logistical Regression*, we compared Casper against the implementations found in the Spark Tutorials [46] (see Figure 7(c)). The reference PageRank implementation was 1.3× faster than the one Casper generated on a dataset of about 2.25 billion graph edges and running 10 iterations. This is because Casper currently does not generate any `cache()` statements, nor does it co-partition data. Deciding when to cache can lead to further performance gains. Prior work [12] suggested heuristics for inserting such statements into Spark algorithms that could be integrated into Casper's code generator to improve performance for iterative workloads. For *Logistical Regression*, we found no noticeable difference in performance.

For TPC-H queries, we compared the performance of Spark code generated by Casper against SparkSQL's implementation. Figure 7(b) plots the results of this experiment. For Q1, Q6 and Q15, Casper implementations executed 2×, 1.8× and 2.8× faster, respectively, than SparkSQL on a scale factor of 100. For Q1 and Q6, we attribute this to the extra data shuffling performed by the SparkSQL query plan. In Q15, SparkSQL's query plan scanned the *lineitem* relation twice, whereas Casper's implementation did so only once, resulting in worse runtime performance. For Q17, SparkSQL executed 1.7× faster because it performed better scheduling of the query operators than the Casper-generated implementation. In sum, results show that the Casper-generated implementations the TPC-H benchmarks have comparable performance to those implemented directly using the MapReduce frameworks. Yet, developers need not learn different MapReduce APIs by using Casper.

[5] Appendix E.2 describes the hiring criteria.

Source	Mean Time (s)	Mean LOC	Mean # Op	Mean TP Failures
Phoenix	944	13.8 (13.1)	2.3 (2.1)	0.35
Ariths	223	9.4 (7.6)	1.6 (1.2)	4
Stats	351	7.6 (5.8)	1.8 (1.8)	0.6
Bigλ	112	13.6 (10)	1.8 (2.0)	0.4
Fiji	1294	7.2 (7.4)	1.4 (1.6)	0.1
TPC-H	476	5.9 (n/a)	7.25 (n/a)	0
Iterative	788	3.3 (3.7)	4.5 (3.5)	2

Table 2: Summary of Casper's compilation performance. Values for the reference implementations are shown in parentheses.

7.3 Compilation Performance

We next evaluate Casper's compilation performance. We discuss the time taken by Casper to compile the benchmarks, the effectiveness of Casper's two-phase verification strategy, the quality of the generated code, and incremental grammar generation.

7.3.1 Compile Time. On average, Casper took 11.4 minutes to compile a single code fragment. However, the median compile time for a single benchmark was only 2.1 minutes: for some benchmarks, the synthesizer discovered a low-cost solution during the first few grammar classes, letting Casper terminate search early. Table 2 shows the mean compilation time for a single benchmark by suite.

7.3.2 Two-Phase Verification. In our experiments, the candidate summary generator produced at least one incorrect solution for 13 out of the 101 successfully translated code-fragments. The synthesizer proposed a total of 76 incorrect summaries across all benchmarks. Table 2 lists the average number of times the theorem prover rejected a solution for each benchmark suite. As an example, the *Delta* benchmark computes the difference between the largest and smallest values in the dataset. It incurred 7 rounds of interaction with the theorem prover before the candidate generator found a correct solution due to errors from bounded model checking (discussed in §4.1).

7.3.3 Generated Code Quality. In addition to measuring the runtime performance of Casper-generated implementations, we manually inspected the code generated by Casper and compared it to the reference implementations for two code quality metrics: lines of code (LOC) and the number of MapReduce operations used. Table 2 shows the results of our analysis. Implementations generated by Casper were comparable and did not use more MapReduce operations or LOC than were necessary to implement a given task.

Benchmark	With Incr. Grammar	Without Incr. Grammar
WordCount	2	827
StringMatch	24	416
Linear Regression	1	94
3D Histogram	5	118
YelpKids	1	286
Wikipedia PageCount	1	568
Covariance	5	11
Hadamard Product	1	484
Database Select	1	397
Anscombe Transform	2	78

Table 3: With incremental grammar generation, Casper produces far less redundant summaries.

Note that the LOC pertain to individual code fragments, not entire benchmarks.

