Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications

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

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

ABSTRACT

MapReduce is a popular programming paradigm for running large-scale data-intensive computation. Recently, many frameworks that implement that paradigm have been developed. To leverage such frameworks, however, developers need to familiarize with each framework's API and rewrite their code. We present Casper, a new tool that automatically translates sequential Java programs to the MapReduce paradigm. Rather than building a compiler by tediously designing pattern-matching rules to identify code fragments to translate from the input, Casper translates the input program in two steps: first, Casper uses *program synthesis* to identify input code fragments and search for a program summary (i.e., a functional specification) of each fragment. The summary is expressed using a high-level intermediate language resembling the MapReduce paradigm. Next, each found summary is verified to be semantically equivalent to the original using a theorem prover. Casper then generates executable code from the summary, using either the Hadoop, Spark, or Flink API. We have evaluated Casper by automatically converting real-world sequential Java benchmarks to MapReduce. The resulting benchmarks perform up to 32.2× faster compared to the original, and are all translated without designing any pattern-matching rules.

1 INTRODUCTION

MapReduce [14] is a popular paradigm for writing data-intensive applications, with highly efficient implementations developed since its introduction [2, 3, 6, 23]. All these frameworks expose an application programming interface (API) for the developer. While the concrete syntax differs slightly across the frameworks' APIs, they all require developers to organize their computation using `map` and `reduce` functions. Once the developer expresses her computation as a series of mappers and reducers, the resulting application will be able to leverage the optimizations provided by the framework.

While exposing optimization via an API cleanly separates the developer from the framework, this approach contains a major drawback: for legacy applications to leverage MapReduce frameworks, developers need to first reverse-engineer the specification of the application if it is not already provided, and subsequently reorganize their computation using mappers and reducers. Similarly, for novice programmers who are unfamiliar with the MapReduce paradigm, they need to first learn the different APIs provided by the frameworks and express their computation accordingly. Both require a significant amount of time and effort, and needless to say each code rewrite or algorithm reformulation is another opportunity to introduce bugs.

One way to alleviate the above issues is to build a compiler that can take code written in another paradigm (e.g., sequential code) and translate it into MapReduce. Classical compilers, like query optimizers, are constructed based on syntax-driven rules, i.e., the compilers are embedded with a number of rules that each recognizes different patterns in the input code (e.g., a sequential loop over lists), and translates the matched code fragment into the target (e.g., a single-stage MapReduce program). Designing such rules is highly nontrivial: they need to be correct, i.e., the translated program has the same semantics as the input, they need to use the appropriate API when generating the target programs, and expressive enough such that they can capture the wide variety of coding patterns that developers might use to express their computation. As a case in point, we are only aware of one such compiler that translates sequential Java programs into MapReduce [25], and the number of syntax-driven rules involved in that compiler and their complexities make it very hard to maintain and modify to generate code for different MapReduce frameworks.

This paper describes a new tool called Casper that translates sequential Java code into semantically equivalent MapReduce programs. Rather than relying on syntax-driven rules to recognize different code patterns, Casper is inspired by prior work on cost-based query optimization [28] which considers the compilation problem as dynamic search. However, given that the inputs are general-purpose programs, the space of possible target programs is much larger compared to query optimization. To address this issue, Casper leverages recent advances in program synthesis [8, 16] to search for MapReduce programs that a given input sequential Java code fragment can be rewritten to. Since the number of programs written in the concrete syntax of the target API is huge, Casper instead searches over the space of *program summaries* expressible using a high-level intermediate language (IR) that we designed. As we will discuss in §3.1, the IR is designed such that it can succinctly express computations written in the MapReduce paradigm, yet easy enough to translate to the actual concrete syntax of the target API.

To search for summaries, Casper first performs lightweight program analysis to describe the space of MapReduce programs that a given input code fragment *might* be equivalent to. The search space is described using our high-level IR. Casper uses an off-the-shelf program synthesizer to perform the search, but guided by an incremental search algorithm and our domain-specific cost model to speed up the process. After the synthesizer finds a candidate program summary, Casper then sends it over to a theorem prover to establish semantic equivalence between the candidate and the input code fragment. Once proved, Casper translates the summary to the concrete syntax of the target MapReduce framework and replaces

the original code fragment with the translated version such that it can be compiled and executed. As the performance of the translated program often depends on characteristics of the input data (e.g., skewness), Casper generates multiple semantically equivalent MapReduce programs for a given input, and produces a monitor module that chooses among different compiled implementations based on runtime statistics, with the code that collects statistics and switches among different implementations all 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 process is completely automatic. We have evaluated Casper using a number of benchmarks and real-world Java applications and have demonstrated both Casper's ability to translate the input program into MapReduce equivalents with significant resulting performance improvement.

In summary, our paper makes the following contributions:

- We propose a high-level intermediate representation (IR) language to express the semantics of sequential Java programs in the MapReduce paradigm. The language is succinct such that it can be easily translated to multiple MapReduce frameworks, yet expressive enough 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) Being a high-level language also enables us to perform various *semantic optimization* using our cost model, which would be difficult to apply using concrete syntax. (§5)
- We describe an efficient technique to search for program summaries expressed in the IR without needing any syntax-driven rewrite rules. Our technique is both sound and complete with respect to the input search space, and incrementally searches for summaries based on cost. It also systematically uses verification failures to prune the search space and uses a hierarchy of search grammars to speed up summary search. (§4.1)
- Since there are often multiple ways to express the same input as MapReduce programs, our technique can generate multiple semantically equivalent versions of the input, and automatically inserts code to collect statistics during program execution to evaluate among the different alternatives using a dynamic cost model. The most efficient implementation is then chosen during program execution by the runtime monitoring module. (§5.2)
- We implemented our methodology in Casper, a tool that converts sequential Java programs to three MapReduce frameworks: Spark, Hadoop, and Flink. We evaluated the feasibility and effectiveness of Casper by translating real-world benchmarks from 5 different suites from multiple domains. Casper can translate 65 out of 84 benchmarks; the translated benchmarks perform up to 32.2× faster when compared to the original, and are competitive even with manual implementations done by experts. (§7)

2 OVERVIEW

In this section we describe how we model the MapReduce programming paradigm in this paper, and demonstrate how Casper translates sequential code to MapReduce programs with an example.

