Optimizing Data-Intensive Applications Automatically By Leveraging Parallel Data Processing Frameworks

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

Alvin Cheung
University of Washington
Seattle, USA
akcheung@cs.washington.edu

ABSTRACT

In this demonstration we will showcase CASPER, a novel tool that enables sequential data-intensive programs to automatically leverage the optimizations provided by parallel data processing frameworks. The goal of CASPER is to reduce the inertia against adaptation of new data processing frameworks—particularly for non-expert users—by automatically re-writing sequential programs written in general purpose languages to the high-level DSLs or APIs of these frameworks. Through CASPER's browser-based interface, users can enter the source code of their Java applications and have it automatically retargeted to execute on Apache Spark. In our interactive presentation, we will use CASPER to optimize sequential implementations of data visualization programs as well as image processing kernels. The optimized Spark implementations along with the original sequential implementations will then be executed simultaneously on the cloud to allow the demo audience compare the runtime performances and outputs in real-time.

1. INTRODUCTION

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

Many parallel data processing frameworks have been developed to process large datasets, and new ones continue to be released. Such frameworks often come with domain-specific optimizations that are exposed either via library APIs or high-level domain-specific languages (DSLs) for developers to express their computations. The goal is to free the developer from worrying about low-level details such as specifying communication and partitioning data across machines. The resulting computations are made efficient due to the highly optimized domain-specific implementations of the framework.

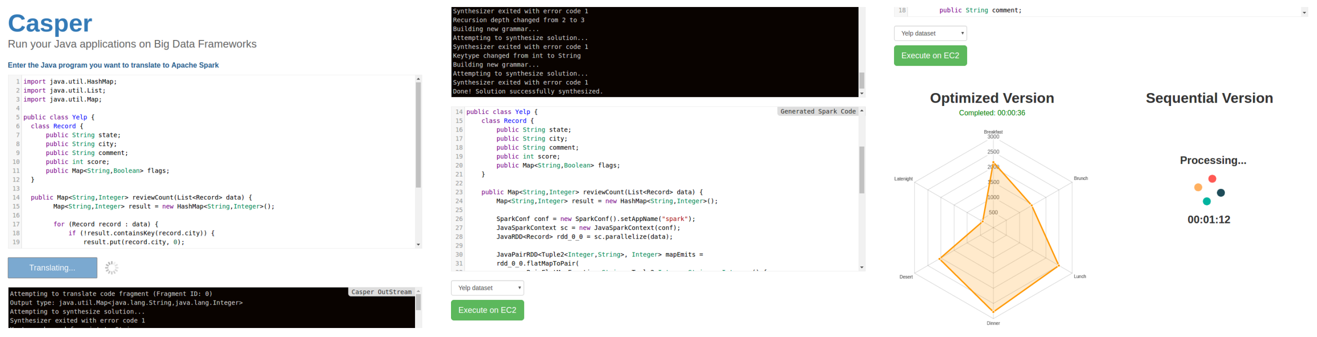
SIGMOD'17, May 14-19, 2017, Chicago, IL, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3056440

However, leveraging such frameworks for large-scale data analytics is often not an easy task. First, choosing the appropriate framework to deploy a given workload requires deep understanding of the optimizations provided by each framework. Second, users must learn the new APIs or DSLs and often rewrite existing code before they can leverage the benefits provided by these frameworks. Doing so requires not only significant time and resources but also risks introducing new bugs into the application. Moreover, even if users are willing to rewrite their applications, they must first understand the intent of their code, which might have been written by others who were targeting a different framework in the past. And manually written, framework-specific optimizations embedded in existing code often obscure high-level intent. Finally, even after learning new APIs and rewriting code, newly emerging frameworks turn freshly rewritten code into legacy applications. Users must then repeat this process to keep pace with new advances, requiring significant time investments that could be better spent in developing new applications instead.