7.3.4 Incremental Grammar Generation. We also measured the effectiveness of incremental grammar generation in optimizing search. To measure its impact on compilation time, we used Casper to translate benchmarks without incremental grammar generation and compared the results. The synthesizer was allowed to run for 90 minutes, after which it was manually killed. The results of this experiment are summarized in Table 3. Exhaustively searching the entire search space produced hundreds of more expensive solutions. The cost of searching, verifying, and sorting all these superfluous solutions dramatically increased overall synthesis time. In fact, Casper timed out for every benchmark in that set (which represents a slowdown by at least one order of magnitude).

7.4 Dynamic Tuning

The final set of experiments evaluated the runtime monitor module and whether the dynamic cost model could select the correct implementations. As explained in §5.2, the performance of some solutions depends on the distribution of the input data. Therefore, we used Casper to generate different implementations for the StringMatch benchmark (Figure 8(a)). Figure 8(d) shows three (out of 400+) correct candidate solutions, with their respective costs based on the formula described in §5.1 and the following values for data-type sizes: 40 bytes for String, 10 bytes for Boolean and 28 bytes for a tuple of Boolean Objects. Solution (a) can be disqualified at compile time because it will have a higher cost than solution (b) for all possible data distributions. However, the cost of solutions (b) and (c) cannot be statically compared due to the unknowns p_1 and p_2 (the respective probabilities that the conditionals will evaluate to true and a key-value pair will be emitted). The values of p_1 and p_2 depend on the input data, i.e., how often the keywords appear in the text, and thus can be determined only dynamically at run-time.

Casper handles this by generating a runtime monitor in the output code. The monitor samples the input data (first 5000 values) in each execution to estimate values for unknown variables in the cost formulas. The estimated values are then plugged back into the original cost functions (Eqn 2 and 3), and the solution with the lowest cost is then executed.

We executed solutions (b) and (c) on three 75GB datasets with different amounts of skew: one with no matching words (i.e., (c) emits nothing), one with 50% matching words (i.e., (c) emits a key-value pair for half of the words in the dataset), and one with 95% matching words (i.e., (c) emits a key-value pair for 95% of the words in the dataset). Figure 8(c) shows the dynamically computed final cost of solution (c) using p_1 and p_2 estimates calculated using sampling. Figure 8(b) shows the actual performance of the two solutions. For datasets with very high skew, it is beneficial to use solution (b) due to the smaller size of its key-value pair emit. Otherwise, solution (c) performs better. Casper, with the help of the dynamic input from the runtime monitor, makes this inference and selects the correct solution for all three datasets.

Dynamic cost estimation is particularly impactful in workloads with multiple join operations. The size of each relation participating in the join in addition to the selectivity of the join predicate dictate the most cost-efficient join ordering. To demonstrate this, we translated a simple query based on the TPC-H schema that implements a 3-way join between the *part*, *supplier*, and *partsupplier* relations. Query parameters are the name of the supplier and the customer_id, and outputs are the customer's name, email address, and the sum of discount savings across all sales between the two parties. We executed this query over two parameter configurations: one where the cardinality of *join(sales, supplier)* was much greater than *join(sales, customer)* and one where it was much smaller. On compilation, Casper generated two semantically equivalent implementations for the query with different join orderings; which one to use depends on the cardinality of the input data. Upon execution, the Casper runtime estimated the cost of each join ordering and executed the faster solution for both configurations, showing the effectiveness of our dynamic tuning approach. We discuss the accuracy of the cost-functions we used in Appendix E.3

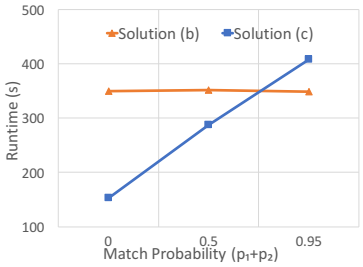
7.5 System Extensibility

The translation techniques Casper uses are not coupled to our IR or the target frameworks used. To demonstrate Casper's extensibility, we implemented the Fold-IR in prior work [22] in our system. Adding the `fold` construct to our IR required just 5 lines of code. An additional 43 lines of code were required to implement compilation of the fold operator to Dafny for verification of synthesized summaries. Since operations such as `min`, `max`, `set.insert` and `list.append` were already available in our IR, hence no extra work was needed. We did not implement any incremental grammar exploration for Fold-IR and used a constant bound to restrict the maximum size of summary expressions. With this minimal amount of work, we synthesized summaries expressed in Fold-IR for all benchmarks in the *Ariths* set. We believe it should be easy to extend Casper's code generator to output the same code as in the original work.

