

Actionable Performance Analyses

Marija Selakovic

marija.selakovic@huawei.com

London, 21.01.2020



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Software Performance

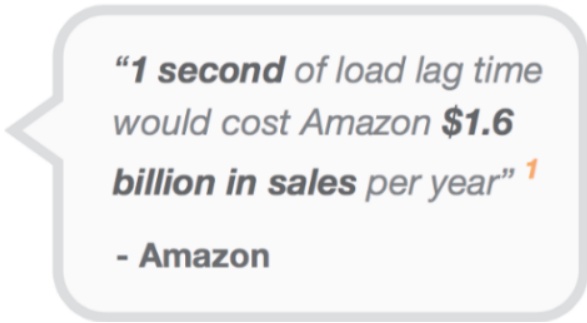
One of the most important aspects of software quality

- Efficiency
- Responsiveness
- Scaling
- Throughput
- User satisfaction

Software Performance

One of the most important aspects of software quality

- Efficiency
- Responsiveness
- Scaling
- Throughput
- User satisfaction



***"1 second** of load lag time
would cost Amazon **\$1.6**
billion in sales per year" ¹*
- Amazon

¹ <http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-1.6-billion-sales>

Software Performance

One of the most important aspects of software quality

- Efficiency
- Responsiveness
- Scaling
- Throughput
- User satisfaction

***"1 second** of load lag time would cost Amazon **\$1.6 billion in sales** per year" ¹*
- Amazon

***"A lag time of 400ms** results in **a decrease of 0.44% traffic** - In real terms this amounts to **440 million abandoned sessions/month** and a massive loss in advertising revenue for Google" ²*
- Google

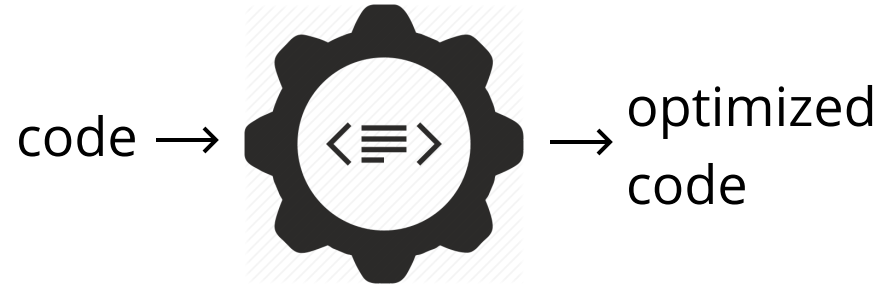
¹ <http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-1.6-billion-sales>

² <http://www.cedexis.com/blog/for-google-400ms-of-increased-page-load-time-results-in-044-lost-search-sessions/>

Approaches for Improving Software Performance



CPU profiling



Compiler optimizations

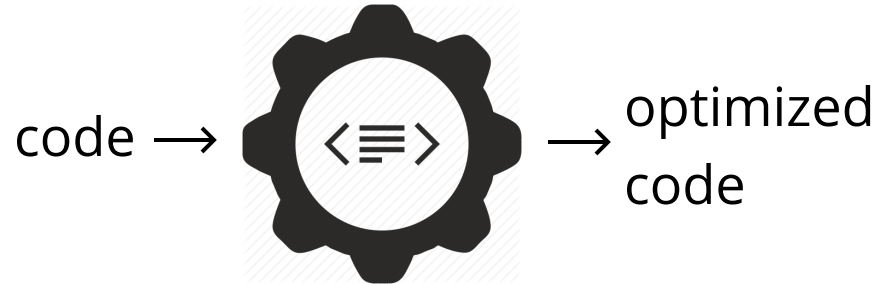


Manual tuning

Approaches for Improving Software Performance



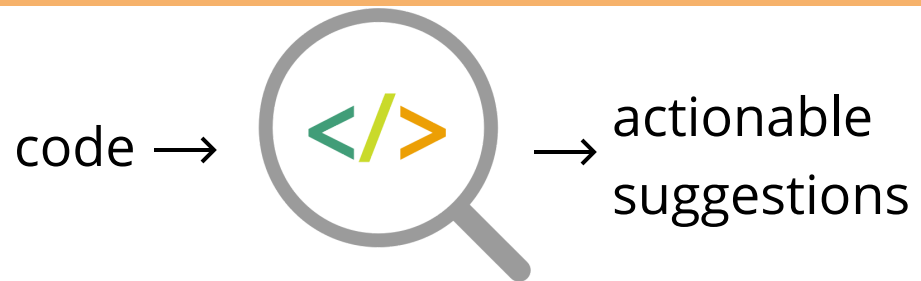
CPU profiling



Compiler optimizations



Manual tuning



Actionable Performance Analyses

Actionable Performance Analyses

- Suggest concrete code changes
- Demonstrate the impact of applying optimizations
- Optimizations that are:
 - *exploitable* - easy to understand and apply
 - *effective* - lead to significant performance improvements
 - *recurring* - applicable across multiple projects