2.1 MapReduce Operators

MapReduce organizes computation using two operators: *map* and *reduce*. In this paper we model the *map* operator as:

$$\boldsymbol{map} : (mset[\tau];\ \lambda_m) \longrightarrow mset[(\kappa; v)]$$
$$\boldsymbol{\lambda_m} : \tau \longrightarrow mset[(\kappa; v)]$$

where *map* takes as input a multiset of type τ and a unary transformer function λ_m that converts a value of type τ to a multiset of key-value pairs of types κ and v respectively. It then concurrently applies λ_m to every element in the multiset and returns the union of all multisets generated by λ_m.

$$\boldsymbol{reduce} : (mset[(\kappa; v)];\ \lambda_r) \longrightarrow mset[(\kappa; v)]$$
$$\boldsymbol{\lambda_r} : (v; v) \longrightarrow v$$

reduce takes as input a multiset of key-value pairs and a binary transformer function λ_r that combines two values of type v. It first groups all the 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 to a single value. The output of *reduce* is another multiset of key-value pairs, where each pair holds a unique key. When the transformer function λ_r has commutative and associative properties, the execution of *reduce* can be further optimized by concurrently applying λ_r to pairs of values within a key-group.

The goal of Casper is to translate a sequential code fragment to a MapReduce program, expressed using the two operators described above. The challenges are to (1) identify the correct sequence of operations to apply, and, (2) implement the corresponding transformer functions. We discuss how Casper accomplishes this next.

2.2 Translating Imperative Code to MapReduce

The input to Casper is Java code with loop nests that sequentially iterate over data, and Casper translates the input 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 [26].

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. To translate the input to a MapReduce program, imagine the code being annotated with a *program summary* that helps with the translation. The program summary describes how the output of the code fragment (in this case `m`) can be computed using a high-level intermediate representation (IR) of a series of *map* and *reduce* with transformer functions on the input data (i.e., `mat`), as shown on lines 1 to 6 of Figure 1(a). While the summary is not executable code, translating from that IR to the concrete syntax of the MapReduce framework is much easier than translating from the input sequential Java loop. This is shown in Figure 1(b) where the *map* and *reduce* primitives from our IR are translated to the appropriate Spark API calls.

But alas, the input code does not come with a program summary, and it must therefore be inferred. In Casper, this is done via program synthesis and verification as we explain next.