We also explored using WeldIR [35] to express summaries. Although WeldIR is an excellent abstraction for data-processing workloads, we believe it is not suited for synthesis because it is too low-level. However, since both our IR and Fold-IR are conceptually subsets of WeldIR, summaries expressed using them can be translated to Weld through simple rewrite rules. To demonstrate, we successfully translated the summary for TPC-H Q6 expressed in our IR to Weld and used the Weld compiler to produce vectorized, multi-threaded code.

```
1  key1_found = false
2  key2_found = false
3  for word in text:
4    if word == key1:
5      key1_found = true;
6    if word == key2:
7      key2_found = true;
```

(a) Sequential code for StringMatch



(b) Performance of solutions over datasets with different levels of skew

Dataset	Cost of Soln (c)	Optimal Solution
0% match	0	(c)
50% match	$75N$	(c)
95% match	$142.5N$	(b)

(c) Dynamic selection of optimal algorithm

	Solution	Static Cost
a	$output = reduceByKey(map(text, \lambda_m), \lambda_r)$ $\lambda_m : (word) \rightarrow \{(key1, word = key1), (key2, word = key2)\}$ $\lambda_r : (v_1, v_2) \rightarrow v_1 \vee v_2$	$\lambda_m : 2 * (40 + 10) * N$ $\lambda_r : 2 * 2 * 50 * N$ Total : $300N$
b	$output = reduce(map(text, \lambda_m), \lambda_r)$ $\lambda_m : (word) \rightarrow \{(word = key1,\ word = key2)\}$ $\lambda_r : (t_1, t_2) \rightarrow (t_1[0] \vee t_2[0],\ t_1[1] \vee t_2[1])$	$\lambda_m : 1 * 28 * N$ $\lambda_r : 2 * 28 * N$ Total : $84N$
c	$output = reduceByKey(map(text, \lambda_m), \lambda_r)$ $\lambda_m : (word) \rightarrow \{if\ (word = key1) : (key1, true), if\ (word = key2) : (key2, true)\}$ $\lambda_r : (v_1, v_2) \rightarrow v_1 \vee v_2$	$\lambda_m : (p_1 + p_2) * 50 * N$ $\lambda_r : (p_1 + p_2) * 2 * 50 * N$ Total : $150(p_1 + p_2)$

(d) Candidate solutions and their statically computed costs

Figure 8: StringMatch benchmark: Casper dynamically selects the optimal implementation for execution at runtime.

8 RELATED WORK

Implementations of MapReduce. MapReduce [21] is a popular programming model that has been implemented by various systems [6–8]. These systems provide their own high-level DSLs that developers 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 from low-level languages into high-level DSLs. MOLD [38], a source-to-source compiler, relies on syntax-directed rules to convert native Java programs to Apache Spark. Unlike MOLD, Casper translates based on program semantics and eliminates the need for rewrite rules, which are difficult to devise and brittle to code pattern changes. Many source-to-source compilers have been built similarly for other domains [34]. Unlike prior approaches in automatic parallelization [3, 10], Casper targets data parallel processing frameworks and translates only code fragments that are expressible in the IR for program summaries.

Synthesizing Efficient Implementations. Prior work used synthesis to generate efficient implementations and optimize programs. [44] synthesizes MapReduce solutions from user-provided input and output examples. QBS [15–17] and STNG [28] both use synthesis to convert low-level languages to specialized high-level DSLs for database applications and stencil computations, respectively. Casper takes inspiration from prior approaches by applying verified lifting to construct compilers. Unlike prior work, however, Casper: (1) addresses the problem of verifier failures and designs a grammar hierarchy to prune away non-performant summaries, (2) has a dynamic cost model and runtime monitoring module for adaptively choosing from different implementations at runtime.