This Talk

- Reordering opportunities [1]
- Method inlining in Big Data system [2]

This Talk

- Reordering opportunities [1]
- Method inlining in Big Data system [2]

An Actionable Performance Profiler for Optimizing the Order of Evaluations

Marija Selakovic
TU Darmstadt
Germany
m.selakovic@gmail.com

Thomas Glaser
TU Darmstadt
Germany
thomas.glaser@stud.tu-darmstadt.de

Michael Pradel
TU Darmstadt
Germany
michael@hinservariant.de

ABSTRACT

The efficiency of programs often can be improved by applying relatively simple changes. To find such optimization opportunities, developers either rely on manual performance tuning, which is time-consuming and requires expert knowledge, or on traditional profilers, which show where resources are spent but not how to optimize the program. This paper presents a profiler that provides actionable advice, by not only finding optimization opportunities but by also suggesting code transformations that exploit them. Specifically, we focus on optimization opportunities related to the order of evaluating subexpressions that are part of a decision made by the program. To help developers find such reordering opportunities, we present *DecisionProf*, a dynamic analysis that automatically identifies the optimal order, for a given input, of checks in logical expressions and in switch statements. The key idea is to assess the computational costs of all possible orders, to find the optimal order, and to suggest a code transformation to the developer only if reordering yields a statistically significant performance improvement. Applying *DecisionProf* to 43 real-world JavaScript projects reveals 52 beneficial reordering opportunities. Optimizing the code as proposed by *DecisionProf* reduces the execution time of individual functions between 2.5% and 59%, and leads to statistically significant application-level performance improvements that range between 2.5% and 6.5%.

CCS CONCEPTS

Software and its engineering — Software maintenance tools; Software testing and debugging

KEYWORDS

JavaScript, Dynamic analysis, Performance, Profiler

ACM Reference format:

Marija Selakovic, Thomas Glaser, and Michael Pradel. 2017. An Actionable Performance Profiler for Optimizing the Order of Evaluations. In *Proceedings of 20th International Symposium on Software Testing and Analysis*, Santa Barbara, CA, USA, July 2017. (ISSTA'17), 17 pages.
DOI: 10.1145/3092703.3092716

1 INTRODUCTION

Optimizing the performance of software is important in various domains, e.g., for achieving high throughput, energy efficiency, or low latency. In many cases, the performance of a program can be improved by applying relatively simple changes. To find such optimization opportunities, developers either rely on manual performance tuning, which is time-consuming and requires expert knowledge, or on traditional profilers, which show where resources are spent but not how to optimize the program. This paper presents a profiler that provides actionable advice, by not only finding optimization opportunities but by also suggesting code transformations that exploit them. Specifically, we focus on optimization opportunities related to the order of evaluating subexpressions that are part of a decision made by the program. To help developers find such reordering opportunities, we present *DecisionProf*, a dynamic analysis that automatically identifies the optimal order, for a given input, of checks in logical expressions and in switch statements. The key idea is to assess the computational costs of all possible orders, to find the optimal order, and to suggest a code transformation to the developer only if reordering yields a statistically significant performance improvement. Applying *DecisionProf* to 43 real-world JavaScript projects reveals 52 beneficial reordering opportunities. Optimizing the code as proposed by *DecisionProf* reduces the execution time of individual functions between 2.5% and 59%, and leads to statistically significant application-level performance improvements that range between 2.5% and 6.5%.

responsiveness, and user satisfaction. Even relatively small performance improvements (measured in milliseconds) in applications such as web sites or search engines can positively influence the page traffic and user experience.

However, detecting and exploiting optimization opportunities is a cumbersome task that often requires significant human effort. Fortunately, many programs suffer from performance bottlenecks where a relatively simple source code change can make the program significantly more efficient [19, 34]. The challenge is to find and exploit such easy-to-use optimization opportunities.

Currently, there are three kinds of approaches to optimize performance. First, compiler optimizations automatically transform a program to a semantically equivalent yet more efficient program. Despite being very powerful for particular classes of optimizations, many other promising optimization opportunities are beyond the capabilities of a typical compiler. The main reason is that the compiler cannot ensure that a transformation preserves the semantics, a problem that is especially relevant for hard-to-analyze languages, such as JavaScript. Second, to complement compiler optimizations, developers use CPU [12] and memory profilers [18] to identify those code locations that use most resources. More recent approaches identify performance bottlenecks based on their symptoms, such as memory bloat [42], inefficient loops [29], and JIT unfairness [11]. While useful to understand why code is slow, these approaches do not show developers how to optimize the code. Finally, developers often fall back on manual performance tuning, which can be effective but is time-consuming and often requires expert knowledge.

This paper presents a novel automated approach to support developers in optimizing their programs, called *actionable performance profiling*. The key idea is to not only pinpoint where and why time is spent, but to also suggest concrete code transformations that speed up the code. A profiler following this idea is actionable in the sense that the developer can take immediate action based on the profiler's suggestions, by deciding whether to apply a suggested transformation. The reason why the profiler does not fully automatically optimize the program, as a compiler would, is that it does not guarantee to preserve the semantics, enabling it to address optimizations out-of-reach for compilers.