One way to improve the accessibility of parallel data processing frameworks is to build compilers that convert applications written in common general-purpose languages, such as Java or Python, to high-performance parallel processing frameworks, such as Hadoop or Spark. Such compilers allow users to write their applications in familiar general-purpose languages and the compilers parallelize their computation by retargeting portions of their code to high-performance DSLs [11]. Unfortunately, such compilers don't always exist, and building one is highly non-trivial given the varying APIs and DSLs offered by different frameworks, and the large number of frameworks available.

In this demonstration, we present CASPER [1], a compiler that uses *verified lifting* [5, 8] to automatically convert sequential code fragments into the MapReduce paradigm [6]. Verified lifting is a general technique that converts code fragments from one language to another by leveraging program synthesis and verification to *automatically find* provably correct functional summaries. These summaries—expressed using a high-level specification language—encode the semantics of the input code fragment, but strip away any domain-specific optimizations. The search process for summaries is completely automated without any predefined rules, as in the case of a syntax-driven compiler. Once found, the summaries (and thus the input code fragment) can be translated to the target language using simple syntax-directed rules.

The general concept of verified lifting has been previously applied to build compilers that convert Java code fragments



```
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Yelp {
  class Record {
      public String state;
      public String city;
      public String comment;
      public int score;
      public Map<String,Boolean> flags;
  }

  public Map<String,Integer> reviewCount(List<Record> data) {
        Map<String,Integer> result = new HashMap<String,Integer>();

        for (Record record : data) {
            if (!result.containsKey(record.city)) {
                result.put(record.city, 0);
```

```
Attempting to translate code fragment (Fragment ID: 0)
Output type: java.util.Map<java.lang.String,java.lang.Integer>
Attempting to synthesize solution...
Synthesizer exited with error code 1
```

(a)

```
Synthesizer exited with error code 1
Recursion depth changed from 2 to 3
Building new grammar...
Attempting to synthesize solution...
Synthesizer exited with error code 1
Keytype changed from int to String
Building new grammar...
Attempting to synthesize solution...
Synthesizer exited with error code 1
Done! Solution successfully synthesized.
```

```
public class Yelp {
    class Record {
        public String state;
        public String city;
        public String comment;
        public int score;
        public Map<String,Boolean> flags;
    }

    public Map<String,Integer> reviewCount(List<Record> data) {
        Map<String,Integer> result = new HashMap<String,Integer>();

        SparkConf conf = new SparkConf().setAppName("spark");
        JavaSparkContext sc = new JavaSparkContext(conf);
        JavaRDD<Record> rdd_0_0 = sc.parallelize(data);

        JavaPairRDD<Tuple2<Integer,String>, Integer> mapEmits =
        rdd_0_0.flatMapToPair(
```

(b)

```
        public String comment;
```

(c)

Figure 1: Casper's Browser-Based Front-End

to SQL [5], and FORTRAN stencil kernels to the Halide DSL [8]. In Casper, we employ verified lifting to convert sequential code fragments written in Java into a high-level functional language that describes the computation using the MapReduce paradigm. By doing so, we enable sequential Java programs to leverage the various optimizations offered by data parallel processing frameworks that implement the MapReduce paradigm (e.g., Hadoop, Spark, etc). While our current prototype targets Apache Spark, our underlying technique is general enough to be used for other backends.

Our demonstration will enable attendees to experience the simplicity of optimizing programs using Casper and assess its capabilities and limitations. We will use Casper's browser-based interface to retarget data visualization queries, as well as an image processing kernel. Furthermore, attendees will be invited to make changes to the available benchmarks or write their own functions. To judge the quality and correctness of the translation, users can manually examine the generated Spark code or simply execute both implementations (original and optimized) simultaneously on a cluster to compare their performance and outputs.

2. DEMONSTRATION DETAILS

Our demonstration of Casper consists of two phases, where each phase involves optimizing a different data intensive application. The applications were manually selected to present the unique challenges for the system and encourage audience participation.