2.3 System Architecture

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

```
@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) {
  Spark.broadcast(cols);
  RDD m = mat.mapToPair(e -> 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: Translation of the row-wise mean benchmark to MapReduce (Spark) using Casper.

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, it prepares for each identified code fragment, (i) a *search space description* for the synthesizer to search for a valid program summary (§3.1), and (ii) *verification conditions* (VCs) (§3.2) to automatically ascertain the induced program summary is semantically equivalent to the input.

Next, the *summary generator* implements the synthesis and verification of program summaries expressed in our high-level IR (§3.3 and §4.1). To speed up the search, it partitions the search space such that it can be efficiently traversed using our incremental synthesis algorithm (§4.2).

Once a summary is inferred, the *code generator* translates it to different Framework's API to be executed. Currently, we have support for three MapReduce frameworks: Spark, Hadoop and Flink. Additionally, the code generator also generates code to collect data statistics during runtime in order to choose among different implementations to execute (§5.2).

3 SYNTHESIZING PROGRAM SUMMARIES

As discussed, Casper discovers a program summary for each code fragment before translation. Technically, a program summary is a *postcondition* of the input code that describes the program state after the code fragment is executed. Much research has been done on inferring postconditions, and Casper uses program synthesis to for this task. In this section, we explain (1) the IR used by Casper

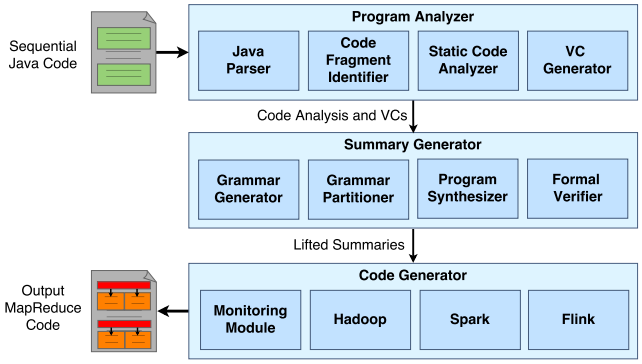


Figure 2: Casper system architecture diagram. Sequential code fragments (green) are translated to equivalent MapReduce tasks (orange).

to express postconditions, (2) how Casper verifies whether a postcondition is valid, and (3) the search algorithm Casper uses to find valid postconditions from the space of all possible postconditions expressible in the IR.

3.1 A High-level IR for Program Summaries

One approach to synthesize summaries is to directly search in programs written in the target framework's API. However, this does not work well; Spark alone offers over 80 high-level operators, which would result in a huge search space. Hence, we design a high-level IR to express the summaries. The goal of the IR is twofold: (1) summaries expressed using the IR need to be translatable to the target framework's API; (2) the synthesizer should be easily able to search through the possible space of summaries described using the IR. To address these goals, Casper uses a functional language containing the two MapReduce primitives similar to the `map` and `fold` operators in Haskell, as explained in §2.1. This allows us to capture a wide array of computations expressible using MapReduce, yet general enough to be translatable to different MapReduce frameworks and keeping the search problem tractable.

In addition, we impose program summaries to be in a stylized form shown in Figure 3 as *PS*. It states that for each output variable v, i.e., variables that are written to within the code fragment, the value of v is equal to a sequence of *map* and *reduce* operations applied to the input data, i.e., the array or collection being iterated inside the code fragment. While *map* and *reduce* operations are known functions in our IR (as stated in §2.1), the implementations of λ_m and λ_r depend on the code fragment being translated and are synthesized. We restrict the body of λ_m to be a sequence of *Emit* statements, where each *Emit* statement produces a single key-value pair. The body of λ_r is an expression that evaluates to a single value of the required type. The synthesizer will decide on both the correct number of *Emit* statements to use, what constitutes the key and value for each emitted key-value pair. Our IR also supports conditionals, allowing the *map* stage to model filter operations. In addition to primitive data-types, the language supports tuples as well. The output of each *reduce* expression is an associative array of key-value pairs, with the variable ID v_{id} of each output variable as the key mapping to its computed value. The variable ID for an output variable is a unique integer assigned by Casper during

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

$v \in$ *Output Variables* $\quad v_{id} \in$ *Variable ID*;
$op \in$ *Operators* $\quad f \in$ *Library Methods*

Figure 3: High-level IR for expressing program summaries (PSs).

compilation that allows extracting the computed values for each variable from the MapReduce program output.

The IR is used to represent the summary for each code fragment. In addition, Casper also uses the same language to describe the search space of summaries for the synthesizer in the form of a grammar, an instance of which is shown in Figure 3 for the code fragment shown in Figure 1(a). During synthesis, the synthesizer will traverse the grammar by expanding on each production rule, and checking whether any of the generated candidate constitutes a valid summary, to be explained in §3.2.

To generate the search space grammar, Casper analyzes the input code to extract properties such as the set of variables in scope, along with operators and library methods used to compute the result. Casper then builds a grammar out of the extracted operators and operators for the synthesizer to generate the implementations of λ_m and λ_r. To make synthesis tractable and the search space finite, Casper imposes recursive bounds on the production rules. For instance, it controls the maximum number of MapReduce operations a program summary may use, or the maximum number of *Emit* in a single transformer function. In §4.2, we discuss how Casper further specializes the search space for each input code fragment by changing the set of production rules available in the grammar or specifying different recursive bounds.

3.2 Proving the Validity of Program Summaries

Once the synthesizer comes up with a candidate summary, Casper will need to ensure that it correctly describes the semantics of the input program. There is extensive literature on proving the validity of program summaries for a given block of code [21, 33]. In Casper, we employ the standard approach of creating *verification conditions* based on Hoare logic [17]. Verification conditions are Boolean predicates which, given a program statement S and a postcondition (i.e., program summary) P, state what needs to be true *before* S is executed in order for P to be a valid postcondition of S. Verification conditions can be systematically generated for most types of imperative program statements, including those processed by Casper [33]. For each loop construct, however, an extra *loop invariant* is required. 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 loop invariants or postconditions is undecidable [21, 33]. Unlike prior work, however, there are two aspects that make our problem solvable: first, our postconditions are restricted to only those expressible using the IR described

$$
\begin{aligned}
&invariant(m; i) \equiv\ \ 0 \le i \le rows\ \ \wedge \\
&\quad m = map(reduce(map(mat[0::i]; \lambda_{m1}); \lambda_r); \lambda_{m2})
\end{aligned}
$$

(a) Outer Loop Invariant

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

(b) Verification Conditions

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

in §3.1, which lacks many features (e.g., pointers) that a general-purpose language would have. Moreover, we are only interested 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* that is sufficiently strong to prove the program summary shown in Figure 1(a). Figure 4(b) shows the verification conditions constructed by Casper that states what the program summary and invariant must satisfy. We can check that this loop invariant and program summary indeed satisfy these verification conditions as follows. First, the initiation statement asserts that the invariant holds before the loop, i.e., when `i` is zero. This is true because the invariant executes the MapReduce expression on only 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 statement asserts that after one more execution of the loop body, the i^{th} index of output vector `m` should now hold the mean for the i^{th} row of the matrix `mat`. This is true since the value of `i` is incremented during 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 that `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 to the number of rows in the matrix, and the loop invariant asserts that `m` equals to 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, while leaving the body of the summary (and any necessary invariants for loops) to be synthesized. The search space for the program summary and invariants are both expressed using the same IR as discussed in §3.1.