Query Optimizers and IRs. Modern frameworks usually ship with sophisticated query optimizers [2, 9, 18, 29, 30] for generating efficient execution plans. However, these tools make users express their queries in the provided APIs. Our objective is orthogonal, i.e., to find the best way to express program semantics using the APIs provided by these tools. We essentially enable these tools to optimize code *not* written in their API. Furthermore, unlike our IR, most IRs meant to capture data-processing workloads [22, 35] are not designed with synthesis in mind. This makes it difficult both to find and verify programs expressed in them.

9 CONCLUSION

We presented Casper, a new compiler that identifies and converts sequential Java code fragments into MapReduce frameworks. Rather than defining pattern-matching rules to search for convertible code fragments, Casper instead automatically discovers high-level summaries of each input code fragment using program synthesis and retargets the found summary to the framework's API. Our experiments show that Casper can convert a wide variety of benchmarks from both prior work and real-world applications and can generate code for three different MapReduce frameworks. The generated code performs up to 48.2× faster compared to the original implementation, and is competitive with translations done manually by developers.

10 ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation through grants IIS-1546083, IIS-1651489, OAC-1739419, and CNS-1563788; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; the Intel-NSF CAPA center, and gifts from Adobe, Amazon, and Google.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Alexander Alexandrov, Asterios Katsifodimos, Georgi Krastev, and Volker Markl. 2016. Implicit Parallelism Through Deep Language Embedding. *SIGMOD Rec.* 45, 1 (June 2016), 51–58.

[3] Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Monica S. Lam, and Chau-Wen Tseng. 1995. An Overview of the SUIF Compiler for Scalable Parallel Machines. In *PPSC*. 662–667.

[4] Apache Flink 2018. https://flink.apache.org/. (2018). Accessed on: 2018-04-09.

[5] Apache Hadoop 2018. http://hadoop.apache.org. (2018). Accessed on: 2018-04-09.

[6] Apache Hive 2018. http://hive.apache.org. (2018). Accessed on: 2018-04-09.

[7] Apache Pig 2018. https://pig.apache.org/. (2018). Accessed on: 2018-04-09.

[8] Apache Spark 2018. https://spark.apache.org. (2018). Accessed on: 2018-04-09.

[9] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1383–1394.

[10] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David A. Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. 1994. Automatic Detection of Parallelism: A grand challenge for high performance computing. *IEEE P&DT* 2, 3 (1994), 37–47.

[11] Rastislav Bodík and Barbara Jobstmann. 2013. Algorithmic program synthesis: introduction. *International Journal on Software Tools for Technology Transfer* 15 (2013), 397–411.

[12] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1425–1436.

[13] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. 2005. A Non-Local Algorithm for Image Denoising. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '05)*. IEEE Computer Society, Washington, DC, USA, 60–65.

[14] Yu-Fang Chen, Lei Song, and Zhilin Wu. 2016. The Commutativity Problem of the MapReduce Framework: A Transducer-based Approach. *CoRR* abs/1605.01497 (2016).

[15] Alvin Cheung, Samuel Madden, Armando Solar-Lezama, Owen Arden, and Andrew C. Myers. 2014. Using Program Analysis to Improve Database Applications. *IEEE Data Eng. Bull.* 37, 1 (2014), 48–59.

[16] Alvin Cheung and Armando Solar-Lezama. 2016. Computer-Assisted Query Formulation. *Foundations and Trends in Programming Languages* 3, 1 (2016), 1–94.

[17] Alvin Cheung, Armando Solar-Lezama, and 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, 3–14.

[18] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. 2014. Tupleware: Redefining Modern Analytics. *CoRR* abs/1406.6667 (2014).

[19] Przemyslaw Daca, Thomas A. Henzinger, and Andrey Kupriyanov. 2016. Array Folds Logic. *CoRR* abs/1603.06850 (2016).

[20] Jerome Darbon, Alexandre Cunha, Tony F. Chan, Stanley Osher, and Grant J. Jensen. 2008. Fast nonlocal filtering applied to electron cryomicroscopy.. In *ISBI*. IEEE, 1331–1334.

[21] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113.

[22] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. 2016. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1781–1796.