2.1 Data Analytics and Visualization

In this phase of the demonstration, we act as a data scientist studying the Yelp dataset and generating visualizations from it. The Yelp dataset consists of millions of reviews and hundreds of thousands of customer and businesses records. To help the users get started, we have pre-coded two simple programs for the users to optimize and run:

- Program 1 counts the number of restaurants in the entire dataset that serve breakfast, brunch, lunch, dinner, late-night and dessert. The output of this program is visualized as a Radar graph (see Figure 1c).

- Program 2 counts the number of businesses with a 4-star rating or higher, grouping the results by city. The output of this program is visualized as a heat map.

As a program is being translated, users will be able to observe the different iterations our synthesis algorithm goes through as it searches for the optimal rewrite (Figure 1a). Once the compilation is complete, the new version of the program, parallelized using Apache Spark, will be displayed to the users (Figure 1b). The users can then select the appropriate dataset to run the benchmark on and click execute. This will compile both sequential and optimized versions of the program and send them to our remote clusters to be executed. As soon as the output from either implementation is ready, it will be displayed to the users in the form of a visualization (see Figure 1c). The time taken for each implementation to finish will also be reported to the users. In our experiments, we found that the optimized implementations of Yelp queries running on our cluster, executed roughly 17× faster (over a 75GB dataset) than the sequential implementations running on a single node.

Users will have the freedom to edit the provided queries or write their own queries for the dataset. Code for both parsing the dataset and generating visualizations has been written into our system, therefore the users are required to program only the actual analytics they want. Users will also have the chance to study the format of the data they are working with using a small sample of the original log.

2.2 Image Processing

In the second phase, we will optimize an application of a different flavor to challenge Casper. The users will be provided a pre-written image processing benchmark which takes a stream of images and blurs consecutive images together using a fixed-sized window.

The goal for this session is to provide a different challenge for the compiler and demonstrate that Casper can be used for a wide array of applications. The input to the benchmark is an array of images, where each image is stored as a flattened 1-D array of RGB tuples. The code iterates the data via two nested loops: the outer loop iterates each pixel index, while the inner loop iterates all images in the current window. The inner loop computes the sum of RGB values in each image for the current pixel index. The outer loop uses the computed sum to calculate the average RGB values, which are then used as the pixel values for the output image. The presence of nested loops requires Casper to synthesize multiple loop invariants and postconditions (see §3.3). Moreover, the optimal Spark solution requires more than two MapReduce operations. All these factors contribute to making the synthesis problem harder.

The users will follow the same instructions to compile

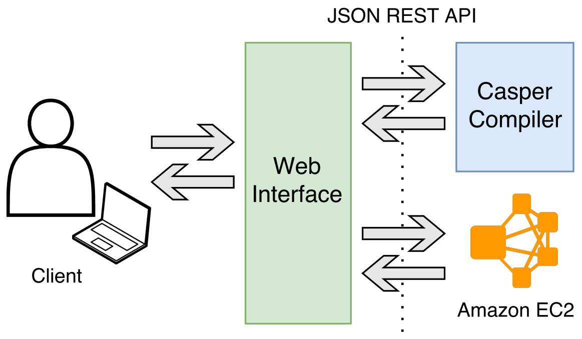


Figure 2: CASPER's deployment

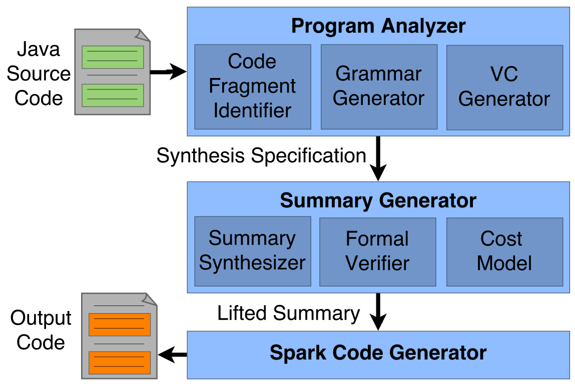


Figure 3: CASPER System Architecture.

the benchmark and run the implementations on our images dataset. The images dataset contains a small number of very large images that must all be blurred together. The final blurred image is displayed to the users when the benchmark finishes. According to our experiments, users should experience a 9× performance boost on the 75GB dataset.