As motivating examples, Figure 1 shows two non-trivial to detect but easy-to-exploit optimization opportunities in popular JavaScript projects. The code in Figure 1a checks three conditions: whether a regular expression matches a given string, whether the value stored in `match[1]` is defined and whether the value of `arg` is higher or equal to zero. This code can be optimized by swapping the first two checks (Figure 1b) because checking the first condition is more expensive than checking the second condition. After this change, when `match[1]` is evaluated to `false`, the overall execution time of evaluating the logical expression is reduced by the time needed to perform the regular expression matching. The second example,

Cross-Language Optimizations in Big Data Systems: A Case Study of SCOPE

Marija Selakovic
TU Darmstadt
Germany
m.selakovic89@gmail.com

Michael Barnett, Madan Musuvathi, Todd Mytkowicz
Microsoft Research
USA
mbarnett.madanm.todd@microsoft.com

ABSTRACT

Building scalable big data programs currently requires programmers to combine relational (SQL) with non-relational code (Java, C#, Scala). Relational code is declarative — a program describes what the computation is and the compiler decides how to distribute the program. SQL query optimization has enjoyed a rich and fruitful history, however, most research and commercial optimization engines treat non-relational code as a black-box and thus are unable to optimize it.

This paper empirically studies over 3 million SCOPE programs across five data centers within Microsoft and finds programs with non-relational code take between 45-70% of data center CPU time. We further explore the potential for SCOPE optimization by generating more native code from the non-relational part. Finally, we present 6 case studies showing that triggering more generation of native code in these jobs yields significant performance improvement: optimizing just one portion resulted in as much as 25% improvement for an entire program.

ACM Reference format:

Marija Selakovic and Michael Barnett, Madan Musuvathi, Todd Mytkowicz. 2018. Cross-Language Optimizations in Big Data Systems: A Case Study of SCOPE. In *Proceedings of 40th International Conference on Software Engineering*, Software Engineering in Practice Track, Gothenburg, Sweden, May 27-June 1, 2018. (ICSE-SEIP '18), 10 pages.
https://doi.org/10.1145/3185159.3185158

1 INTRODUCTION

Large-scale data-processing frameworks, such as MapReduce [10], SCOPE [4], Hadoop [12], Spark [34], have become an integral part of computing today. One reason for their immense popularity is that they provide a programming model that greatly simplifies the distribution and fault-tolerance of big data processing. For instance, frameworks like SCOPE and Spark provide a SQL-like declarative interface for specifying the relational skeleton of data-processing jobs while providing extensibility by supporting expressions and functions written in general-purpose languages like C#, Java, or Scala.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be retained. No copying may be made for general distribution, for advertising or promotional purposes, for creating new collective works, or for resale. Copyrights for components of this work owned by others than ACM must be retained. No copying may be made for general distribution, for advertising or promotional purposes, for creating new collective works, or for resale. Request permissions from permissions@acm.org. ICSE-SEIP '18, May 27-June 1, 2018, Gothenburg, Sweden © 2018 Association for Computing Machinery. DOI: 10.1145/3185159.3185158

The relational aspect is crucial: it is what enables the automatic parallelization for efficiently scaling out to arbitrary amounts of data. Big data systems assume that the non-relational part is written carefully enough so that it does not violate the assumptions needed for automatic parallelization: e.g., programmers must write their non-relational logic to be deterministic and insensitive to the ordering of the input.

However, these systems are known to lag far behind traditional database systems in runtime efficiency [21, 25], primarily because of the flexibility of the programming model they support. For instance, a key bottleneck in Spark is neither the disk nor the network, but the time spent by the CPU on compression/decompression of data, serialization/deserialization of the input into/from Java objects, and the JVM garbage collection [27]. SCOPE, described more fully in Section 2, supports a hybrid native (C++) and C# runtime partly to alleviate this overhead. Like SCOPE, Hadoop Streaming lets programmers write programs in a mix of languages [2]. Our analysis shows that this cross-language interaction (in SCOPE, between the native and C# runtimes) is a significant cost in the overall system. Equally importantly, the presence of non-relational code blocks the powerful relational optimizations implemented in these data-processing runtimes, e.g. [16].