Formally, the synthesis problem is stated as follows:

$$\exists\, ps;\ inv_1; :::; inv_n:\ \forall \sigma: VC(P; ps; inv_1; :::; inv_n; \sigma) \quad (1)$$

In other words, Casper's goal is to find a program summary ps and any necessary invariants $inv_1; :::; inv_n$ such that for all program states σ, the verification conditions for the input code fragment P hold true. In Casper, this check is done by sending the verification conditions and the candidate program summary to a theorem prover, to be discussed in §4.1. The verified summary is then sent to the code generator to be translated to an executable MapReduce program.

3.3 Search Strategy

Casper uses an off-the-shelf program synthesizer, Sketch [29], to infer program summaries and loop invariants. Sketch takes as input a set of candidate summaries and invariants encoded using a grammar (e.g., Figure 3), and the correctness specification for the summary expressed using the verification conditions, and attempts to find a program summary (and any invariants needed) using the provided grammar such that the verification conditions hold true.

The existence of universal quantifiers in the Eq.1 makes the synthesis problem particularly challenging. To make this tractable, Casper implements a two-step verification process on top of the Sketch synthesizer. The first step, bounded model checking, checks a candidate program summary against the verification conditions over only a finite (i.e., "bounded") subset of all possible program states. For example, Casper will restrict 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 in Sketch, which uses Counter-Example Guided Inductive Synthesis (CEGIS) [31] for this purpose. Once Sketch finds a candidate program summary that 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. This two-step verification makes Casper's synthesis algorithm sound, without compromising on efficiency. Bounded checking done by the synthesizer acts as a filter to quickly eliminate all obviously incorrect program summaries. The more computationally expensive verification offered by the theorem prover is reserved only for the set of program summaries generated by the synthesizer in the first step.

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 $inv_1; ::: ; inv_n$ generated by the algorithm, the verification conditions hold over all possible program states after we execute the input code fragment P, in other words, $8\sigma : VC(P; ps; inv_1; ::: ; inv_n; \sigma)$.

3.3.1 Counter-Example Guided Inductive Synthesis. Figure 5 (lines 1 to 11) shows the core CEGIS algorithm that is used in the synthesizer that Casper builds upon. 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 generates a random program summary candidate ps and any needed invariants $inv_1; ::: ; inv_n$, from G, and randomly populates program states in Φ such that $8\sigma \in \Phi : VC(ps; inv_1; ::: ; inv_n; \sigma)$ is true. Next, the bounded model checker verifies if the candidate program summary holds over the entire bounded domain. If verification succeeds, CEGIS returns the summary as the solution. Otherwise, the bounded model checker produces a counter-example ϕ such that $VC(ps; inv_1; ::: ; inv_n; \phi)$ is false. The algorithm then adds ϕ to Φ and restarts the candidate program summary generator. This cycle iteratively executes until either a program summary is found to pass bounded model checking, or the search space is exhausted, i.e.,

$$8ps; inv_1; ::: ; inv_n \in G : 9\sigma \in \Phi : \neg VC(ps; inv_1; ::: ; inv_n; \sigma)$$

One issue with this approach is that while efficient, the found program summary might only be true for the finite domain, and thus will be rejected by the theorem prover when checking for general validity. In that case, Casper will dynamically change the search space grammar to exclude the candidate program summary that does not verify, and ask the synthesizer to generate a new program summary using the algorithm above. We will discuss this in detail in §4.1.

4 IMPROVING SUMMARY SEARCH

In this section, we discuss the techniques Casper uses to make the search for program summaries more robust and efficient.

4.1 Leveraging Verifier Failures

As mentioned, there might be cases where the program summary found by the synthesizer fails to be validated by the theorem prover. For instance, suppose the synthesizer sets the bound for the maximum value of integers to be 4. In that case, it will consider the expressions `v` and `Math.min(4,v)` (where `v` is program variable) to be equivalent and return either one as part of a program summary, even though they are not equal in practice. While prior work [11, 19] simply fails to translate such benchmarks if the candidate summary is rejected by formal verification, 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.

Definition 2. *(Completeness of Search)* An algorithm for generating program summaries is complete if and only if when there exists $ps; inv_1; ::: ; inv_n \in G$, then $8\sigma : VC(P; ps; inv_1; ::: ; inv_n; \sigma)) \rightarrow (\Delta \neq \emptyset)$, where G is the grammar being 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 it exists within the search space.

To achieve completeness, Casper must first prevent the same summaries that failed the theorem prover from being regenerated by the synthesizer. A naive approach would be to restart the synthesizer until a new summary is found, assuming that the algorithm implemented by the synthesizer is deterministic. This approach, however, is incomplete as the algorithm might never terminate since it will return the same incorrect summary. Casper instead modifies the search space. Recall from §3.3 that the search space for candidate summaries $\{c_1; ::: ; c_n\}$ 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 instead. This is implemented by passing additional constraints to the synthesizer to block a summary from being regenerated.

Figure 5 shows how Casper infers program summaries and invariants. Casper calls the synthesizer to generate a candidate

summary c on line 20, and attempts to verify c by passing it to the theorem prover on line 26. 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 later in §4.3.

Completeness and Soundness of Casper. Casper's algorithm for inferring program summaries is sound, as any program summary generated by the algorithm is verified by a theorem prover to be correct over all possible inputs. Our algorithm is also complete w.r.t. the input grammar as it does not terminate until either a correct summary is found or the search space has exhausted.

§7.5.4 provides experimental results that highlight the necessity of having a two phase verification in making search for the correct summary efficient, and how doing so enables us to translate code fragments that prior work is unable to do due to verifier failures.

4.2 Incremental Grammar Generation

While Casper's search algorithm is complete, the space of possible summaries to consider remains large. In this section, we discuss how Casper incrementally expands the space for program summaries to speed up the search. This is done by (1) adding new production rules to the grammar, and (2) increasing the number of times that each product rule is expanded.

There are two benefits to this approach. First, since search time for a valid summary is proportional to the size of the search space, Casper is often able to find valid summaries quickly, as our experiments show. Second, as higher grammar levels involve more expressive syntactically features, the found summaries will likely be more expensive computationally. Hence, biasing the search towards smaller grammars will likely produce program summaries that run efficiently.

4.2.1 Grammar Classes. Casper partitions the space of program summaries into different grammar classes, where each class is defined based on a number of syntactical features: (1) the number of Map/Reduce 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 the production rules in the search space grammar. A grammar hierarchy is created such as all program summaries that are expressible in a grammar class G_i is a syntactic subset of those expressible in a higher level class, i.e., G_j where $j > i$.

4.3 Casper's Search Algorithm for Summaries

Figure 5 shows the algorithm deployed in Casper 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 16). Then, it incrementally searches each grammar class $\gamma \in \Gamma$, invoking the synthesizer to find summaries in γ (line 20). Each summary (and invariants) returned by the synthesizer is checked by a theorem prover (line 26); 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