3. SYSTEM OVERVIEW

In this section, we present an overview of CASPER's design and briefly discuss the methodology used to translate sequential code fragments to the MapReduce paradigm (and ultimately Spark code).

3.1 System Deployment

As shown in Figure 2, CASPER is deployed as an online web service, accessible through a RESTful JSON API. The user interacts with the browser-based front-end (see Figure 1) that we have developed for the web service. Once the users submit their code through the webpage, it is sent to the CASPER compiler where it is translated to Spark. The console output generated by the compiler is printed back to the user in realtime to help them debug their applications and track the compilation process (see Figure 1a).

Once (and if) the compiler has successfully translated the code, the retargeted version of the program is sent back to the user for review (see Figure 1b). For the demonstration, we will set up an AWS cluster used to run the Spark implementations using 10 m3.2xlarge instance nodes running Spark 2.0.1. The sequential version is run on a separate m3.xlarge machine simultaneously. All datasets accessed by the programs are stored on the cluster HDFS. Once the output of the applications is ready, it is printed back to the user. (Figure 1c).

3.2 System Architecture

Figure 3 shows the overall architecture of CASPER. The input to CASPER is Java code that iterates data sequentially. As output, CASPER generates an updated version of the original program where the translated code fragments have been replaced with semantically equivalent Spark tasks.

The first component in the compilation pipeline is the *Program Analyzer*. It parses the input code into an Abstract Syntax Tree (AST) and automatically identifies loop nests that iterate over data as candidate code fragments for translation. Once such code fragments have been identified, each of them is individually examined through static program analysis to generate a synthesis specification. The synthesis specification defines both the space of high-level MapReduce representations to search from and the verification conditions that a candidate MapReduce representation must satisfy to preserve the semantics of the original code.

The *Summary Generator* module uses a program synthesizer called SKETCH [12] to find (i.e. lift) semantically equivalent MapReduce representations of the input code fragment (also referred to as summaries of the code fragment). CASPER currently uses the Dafny [9] theorem prover to confirm that the lifted result is sound, i.e., the generated summary and the original input code fragment are semantically equivalent for all possible program executions. Since a sequential program may have more than one valid MapReduce representation, a cost model is used to select the best one. §3.3 discusses the synthesis process in greater detail.

Finally, the *Spark Code Generator* uses the optimal MapReduce representation to generate syntactically correct Apache Spark code. This is achieved through relatively straightforward syntax directed rules. The code generator also generates a new version of the original program where the successfully translated code fragments have been replaced by the generated Spark code. The code generator is also responsible for generating any supplementary code necessary to incorporate the generated Spark job back into the original program, such as creating a Spark context.

3.3 Methodology

The translation of sequential Java code to Spark is done in two distinct stages: first, the input code is lifted, through synthesis, into a high-level MapReduce specification language. Then, simple re-write rules are used to generate code from the specification language to the target language (in this case Spark). Inferring program semantics in a custom intermediate language rather than the target DSL directly enables us to make synthesis more efficient, and allows easy extension of support to other data processing frameworks.

MapReduce Specification Language: The goal of verified lifting is to generate a summary of the selected code fragment. The summary must correctly capture the behavior of the code by specifying how the program state is manipulated as the code executes. In CASPER, the summaries define the value of output variables in the form of functional MapReduce primitives, such as `flatMap` and `reduceByKey`, applied to the input data. The summaries can be seen as postconditions of the input code fragment (i.e. logical expressions that must be true after the input code fragment terminates). The language has been designed such that any postcondition expressed using it can be easily translated to parallel data-processing frameworks (e.g., Spark) that implement the MapReduce paradigm.