The goal of this work is to study and better understand the key performance bottlenecks in modern data-processing systems, and demonstrate the potential for cross-language optimizations. While this paper is primarily about SCOPE, we believe our results and optimizations generalize to other data-processing systems. SCOPE is the key data-processing system used at Microsoft running at least half a million jobs daily on several Microsoft data centers. Figure 1 shows a simple example of a SCOPE program (hereafter referred to as a script) that interleaves relational logic with C# expressions. In Figure 1a, the predicate in the WHERE clause is subject to two potential optimizations:

- (1) The optimizer may choose to promote one (or both) of the conjuncts to an earlier part of the script, especially if either A or B are columns used for partitioning the data. This can dramatically reduce the amount of data needed to be transferred across the network.
- (2) The SCOPE compiler has a set of methods that it considers to be intrinsic. An intrinsic is a .NET method for which the SCOPE runtime has a semantically equivalent native function, i.e., implemented in C#. For instance, the method `String.IsNullOrEmpty` checks whether its argument is either null or else the empty string. The corresponding native method is able to execute on the native data encoding which

[1] Selakovic et al. *An Actionable Performance Profiler for Optimizing the Order of Evaluations* (ISSTA'17)

[2] Selakovic et al. *Cross-Language Optimizations in Big Data Systems: A Case Study of SCOPE* (ICSE-SEIP'18)

Inefficient Order of Evaluations

expensiveAndUnlikely() && cheapAndLikely()

Inefficient Order of Evaluations



expensiveAndUnlikely() && cheapAndLikely()

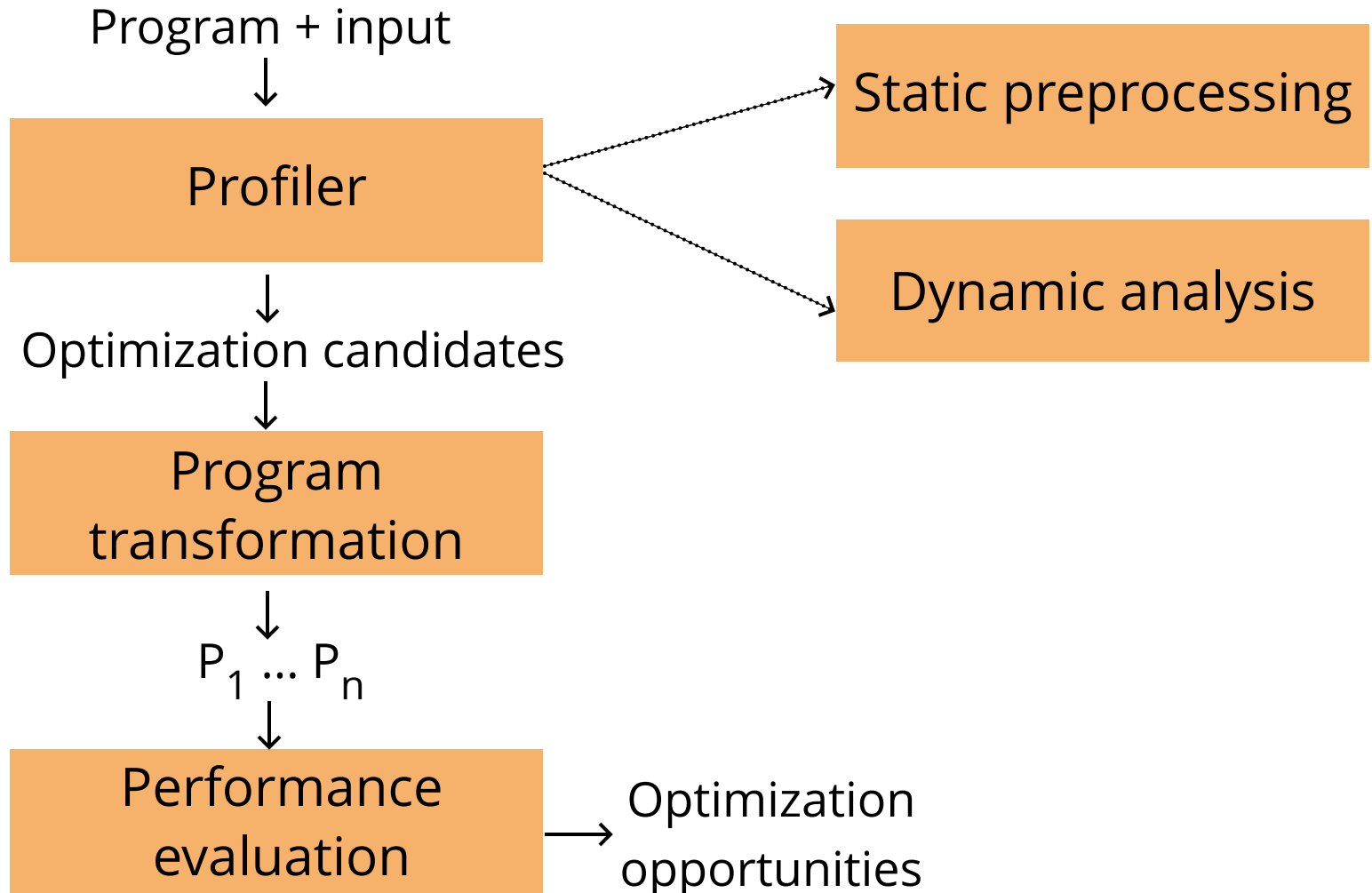
The diagram consists of two large, curved orange arrows forming a circle around the text. The top arrow points from the left side of the text to the right, and the bottom arrow points from the right side of the text back to the left, indicating a mutual dependency or a loop in the evaluation process.