```
1  function synthesize (G, VC):
2    Φ = {} // summaries that failed bounded verification
3    while true do
4      ps, inv_{1..n} = generateCandidate(G, VC, Φ)
5      if ps is null then
6          return null // no summary exists in G
7      ϕ = boundedVerify(ps, inv_{1..n}, VC)
8      if ϕ is null then
9          return (ps, inv_{1..n}) // summary found
10     else
11         Φ = Φ ∪ ϕ // add failed candidate to set
12
13 function findSummary (A, VC):
14   G = generateGrammar(A)
15   Γ = generateClasses(G)
16   for γ ∈ Γ do
17     Ω = {} // summaries that failed full verification
18     Δ = {} // fully verified summaries
19     while true do
20       c = synthesize(γ - Ω - Δ, VC)
21       if c is null and Δ is null then
22         break // move to next grammar class
23       else if c is null then
24         return Δ // search complete
25       else if fullVerify(c, VC) then
26         Δ = Δ ∪ c
27       else
28         Ω = Ω ∪ c
29   return null // no solution found
```

Figure 5: Casper's search algorithm for program summaries

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 returns null, Casper returns the set of correct summaries Δ if it is non-empty. Otherwise, that means no valid solution was found, and the algorithm proceeds to search the next grammar class in Γ. If Δ is empty for 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 using `findSummary` to search for program summaries using the row-wise mean benchmark discussed in §2.2. Figure 6 shows three sample (incremental) grammars generated by Casper as a result of calling `generateClasses` in line 15 in Figure 5 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 or λ_r is 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 entry in the matrix to its row and column as the output.

As `findSummary` fails to find a valid summary in G_1 for the benchmark, the algorithm advances to the next grammar class G_2. G_2 expands upon G_1 by including summaries that consist of two *map* or *reduce* operator, and each λ_m can emit up to 2 key-value pairs. Not finding a valid summary again, the search moves to G_3, where G_3 expands upon G_2 with summaries that include up to three *map* or *reduce* operator, and the transformers can emit either integers or tuples. As shown in Figure 1(a), a valid summary is found within G_3 and added to Δ. Search continues in G_3 for other valid summaries in the same grammar class. The search terminates

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>

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

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

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

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

after all valid summaries, i.e., those that are 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 (see §7.6.1). While many frameworks come with optimizers that perform low-level optimization (e.g., fusing multiple map operators), performing *semantic transformations* are often difficult. For instance, as shown in the example to be discussed in §7.7, there are three distinct ways of encoding the word count problem, and they differ in the type of key-value pair emitted by *map*. While it is difficult for a low-level optimizer to discover such equivalences by analyzing actual Java code, Casper can instead perform such optimization as it searches for a high-level program summary expressed using the IR language. In this section we discuss how Casper uses a cost model and runtime monitoring module for this purpose.

5.1 Cost Model

We designed a cost model for Casper to evaluate different semantically equivalent program summaries that are found for a code fragment. As Casper aims to translate data-intensive applications, the cost model focuses on estimating data transfer costs as opposed to compute costs.

Each synthesized program summary is a sequence of *map* and *reduce* 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 cost of the corresponding operation. We define the cost functions of the *map* and *reduce* operations as a ternary operator $cost(\lambda; N; W)$, where λ is the transformer function for each *map* or *reduce*, N is the number of elements in the input data, and W is a weight factor.

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

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

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 all summed together to get the expected size of the output record. The total amount of data emitted during the map stage is equal to the product of expected record size and the number of times λ_m is executed (N). The function $cost_r$ is defined similarly, except that λ_r only produces 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 $W_{cs\partial}$. In our experiments, we used the weights 1, 2 and 50 for W_m, W_r and $W_{cs\partial}$ respectively.

To estimate the cost of a program summary, we sum the cost of each individual *map* or *reduce* operator. The first operator in the pipeline takes the variable N as the number of input records. For each subsequent stage, we use the number of key-value pairs generated by the previous stage (expressed as a function over N):

$$cost_{mr}([(op_1; \lambda_1); (op_2; \lambda_2); :::]; N) = cost_{op1}(\lambda_1; N; W) + cost_{mr}([(op_2; \lambda_2); :::]; count(\lambda_1; N))$$

The function count returns the number of key-value pairs generated by a given stage. For *map* stages, this is equal to $\sum_{i=1}^{|emits|} p_i$, and for *reduce* stages, it is equal to the number of unique key values that the reducer was executed on.

5.2 Dynamic Cost Estimation

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

In such cases, Casper will generate code for all the remaining summaries that have been validated, and uses a runtime monitoring module to evaluate their costs dynamically as the generated program executes. As the program executes, the runtime module samples values from the input data set (Casper currently uses first-k values sampling, although different sampling method may be used), and then uses the samples to compute the probabilities of conditionals by counting the number of data elements in the sample that a conditional will evaluate to true, and counting the number

of unique data values that are emitted as keys. These estimates are inserted into Eqn 2 and Eqn 3 to get the final comparable cost values for each program summary. Finally, the summary with the lowest cost is executed at runtime. Hence, if the generated program is executed over different data distributions, it may run different implementations for each input, as illustrated in §7.7.

6 IMPLEMENTATION

We have implemented Casper using the Polyglot framework [24] to parse Java code, identify candidate code fragments, and perform program analysis. We use Sketch [29] to implement the candidate generator and bounded checker, and Dafny as the theorem prover. Code generation modules for all three target frameworks are also implemented using Polyglot. In this section, we discuss how Casper identifies code fragments for translation and stitches the generated MapReduce code back into the original program.

6.1 Identifying Code Fragments

Casper iterates through the entire source code to identify candidate code fragments for translation. Casper currently targets loop nests that iterate over Java arrays or `java.lang.Collections` as candidate code fragments since those are most amenable to be translated. Casper currently does not translate any code with recursive function calls, but support calls to a number of common Java library methods (e.g., `java.lang.Math`). Supporting more methods will require additional engineering and does not affect our underlying technique. As our experiments show, our current prototype can already convert a wide variety of code fragments from sequential Java to MapReduce.

6.2 Code Generation