Verification: Given a specific code fragment and a candidate summary of the code, Casper must be able to prove that the summary is a valid postcondition of the code fragment. To do so, Casper employs the standard approach of creating *verification conditions* based on Hoare logic [7]. Verification conditions are Boolean predicates which, given a program statement s and a postcondition *post*, state what needs to be true before s is executed. If the verification condition holds for all possible program states, then the postcondition *post* also holds. Verification conditions can be systematically generated but require a sufficiently strong loop invariants for each loop in the program. Since these loop invariants are unknown to Casper, they must also be synthesized.

Search Strategy: To infer these postconditions and loop invariants, Casper uses Syntax-Guided Synthesis (SyGuS). SyGuS takes as input a set of candidate postconditions expressed as a grammar, and a correctness specification for the postcondition expressed as a logical formula. The goal for the synthesizer is to construct a postcondition using the provided grammar such that the formula evaluates to true. By specifying a grammar that can only generate postconditions in our specification language, Casper ensures that any synthesized postcondition will be easily translatable to the target DSL. Once the synthesis problem has been defined, Casper uses Counter-Example Guided Inductive Synthesis (CEGIS) to solve the search problem. Once a solution is found, and verified by Dafny, the search space is pruned—by updating the grammar—to eliminate all candidate solutions of equal or worse quality (as determined by our cost model) after which the search is restarted to find a better solution until the search space is exhausted.

4. RELATED WORK

Source-to-Source Compilers. Many efforts have been made to translate programs directly from low-level languages into high-level DSLs. MOLD [11], a source-to-source compiler, relies on syntax-directed rules to convert native Java programs to Apache Spark. Unlike MOLD, we translate on the basis of program semantics. This eliminates the need for rewrite rules, which are difficult to generate and brittle to code pattern changes. Many source-to-source compilers have been built in other domains for similar purposes. For instance, [10] evaluates numerous tools for C to CUDA transformations. However, these compilers require manual effort to annotate the original source code. Our methodology works with code without any user annotation, and is built upon our earlier work [1]. Unlike prior approaches in automatic parallelization [2, 3], 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. [13] is the most recent research that attempts to synthesize MapReduce solutions with user-provided input and output examples. QBS [5] and STNG [8] 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 cost model to prune away non-performant summaries, with the latter inspired by recent work on cost-based synthesis [4].

5. CONCLUSION

Casper aims to improve adaptation of new data processing frameworks by automatically retargeting applications written in general purpose languages to the high-level DSLs provided by parallel data processing frameworks. In our demonstration we showcase Casper's browser-based interface and have users parallelize data-processing applications using Casper.

6. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through grants IIS-1546083, and CNS-1563788; DARPA award FA8750-16-2-0032; DOE award DE- SC0016260; and gifts from Adobe, Amazon, and Google.

7. REFERENCES

[1] M. B. S. Ahmad and A. Cheung. Leveraging parallel data processing frameworks with verified lifting. In *SYNT@CAV*, 2016.

[2] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *PPSC*, 1995.

[3] W. Blume, R. Eigenmann, J. Hoeflinger, D. A. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic detection of parallelism: A grand challenge for high performance computing. *IEEE P&DT*, 2(3), 1994.

[4] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze. Optimizing synthesis with metasketches. In *POPL*, 2016.

[5] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, 2013.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), Jan. 2008.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[8] S. Kamil, A. Cheung, S. Itzhaky, and A. Solar-Lezama. Verified lifting of stencil computations. *PLDI*, June 2016.

[9] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.

[10] C. Nugteren and H. Corporaal. Introducing 'bones': A parallelizing source-to-source compiler based on algorithmic skeletons. In *GPGPU-5*, 2012.

[11] C. Radoi, S. J. Fink, R. Rabbah, and M. Sridharan. Translating imperative code to mapreduce. In *OOPSLA*, 2014.

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

[13] C. Smith and A. Albarghouthi. Mapreduce program synthesis. *PLDI*, June 2016.