Inefficient Order of Evaluations

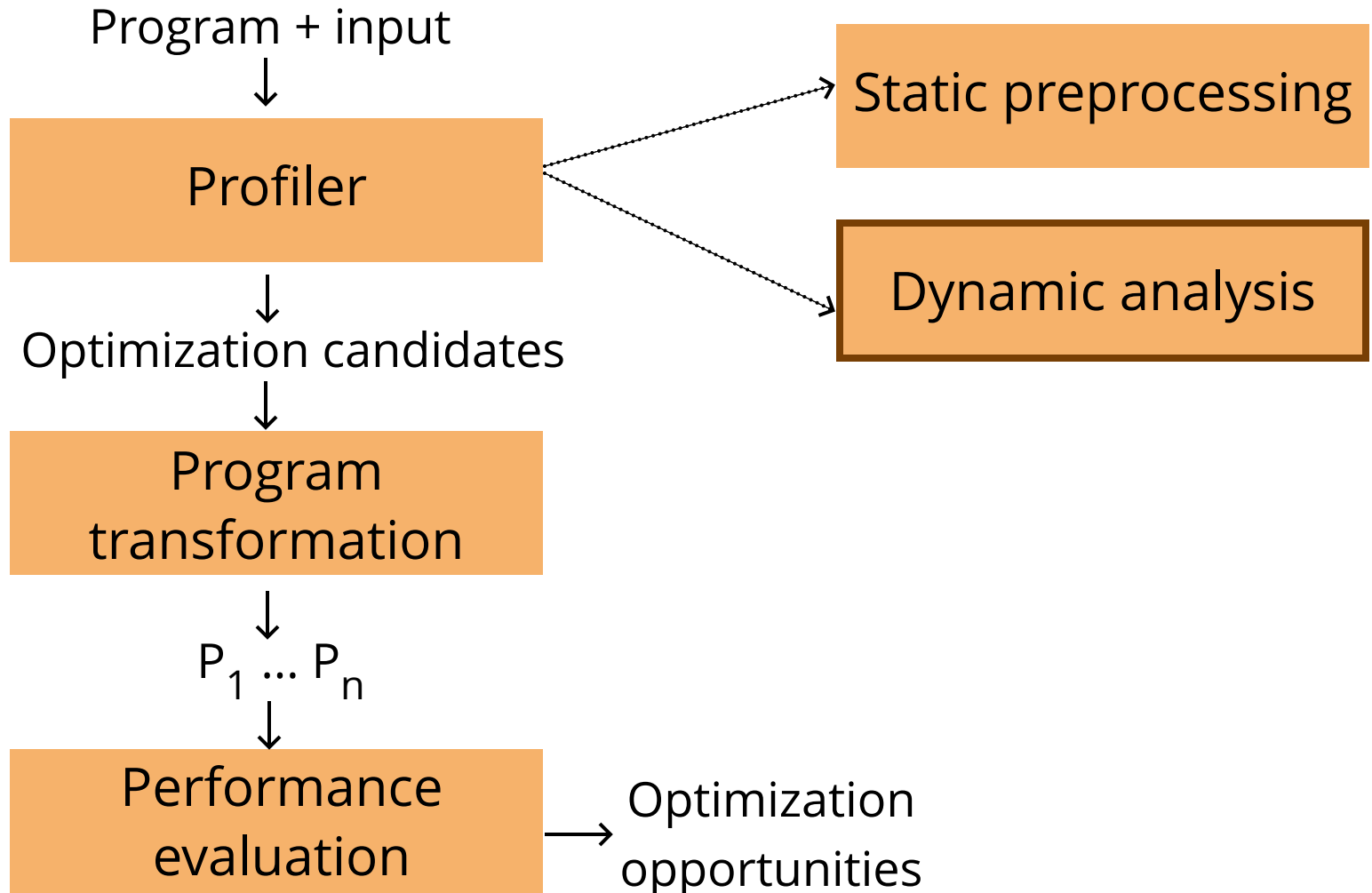


- Analysis of all *conditions* in in logical expressions or switch statements
- Assessment of the computational cost
- Safe to apply and beneficial optimizations

DecisionProf: An Analysis for Optimizing Orders of Evaluations



DecisionProf: An Analysis for Optimizing Orders of Evaluations

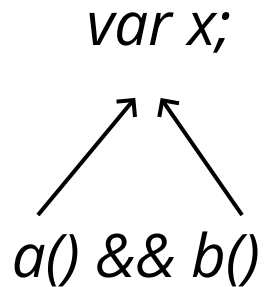


Dynamic Analysis

- Collecting runtime data:
 - Cost - number of branching point
 - Value - true/false
- Assessing the optimal order