[23] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. 2017. Gradual Synthesis for Static Parallelization of Single-pass Array-processing Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 572–585.

[24] Fiji: ImageJ 2018. https://github.com/fiji. (2018). Accessed on: 2018-04-09.

[25] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '10)*. ACM, New York, NY, USA, 13–24.

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

[27] ImageJ 2018. https://imagej.net/Welcome. (2018). Accessed on: 2018-04-09.

[28] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. *SIGPLAN Not.* 51, 6 (June 2016), 711–726.

[29] Alfons Kemper, Thomas Neumann, Florian Funke, Viktor Leis, and Henrik Mühe. 2012. HyPer: Adapting Columnar Main-Memory Data Management for Transactional AND Query Processing. *IEEE Data Eng. Bull.* 35, 1 (2012), 46–51.

[30] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-level Language. *Proc. VLDB Endow.* 7, 10 (June 2014), 853–864.

[31] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

[32] MagPie Analysis Repository 2018. https://github.com/thisMagpie/Analysis. (2018). Accessed on: 2018-04-09.

[33] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. 2006. Verification Condition Generation via Theorem Proving. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'06)*. Springer-Verlag, Berlin, Heidelberg, 362–376.

[34] Cedric Nugteren and 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, 1–10.

[35] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. 2017. Weld: A Common Runtime for High Performance Data Analytics. (January 2017).

[36] Spiros Papadimitriou and Jimeng Sun. 2008. DisCo: Distributed Co-clustering with Map-Reduce: A Case Study Towards Petabyte-Scale End-to-End Mining. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining (ICDM '08)*. IEEE Computer Society, Washington, DC, USA, 512–521.

[37] Polyglot 2018. http://www.cs.cornell.edu/Projects/polyglot/. (2018). Accessed on: 2018-04-09.

[38] Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and 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, 909–927.

[39] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and 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, 13–24.

[40] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing User-defined Aggregations Using Symbolic Execution. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 153–167.

[41] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD '79)*. ACM, New York, NY, USA, 23–34.

[42] Sketch 2018. https://people.csail.mit.edu/asolar/. (2018). Accessed on: 2018-04-09.

[43] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69.

[44] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. *SIGPLAN Not.* 51, 6 (June 2016), 326–340.

[45] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav.

[46] Spark GitHub Repository 2018. https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples. (2018). Accessed on: 2018-01-20.

[47] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA.

A PROOF SKETCH FOR SOUNDNESS AND COMPLETENESS

Here, we first formalize the definitions of soundness and completeness, and then we present a proof sketch to show that Casper's synthesis algorithm for program summaries has these properties. We use terms and acronyms defined in the paper without explaining them again here.

Definition 1. *(Soundness of Search)* An algorithm for generating program summaries is sound if and only if, for all program summary ps and loop invariants $inv_1, \ldots, inv_n$ generated by the algorithm, the verification conditions hold over all possible program states after we execute the input code fragment P. In other words, $\forall \sigma.\ VC(P, ps, inv_1, \ldots, inv_n, \sigma)$.

Definition 2. *(Completeness of Search)* An algorithm for generating program summaries is complete if and only if when there exists $ps, inv_1, \ldots, inv_n \in G$, then $\forall \sigma.\ VC(P, ps, inv_1, \ldots, inv_n, \sigma)) \rightarrow (\Delta \neq \emptyset)$. Here, G is the search space traversed, P is the input code fragment, VC is the set of verification conditions, and Δ is the set of sound summaries found by the algorithm. In other words, the algorithm will never fail to find a correct program summary as long as one exists in the search space.

Proof of Soundness. The soundness guarantee for Casper's synthesis algorithm is derived from the soundness guarantees offered by Hoare-style verification conditions. The proof is constructed using a *loop-invariant*, namely, a statement that is true immediately before and after each loop execution. Hoare logic dictates that in order to prove correctness of a given postcondition (i.e., program summary) for a given loop, we must prove the following holds over all possible program states:

(1) The invariant is true before the loop.
(2) Each iteration of the loop maintains the invariant.
(3) Once the loop has terminated, the invariant implies the postcondition.