Once an identified code fragment is translated, Casper will replace the original code fragment with the translated MapReduce program. In addition, Casper also generates "glue" code to weave the MapReduce task into the program, this includes importing necessary Spark packages, creating a `SparkContext` (or an `ExecutionEnvironment` for Flink), converting data into an RDD (or Flink's DataSet), parallelizing input Collections, broadcasting required variables or updating the program state using the computed output. Since some APIs such as Spark's `reduceByKey` are not defined for non-commutative associative reducers, Casper only uses these API if the commutative associative properties can be proved by Dafny. Finally, Casper also generates code for sampling input data statistics and dynamic switching, as discussed in §5.2.

7 EVALUATION

We evaluated Casper by using it to translate 84 benchmarks from 5 different suites collected from prior work and real-world applications. We evaluated various aspects of the Casper generated code as described below. For all our experiments, we executed the benchmarks on an AWS cluster of 10 m3.2xlarge instances, 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 2.1.0, Hadoop 2.7.3, and Flink 1.2.0. The data files for all experiments were stored on HDFS.[1]

7.1 Benchmark Suites

To evaluate Casper's ability to translate different varieties of code, we chose our benchmarks from various domains:

- **Phoenix** [26] is a collection of standard MapReduce problems such as *WordCount*, *StringMatch*, *Histogram*, etc as used in prior work [25]. Since the original sequential implementations were 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.
- **Bigλ** [30] comprises of several data analysis tasks such as *sentiment analysis*, *database operations* (such as 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. The results in 211 lines of code across 7 files.
- **Fiji** [15] is a popular distribution of the ImageJ [18] library for scientific image analysis. We ran Casper on the source code for four Fiji packages (aka plugins): *NL Means* is a plugin for denoising images via the non-local-means algorithm [9] with optimizations [13]. *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, and *Trails* averages pixel intensities over a time window in an image sequence. These packages, authored by different developers, consist of 1411 lines of code across 5 files.
- **Stats** were benchmarks automatically extracted by Casper from an online repository for statistical analysis of data [20]. Examples include *Covariance*, *Standard Error*, and *Hadamard Product*. The repository consists of 1162 lines of code across 12 Java files, mostly consisting of vector and matrix operations.
- **Ariths** are simple mathematical functions and aggregations collected from prior work [10, 12, 27]. Examples include *Min*, *Max*, *Delta*, and *Conditional Sum*. The suite contains 245 lines of code across 11 files.

7.2 Benchmark Extraction

For each benchmark suite, we ran Casper on all Java files to identify and translate benchmarks. For benchmark suites obtained from prior work, such as Ariths and Bigλ, benchmark extraction was trivial since each benchmark was implemented in a separate file and usually consisted of a single method containing a single data processing loop. For suites gathered from real-world sources, such as Stats and Fiji, Casper identified candidate code fragments automatically as described in §6.1. We manually inspected all source files to identify 84 benchmarks (across all 5 suites) that were possible to translate to MapReduce and met Casper's criteria. Casper was able to successfully detect all of them as candidates code fragments.

The benchmarks extracted by Casper form a diverse and challenging problem set. As shown in Table 1, they vary across programming style as well as the structure of their solutions.

[1] We have uploaded the source and translated benchmarks to our GitHub repository: https://github.com/uwplse/Casper/tree/master/bin/benchmarks.

7.3 Feasibility Analysis

We first evaluated whether Casper can translate the extracted benchmarks. Table 2 shows the results. Of the 84 benchmarks, 65 were successfully translated. 3 of the 19 failures were caused by calls to library methods that are currently not handled by Casper. Another 6 benchmarks could not be translated as they required broadcasting data to multiple reducers, and the lack of loop construct in Casper's IR for program summaries prevents such benchmarks from being translated. 10 benchmarks could not be synthesized because Casper timed out of 90 mins as the search space grammar was not expressive enough. All loops other than the 84 counted above were never recognized as candidates for translation because they did not iterate over any data, such as loops printing output to console or loops for populating arrays.

As MOLD is not publicly available, we obtained the generated code from the MOLD authors for the benchmarks used in their evaluation [25]. Of the 7 benchmarks, MOLD was unable to translate 2 of them (PCA and KMeans). Another 2 (Histogram and MatrixMultiplication) generated semantically correct translations but failed to execute on the cluster as they ran out of memory. For the remaining 3 benchmarks (WordCount, StringMatch and LinearRegression), MOLD generated working implementations. In contrast, Casper translated 4 out the 7 MOLD benchmarks. For PCA and KMeans, Casper translated and successfully executed a subset of all the loops found, while for other loops, along with the MatrixMultiplication benchmark, require broadcasting data across different nodes, which we suspect will not lead to an efficient implementation by Casper even if translated.

7.4 Performance of the Generated Benchmarks

In the previous section, we showed that Casper can translate a variety of benchmarks. In this section, we examine the quality of the translations produced by Casper.

7.4.1 Speedup. We used Casper to automatically translate the identified 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 randomly generated datasets of sizes 25GB, 50GB, and 75GB. On average the Spark implementations generated by Casper are 18:1× faster than their sequential counterparts, with max improvement up to 32:2×. Table 2 shows the mean and max speedup observed for each benchmark using Spark on a 75GB dataset.

Figure 7 plots the speedup achieved by the MOLD generated implementations for the StringMatch, WordCount and LinearRegression benchmarks. Spark translations for the three benchmarks generated by MOLD performed 12.3× faster than the sequential versions. The solutions generated by Casper for the StringMatch and LinearRegression benchmarks were faster than the ones generated by MOLD. In StringMatch, Casper found an efficient encoding to reduce the amount of data emitted in the map stage (see §7.7), 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 whereas Casper computed the result for all keywords in the same operations. In LinearRegression, MOLD discovered the same overall algorithm as Casper except

Benchmark Properties	# Extracted	# Translated
Conditionals	22	15
User Defined Types	10	6
Nested Loops	35	17
Multiple Datasets	17	13
Multidim. Dataset	38	23

Table 1: Properties of benchmarks translated by Casper.

Source	# Translated	Mean Speedup	Max Speedup
Phoenix	7 / 11	14.8x	24.2x
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

Table 2: Number of benchmarks translated by Casper.

that the MOLD implementation zipped the input RDD by its index as a pre-processing step, almost doubling the size of input data and hence the amount of time spent in data transfers.

We also executed the Hadoop and Flink implementations generated by Casper on the subset of benchmarks shown in Figure 7. The average speedups observed by Hadoop and Flink implementations (over the 10 benchmarks) are 6.4× and 10.8× respectively. We have also manually translated a subset of the benchmarks to the new Spark DataSet API [32], but we did not experience better performance on the new API for these benchmarks as compared to the RDD API. Hence our current prototype only translates to Spark's RDD API.