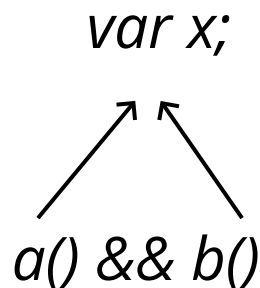
Dynamic Analysis

- Collecting runtime data:
 - Cost - number of branching point
 - Value - true/false
- Assessing the optimal order



Dynamic Analysis

- Collecting runtime data:
 - Cost - number of branching point
 - Value - true/false
- Assessing the optimal order

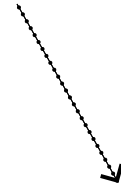
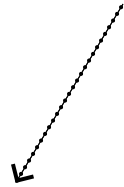


a() && b()

cost	value	cost	value
c_{a1}	v_{a1}	c_{b1}	v_{b1}
c_{a2}	v_{a2}	c_{b2}	v_{b2}
c_{a3}	v_{a3}	c_{b3}	v_{b3}

Dynamic Analysis: Example

_.isNumber(input) && isNaN(input)



Execution 1

cost	value
3	true
2	true
4	true

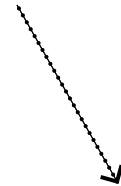
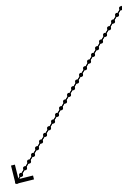
Execution 2

Execution 3

cost	value
1	false
1	false
1	true

Dynamic Analysis: Example

_.isNumber(input) && isNaN(input)



Execution 1

cost	value
3	true
2	true
4	true

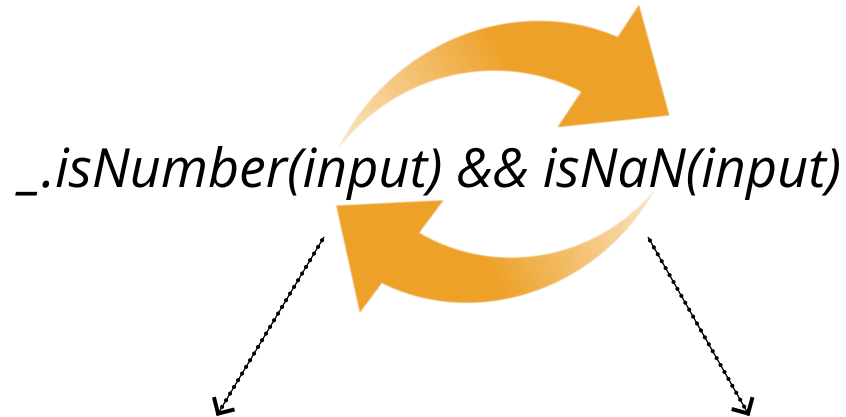
Execution 2

Execution 3

cost	value
1	false
1	false
1	true

Overall cost = 12

Dynamic Analysis: Example



Execution 1

cost	value
3	true
2	true
4	true

Execution 2

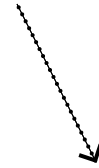
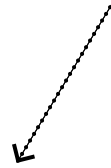
Execution 3

cost	value
1	false
1	false
1	true

Overall cost = 12

Dynamic Analysis: Example

*isNaN(input) && _.isNumber(input)*¹



Execution 1

Cost	Value
1	false
1	false
1	true

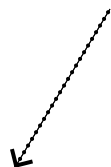
Execution 2

Execution 3

Cost	Value
3	true
2	true
4	true

Dynamic Analysis: Example

*isNaN(input) && _.isNumber(input)*¹



Execution 1

Cost	Value
1	false
1	false
1	true

Execution 2

Execution 3

Cost	Value
3	true
2	true
4	true

Overall cost = 7

¹ See pull request #2496 of Underscore.js

Pruning Non-Commutative Conditions

Non-commutative conditions: change program semantics

Two approaches:

Static: known patterns

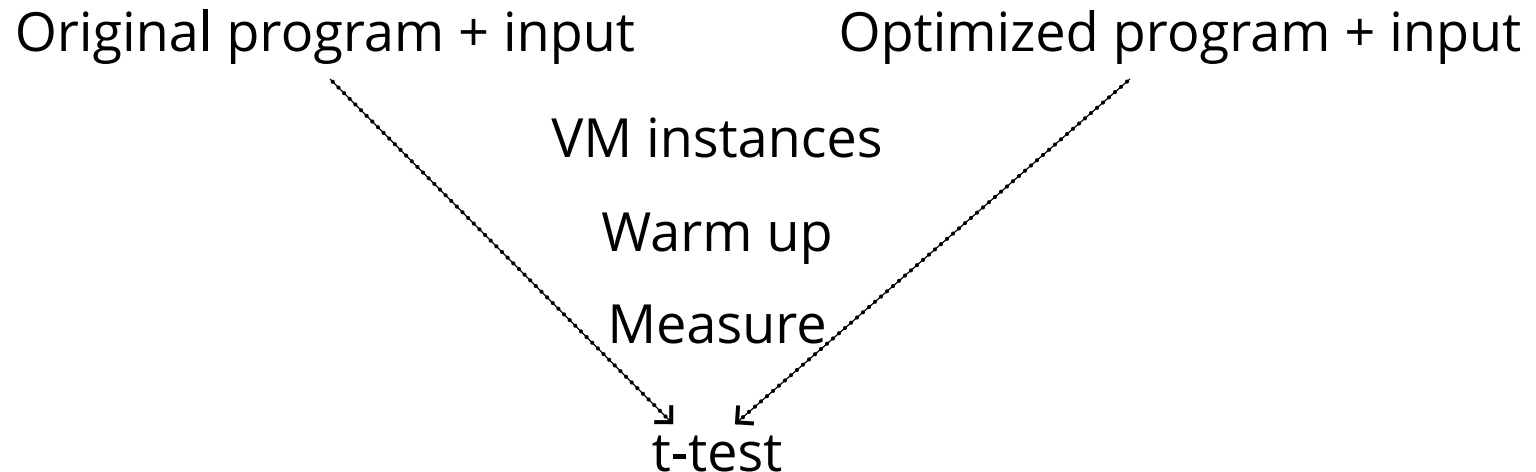
```
1  if (a && a.x) {...}  
2  
3  y = x || "abc"
```