This is essentially an inductive proof. The first two constraints prevent Casper from finding a loop invariant strong enough to imply an incorrect program summary. Our correctness guarantee is, of course, subject to the correct implementation of our VC generation module and of the theorem prover we use (Dafny). Establishing that the summary is a correct postcondition is sufficient to establish that it is a correct translation. This is so because summaries in our IR must describe the final value of *all* output variables (i.e., variables that were modified) as a function over the inputs (see Figure 3).

Proof of Completeness. To understand that Casper's algorithm is complete with respect to the search space, we first show that that the algorithm always terminates. Recall that we use recursive bounds to finitize the number of solutions expressible by our IR's grammar. As explained in §4.1, we prevent the same solution from being regenerated, thus ensuring forward progress in search. These two facts imply that our algorithm always terminates. There are only two possible exit points for the `while(true)` loop in our algorithm: line 24 and line 21 of Figure 5. The first is only reached once the entire search space has been exhausted. The second implies that a solution is successfully returned as Δ is not empty. It is important to note that our search algorithm is complete only for *verifiably correct* summaries. If a correct summary exists in the search space but cannot be proven correct using the available automated theorem prover, it will not be returned. Therefore, the completeness of the algorithm is modulo the completeness of the theorem prover.

B INTERMEDIATE REPRESENTATION SPECIFICATION

Here, we list the full set of types available in our IR and provide examples to demonstrate how they may be used to express models for library methods and types.

Primitive Data Types	
Scalars	`bool`, `int`, `float`, `string`, `char`, ...
Structures	`class(id:Type, id2:Type2, ..)`
List	`list(Type)`
Array	`array(dimensions, Type)`
Functions	`name(arg1:Type1, ...) : Type -> Body`
Conditionals	`if` *cond* `then` e_1 `else` e_2
Synthesis Construct	`choose(`e_1`,` e_2`, ... ,` e_n`)`

Built-in operations	
Arithmetic	+, −, ∗, /, %, ...
Bitwise	<<, >>, &, ...
Relational	<, >, ≤, ≥, ...
Logical	&&, ∥, ==, ! =
List	`len, append, get, equals, concat, slice`
Array	`select, store`

To provide support for a datatype found in a Library, users must define the type of the object using our IR and annotate it with the fully qualified name, as follows:

```
@java.awt.Point
class Point(x:int, y:int)
```

Similarly, users may also provide support for library methods, for instance the following defines a model for the absolute value function:

```
@java.lang.Math.abs
abs(val: int) : int ->
  if val < 0 then val * -1 else val
```

Using the core IR described above, we implemented in Casper the *map*, *reduce* and *join* primitives used to synthesize summaries. We have also implemented commonly used methods from Java standard libraries such as `java.util.Math,String,Date` and other essential data-types, along with methods that were needed to translate the Fiji plugins.

The *choose* operator in the IR is a special construct that enables us to express a search space using the IR. The parameters to *choose* are one or more expressions of matching types. The synthesizer is then free to select any expression from the list of choices in order to satisfy the correctness specification.

C CODE GENERATION RULES

To generate target DSL code from the synthesized program summary, we implemented in Casper a set of translation rules that map the operators in our IR to the concrete syntax of the target DSL. Here, we list a subset of such code-generation rules for the Spark RDD API.

$$
\begin{aligned}
TR[\![\textbf{map}(l,\ \lambda_m : T \rightarrow list(Pair))]\!] &= \texttt{l.flatMapToPair(}[\![\lambda_m]\!]\texttt{);} \\
TR[\![\textbf{map}(l,\ \lambda_m : T \rightarrow list(U))]\!] &= \texttt{l.flatMap(}[\![\lambda_m]\!]\texttt{);} \\
TR[\![\textbf{map}(l,\ \lambda_m : T \rightarrow Pair)]\!] &= \texttt{l.mapToPair(}[\![\lambda_m]\!]\texttt{);} \\
TR[\![\textbf{map}(l,\ \lambda_m : T \rightarrow U)]\!] &= \texttt{l.map(}[\![\lambda_m]\!]\texttt{);} \\
TR[\![\textbf{reduce}(l : list(Pair),\ \lambda_r)]\!] &= \texttt{l.reduceByKey(}[\![\lambda_r]\!]\texttt{);} \\
TR[\![\textbf{reduce}(l : list(U),\ \lambda_r)]\!] &= \texttt{l.reduce(}[\![\lambda_r]\!]\texttt{);} \\
\hline
TR[\![\lambda_m(e) \rightarrow e_b]\!] &= \texttt{(e -> } [\![e_b]\!]\texttt{)} \\
TR[\![e_1 + e_2]\!] &= [\![e_1]\!] + [\![e_2]\!]
\end{aligned}
$$