7.4.2 Comparison to Expert Implementations. To further evaluate the quality of the Casper generated implementations, we recruited expert Spark developers to manually rewrite 50 of the benchmarks.[2] Figure 7 shows a subset of the performance comparison of the Casper generated and the manually translated implementations. The results show that the Casper generated implementations perform competitively with those written by experts. In fact, Casper discovered the same high level algorithm as the one used by the experts for 24 of them, 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, we did not observe much performance difference. One interesting case is the 3D Histogram benchmark, where the expert developer exploited knowledge about the data to improve the runtime performance. Specifically, the developer recognizes that since the RGB values always range between 0-255, the histogram data structure will never grow larger than 768 values. Therefore, they used Spark's more efficient *aggregate* operator to implement the solution. Casper, not knowing that the pixel RGB values are bounded, had to assume that the number of keys can grow to be arbitrarily large and thus using the aggregate operator may cause out of memory errors.

7.4.3 Scalability. Next, we evaluated the scalability of the Casper generated implementations. To do so, we executed our benchmarks on different amounts of data and measured the resulting speedups. As shown in Figure 8, the Casper generated Spark implementations exhibit good data parallelism and show a steady increase in

[2]We considered the number of hours worked on the platform, education, reviews and relevant past projects as metrics to recruit developers.

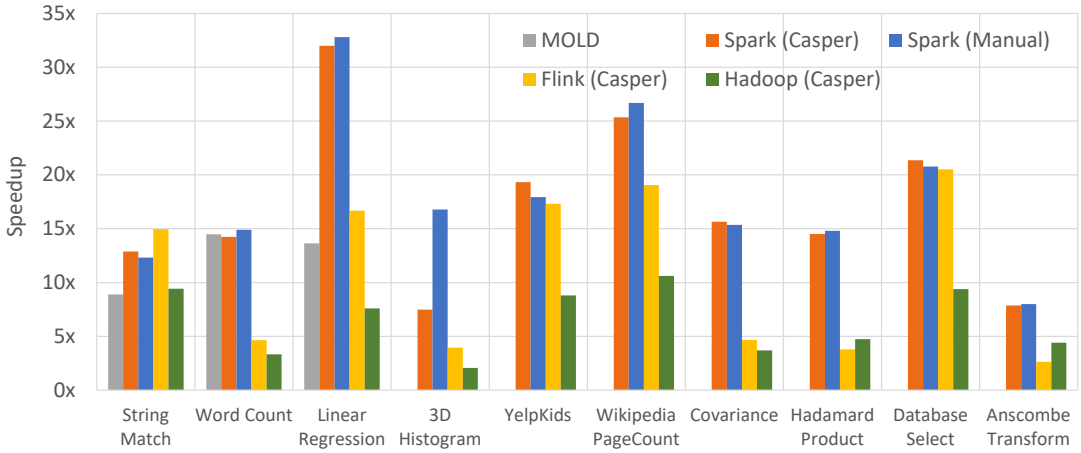


Figure 7: Implementations generated by Casper perform competitively against manual translations.

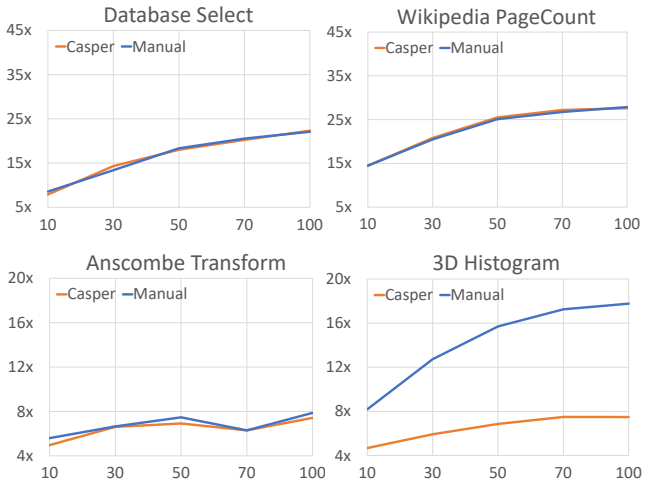


Figure 8: The top 2 benchmarks with the most performance improvement along with the bottom 2. The x-axis plots the size of input data, while the y-axis plots the runtime speedup over sequential implementations.

speedups across all translated benchmarks as the input data size increases, up until the cluster reaches maximum utilization.

7.5 Compilation Performance

Next, we measured the compilation time taken by Casper on the benchmarks, the effectiveness of Casper's two-phase verification strategy, effectiveness of incremental grammar generation and the quality of the generated code.

7.5.1 Compile Time. On average, Casper took 6.4 minutes to compile a single benchmark. However, the median compile time for a single benchmark was only 104.6 seconds. This is because for some benchmarks, the synthesizer discovered a low-cost solution during the first few grammar classes, enabling Casper to terminate search early. Table 3 shows the mean compilation time for a single benchmark by suite.

Source	Mean Time (s)	Mean LOC	Mean # Op	Mean TP Failures
Phoenix	944	13.8 (13.1)	2.3 (2.1)	0.3
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

Table 3: Summary of Casper's compilation performance. Values for the manual translations are shown in brackets.

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 4: With incremental grammar generation Casper generates far less summaries.

7.5.2 Two-Phase Verification. In our experiments, the candidate summary generator produced at least one incorrect solution for 12 out of the 65 successfully translated benchmarks. A total of 67 incorrect summaries were proposed by the synthesizer across all benchmarks. Table 3 lists the average number of times a solution was rejected by the theorem prover for each benchmark group. As an example, the Delta benchmark computes the difference between the largest and the smallest value 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 as discussed in §4.1.

7.5.3 Generated Code Quality. Besides measuring the runtime performance of the Casper generated implementations, we manually inspected the code generated by Casper and compared that to the expert implementations for two code quality comparisons:

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

lines of code (LOC) and the number of MapReduce operations used. Table 3 shows the average values in both sets of implementations. The results show that the implementations generated by Casper are comparable and do not use more MapReduce operations or LOC than are necessary to implement a given task.

7.5.4 Incremental Grammar Generation. We next measured the effectiveness of incremental grammar generation in optimizing search. To measure the impact of incremental grammar generation on compilation time, we used Casper to translate benchmarks without incremental grammar generation and compared the results. The synthesizer was allowed to run for at most an hour, after which the process was manually killed. The results of this experiment are summarized in Table 4. As the results show, 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 (a slowdown by at least one order of magnitude).

7.6 Cost Model

In this section, we measured whether Casper's cost model model can efficiently identify efficient solutions during the search process.

7.6.1 Accuracy. 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 [14], 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 5 shows the amount of data shuffled along with the corresponding runtimes for both implementations using the 75GB dataset. As shown, the implementation that uses combiners to reduce data shuffling was almost an order of magnitude faster.

Next, we verify 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 9(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 solution 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 5, the implementation that minimized the amount of data emitted in the map-phase executed almost twice as fast.

Finally, we randomly selected the Arithmetic Average benchmark and exhaustively generated all (160) correct solutions in the search space. Then, we executed each solution on a 25GB dataset to identify the solutions with the actual lowest runtime cost. We next used the cost model to sort the generated solutions and found that the solution picked by the cost model as optimal was indeed one of the solutions with the lowest runtime cost.

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

7.7 Dynamic Tuning

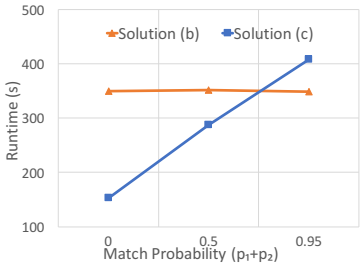
In the final set of experiments, we evaluated the runtime monitor module and whether the dynamic cost model is able to successfully select the correct implementations. As explained in §5.2, the performance of some solutions depends on the distribution of the input data. To evaluate, we used Casper to generate the three different implementations for the StringMatch benchmark (Figure 9(a)). Figure 9(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: 40bytes for String, 10bytes for Boolean and 28bytes for a Tuple of Boolean Objects. Solution (a) can be disqualified at compile time since as it is not optimal for any data distribution. 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 skew of the input data, i.e., how often do the keywords appear in the text, and they determine which solution is the most optimal.

Casper handles this by generating a runtime monitor in the output code. The monitor samples the input data (first 5000 values) on each execution to estimate the probabilities for conditionals that depend on the input data. The estimated values (0.988 and 0.94 for p_1 and p_2 respectively) are then plugged back into the Eqn 2 and 3 to get the final costs. The solution with the lowest cost is then selected for execution at runtime.

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 9(c) shows the dynamically computed final cost of solution (c) using p_1 and p_2 estimates calculated by using sampling. The actual performance of the two solutions is illustrated in Figure 9(b). For datasets with very high skew, it is beneficial to use solution (b) due to the smaller size of a key-value pair emit. Otherwise, solution (c) performs better. Casper, with the help of