Dynamic: write to the same
memory location

```
1  var x = 0;  
2  function a(){  
3    x++;  
4  }  
5  
6  if (a() && x) {...}
```

Performance Evaluation

- Program transformation for each optimization candidate
- Methodology by Georges et al. [1]



DecisonProf: Evaluation

- Subject programs:
 - 9 JavaScript libraries and test suites
 - 34 benchmarks from JetStream suite
- Results:
 - 23 opportunities across libraries
 - 29 opportunities across benchmarks
 - Performance improvements: **2.5% - 59%** (function level) and **2.5% - 6.5%** (application level)

Examples of Reordering Opportunities

Cheerio library:

```
//code before
isTag (elem) && elems.indexOf(elem) === -1

//code after
elems.indexOf(elem) === -1 && isTag (elem)
```

tests: 26%, 34%

Gbemu benchmark:

```
//code before
numberType != "float32" && GameBoyWindow.opera
    && this.checkForOperaMathBug ()

//code after
GameBoyWindow.opera && numberType != "float32"
    && this.checkForOperaMathBug ()
```

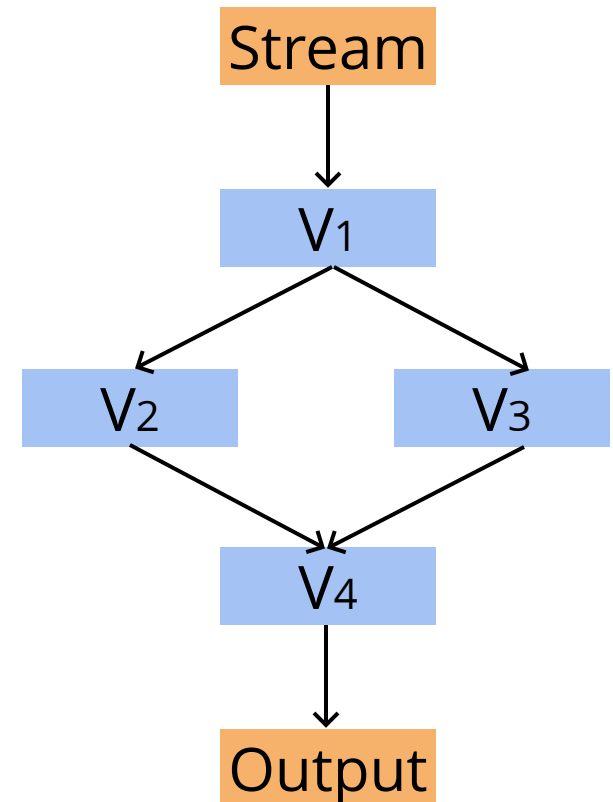
application:
5.8%

DecisionProf: Summary

- The first profiler to detect inefficient orders of evaluations
- Simple and easy to exploit optimizations
- Suggests program refactorings
- Guaranteed improvements for given inputs

Cross-language Optimizations in Big Data Systems

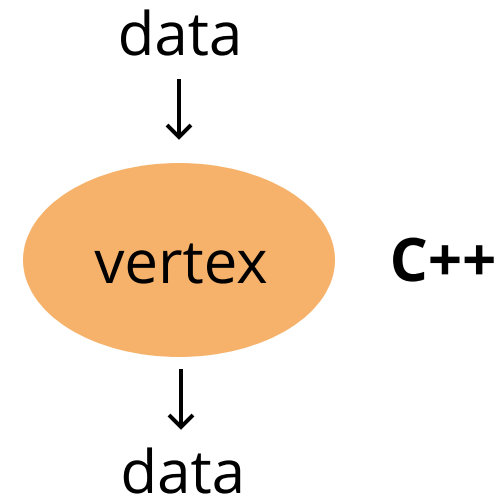
- **SCOPE** - **S**tructured **C**omputations
Optimized for **P**arallel **E**xecutions
- Relational (SQL) + non-relational (C#)
- SCOPE job is DAG where:
 - Vertices - processes
 - Edges - data flows