The translation function TR takes as input an expression in our IR language and maps it to an equivalent expression in Spark. Since Spark provides multiple variations for the operators defined in our IR, such as *map*, we can select the appropriate variation by looking at the type information of the λ_m function used by *map*. For example, if λ_m returns a list of Pairs, we translate to `JavaRDD.flatMapToPair`. If it instead returns a list of a non-Pair type, we use the more general rule that translates *map* to `JavaRDD.flatMap`. Translation for the other expressions proceeds similarly.

D PROGRAM ANALYZER OUTPUTS

Here, we use TPC-H Query 6 to illustrate the outputs computed by Casper's program analyzer. Since the queries are originally in SQL, we have manually translated them to Java as follows:

```
1  double query6(List<LineItem> lineitem){}
2    List<LineItem> lineitem = new ArrayList<LineItem>();
3    Date dt1 = Util.df.parse("1993-01-01");
4    Date dt2 = Util.df.parse("1994-01-01");
5    double revenue = 0;
6    for (LineItem l : lineitem) {
7      if (
8        l.l_shipdate.after(dt1) &&
9        l.l_shipdate.before(dt2) &&
10       l.l_discount >= 0.05 &&
11       l.l_discount <= 0.07 &&
12       l.l_quantity < 24
13     )
14       revenue += (l.l_extendedprice * l.l_discount);
15   }
16   return revenue;
17 }
```

First, Casper's program analyzer normalizes the loop starting on Line 6 into an equivalent `while(true){..}` loop, and then traverses the loop to identify the set of input/output variables and operators used:

Program Analaysis Results	
Inputs Vars	`l: list(LineItem), dt1: Date, dt2: Date`
Output Vars	`revenue: double`
Constants	`[(24, int), (0.05, double), (0.07, double)]`
Operators	$+, -, *, \geq, \leq, <$
Methods	`Date.before, Date.after`

With this information, Casper generates verification conditions like those shown in Figure 4(b) for the row-wise mean benchmark. Next, the program analyzer defines a search space within which Casper searches for summaries and the needed loop-invariant. Since the full search space description is too large to show, we only show a small snippet below:

```
generator doubleExpr(val:LineItem, depth:int) : double ->
  if depth = 0 then
    choose(
      val.l_quantity,
      val.l_extendedprice,
      val.l_discount,
      0.05,
      0.07,
      24
    )
  else
    choose(
      doubleExpr(val, 0),
      doubleExpr(val, depth-1) + doubleExpr(val, depth-1),
      doubleExpr(val, depth-1) * doubleExpr(val, depth-1),
      doubleExpr(val, depth-1) / doubleExpr(val, depth-1)
    )
```

The `doubleExpr` is the part of the grammar used to construct expressions that evaluate to *double*. The `generator` keyword indicates that this is a special type of function, one that can select a different value from the `choose` operators on each invocation. The depth parameter controls how large the generated expression is allowed to grow. The `choose` construct is used to present a set of possible productions to the synthesizer. This grammar is tailored specifically to our implementation of TPC-H Query 6.

E SUPPLEMENTARY EXPERIMENTS

E.1 Benchmark Details

The benchmarks Casper extracted form a diverse and challenging problem set. As shown in the table below, they vary across programming style as well as the structure of their solutions.

Benchmark Properties	# Extracted	# Translated
Conditionals	26	19
User Defined Types	14	10
Nested Loops	40	22
Multiple Datasets	22	18
Multidim. Dataset	38	23

E.2 Developer Selection Criteria

To get reference Spark implementations for non-SQL benchmarks, we hired developers through the online freelancing platform UpWork.com. While hiring, we ensured all candidates met the following basic criteria:

(1) At least an undergraduate or equivalent degree in computer science.
(2) Minimum 500 hours of work logged at the platform.
(3) Minimum 4 star rating for previous projects (scale of 5).
(4) A portfolio of at least one or more successfully completed contracts using Spark.