```
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 9: StringMatch Benchmark: Casper dynamically selects the optimal implementation for execution at runtime.

the dynamic input from the runtime monitor, makes this inference and selects the correct solution for all three datasets.

8 RELATED WORK

Implementations of MapReduce. MapReduce [14] is a popular programming model that has been implemented by various systems [4–6]. Such systems provide their own high-level DSLs that users must use to express their computation. In contrast, Casper works with native Java programs and infers rewrites automatically.

Source-to-Source Compilers. Many efforts translate programs from low-level languages into high-level DSLs. MOLD [25], 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 [22]. Unlike prior approaches in automatic parallelization [1, 7], Casper targets data parallel processing frameworks, and only translates code fragments that are expressible in the DSL for program summaries.

Synthesizing Efficient Implementations. Prior work has used synthesis to generate efficient implementations and optimizing programs. [30] synthesizes MapReduce solutions from user-provided input and output examples. QBS [11] and STNG [19] both use verified lifting and synthesis to convert low-level languages to specialized high-level DSLs for database applications and stencil computations respectively. Casper is inspired by prior approaches in applying verified lifting to construct compilers. Unlike prior work, however, Casper addresses the problem of verifier failures and designs a grammar hierarchy to prune away non-performant summaries, in addition to a dynamic cost model and runtime monitoring module for choosing among different implementations at runtime.

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 discover high-level summaries of each input code fragment using program synthesis, and retarget the found summary to the MapReduce framework. 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 32.2× as compared to the original implementation, and is competitive with translations done manually by expert developers.

REFERENCES

[1] 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.

[2] Apache Flink 2017. Apache Flink. https://flink.apache.org/. (2017). Accessed: 2016-11-16.

[3] Apache Hadoop 2017. Apache Hadoop. http://hadoop.apache.org. (2017). Accessed: 2016-11-16.

[4] Apache Hive 2016. Apache Hive. http://hive.apache.org. (2016). Accessed: 2016-04-20.

[5] Apache Pig 2016. Apache Pig. http://tensorflow.org/. (2016). Accessed: 2016-05-01.

[6] Apache Spark 2017. Apache Spark. https://spark.apache.org. (2017). Accessed: 2016-11-16.

[7] 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.

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

[9] 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) - Volume 2 - Volume 02 (CVPR '05)*. IEEE Computer Society, Washington, DC, USA, 60–65. https://doi.org/10.1109/CVPR.2005.38

[10] 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). http://arxiv.org/abs/1605.01497

[11] 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. https://doi.org/10.1145/2491956.2462180

[12] Przemyslaw Daca, Thomas A. Henzinger, and Andrey Kupriyanov. 2016. Array Folds Logic. *CoRR* abs/1603.06850 (2016). http://arxiv.org/abs/1603.06850

[13] JÃĪrÃťme 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. http://dblp.uni-trier.de/db/conf/isbi/isbi2008.html#DarbonCCOJ08

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[15] Fiji 2016. Fiji: ImageJ. https://github.com/fiji. (2016). Accessed: 2016-04-20.

[16] 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. https://doi.org/10.1145/1836089.1836091

[17] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

[18] ImageJ 2016. ImageJ. https://github.com/thisMagpie/Analysis. (2016). Accessed: 2016-04-20.

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

[20] MagPie Analysis Repository 2016. MagPie Analysis Repository. https://dato.com/. (2016). Accessed: 2016-04-20.

[21] 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. https://doi.org/10.1007/11916277_25

[22] 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. https://doi.org/10.1145/2159430.2159431

[23] 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. https://doi.org/10.1109/ICDM.2008.142

[24] Polyglot 2016. Polyglot. http://www.cs.cornell.edu/Projects/polyglot/. (2016). Accessed: 2016-05-01.

[25] 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. https://doi.org/10.1145/2660193.2660228

[26] 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. https://doi.org/10.1109/HPCA.2007.346181

[27] 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. https://doi.org/10.1145/2815400.2815418

[28] 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. https://doi.org/10.1145/582095.582099

[29] Sketch 2016. Sketch. https://people.csail.mit.edu/asolar/. (2016). Accessed: 2016-05-01.

[30] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. *SIGPLAN Not.* 51, 6 (June 2016), 326–340. https://doi.org/10.1145/2980983.2908102

[31] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

[32] Spark Dataset API 2017. Spark DataSet API. https://spark.apache.org/docs/2.1.1/api/java/org/apache/spark/sql/Dataset.html. (2017). Accessed: 2017-11-02.

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