Finally, applicants were required to answer three test questions regarding Spark API internals to bid on our contract.

E.3 Evaluating Cost Model Heuristics

We present here some experiments that measure whether Casper's cost model model can effectively identify efficient solutions during the search process.

Program	Emitted (MB)	Shuffled (MB)	Runtime (s)
WC 1	105k	30	254
WC 2	105k	58k	2627
SM 1	16	0.7	189
SM 2	90k	0.7	362

Table 4: The correlation of data shuffle and execution. (WC = WordCount, SM = StringMatch).

As discussed in §5.1, Casper uses a data-centric cost model. The cost model is based on the hypothesis that the amount of data generated and shuffled during the execution of a MapReduce program determines how fast the program executes. For our first experiment, we measured the correlation between the amount of data shuffled and the runtime of a benchmark to check the validity of the hypothesis. To do so, we compared the performance of two different Spark WordCount implementations: one that aggregates data locally before shuffling (WC 1) using combiners [21], and one that does not (WC 2). Although both implementations processed the same amount of input data, the former implementation significantly outperformed the latter, as the latter incurred the expensive overhead of moving data across the network to the nodes responsible for processing it. Table 4 shows the amount of data shuffled along with the corresponding runtimes for both implementations using the 75GB dataset. As shown, the implementation that used combiners to reduce data shuffling was almost an order of magnitude faster.

Next, we verified the second part of our hypotheses by measuring the correlation of the amount of data generated and the runtime of a benchmark. To do so, we compared two solutions for the StringMatch benchmark (sequential code shown in Figure 8(a)). The benchmark determines whether certain keywords exist in a large body of text. Both solutions use combiners to locally aggregate data before shuffling. However, one solution emits a key-value pair only when a matching word is found (SM 1), whereas the other always emits either `(key, true)` or `(key, false)` (SM 2). Since the data is locally aggregated, each node in the cluster only generates 2 records for shuffling (one for each keyword) regardless of how many records were emitted during the map phase. As shown in Table 4, the implementation that minimized the amount of data emitted in the map-phase executed almost twice as fast.

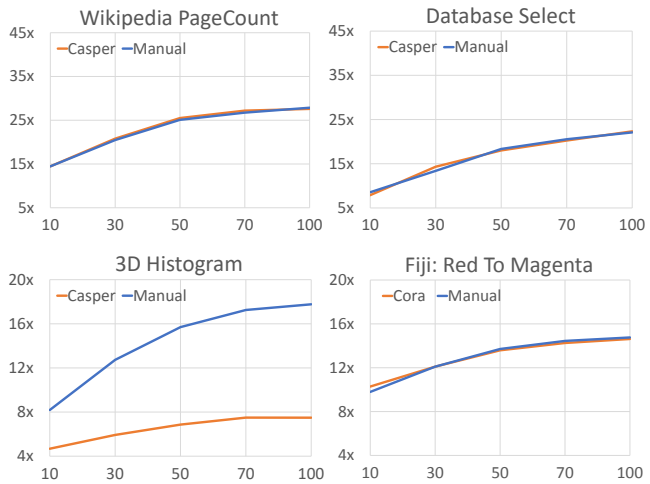


Figure 9: The top 2 benchmarks with the most performance along with the bottom 2. The x-axis plots the size of input data, while the y-axis plots the runtime speedup over sequential implementations.

In sum, the two experiments confirm that the heuristics used in our cost model are accurate indicators of runtime performance for MapReduce applications. We also demonstrated the need for a data-centric cost model; solutions that minimize data costs execute significantly faster than those that do not.

E.4 Evaluating Scalability of Generated Implementations

To observe how implementations generated by Casper scale, we executed our benchmarks on different amounts of data and measured the resulting speedups. As shown in Figure 9, the Casper-generated Spark implementations exhibited good data parallelism and showed a steady increase in speedups across all translated benchmarks as the input data size increased, until the cluster reached maximum utilization.