Performance Problem in SCOPE

- Cross runtime interaction
- Intrinsic vs non-intrinsic

```
data =  
  SELECT *  
  FROM inputStream;
```



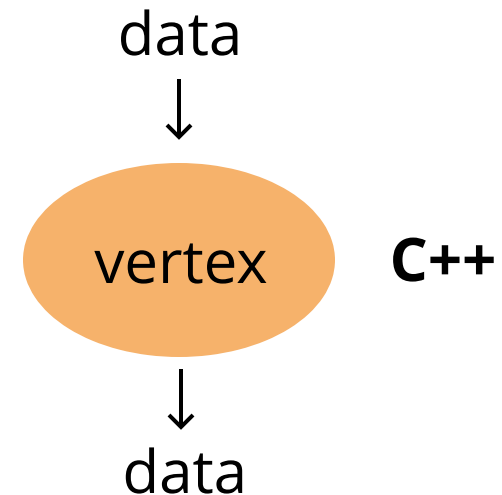
Performance Problem in SCOPE

- Cross runtime interaction
- **Intrinsics** vs non-intrinsics

```
data =  
    SELECT *  
    FROM inputStream;
```

```
data =  
    SELECT *  
    FROM inputStream  
    WHERE !String.IsNullOrEmpty(A);
```

→ intrinsic
(has c++ impl.)

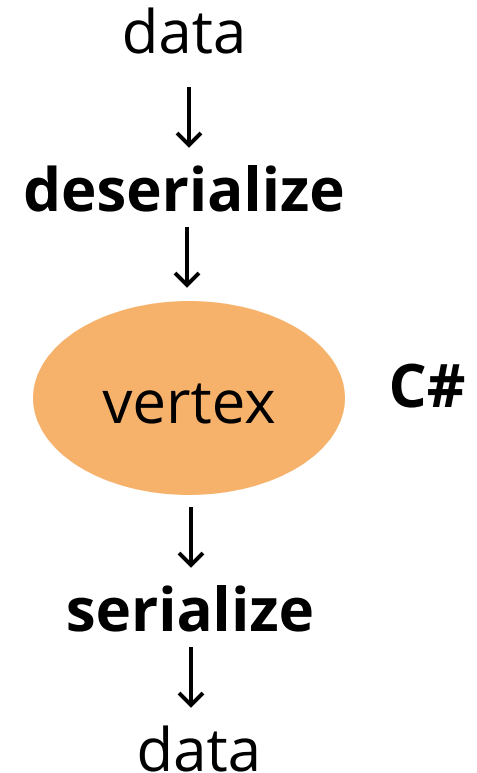


Performance Problem in SCOPE

- Cross runtime interaction
- Intrinsic vs **non-intrinsic**

```
data =  
    SELECT *  
    FROM inputStream  
    WHERE A.Equals('ColumnA');
```

→ non-
intrinsic



Method Inlining: Example

- Replacing function call with the body of a function

```
data =                                     C#
    SELECT *
    FROM inputStream
    WHERE filter(JobID);

#CS

bool filter (string s) {
    return (!string.IsNullOrEmpty(s)
        && s.StartsWith("07"));
}

#ENDCS
```


Method Inlining: Example

- Replacing function call with the body of a function

```
data =  
    SELECT *  
    FROM inputStream  
    WHERE !string.IsNullOrEmpty(JobID)  
           && JobID.StartsWith("07");
```

C++

Method Inlining: Example

- Replacing function call with the body of a function

```
data =  
    SELECT *  
    FROM inputStream  
    WHERE !string.IsNullOrEmpty(JobID)  
           && JobID.StartsWith("07");
```

C++

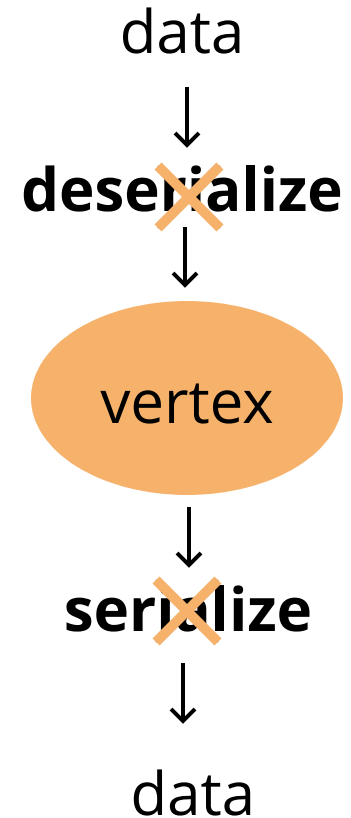
~ 4X faster

Static Analysis

Goal: C# to C++ translation for a vertex

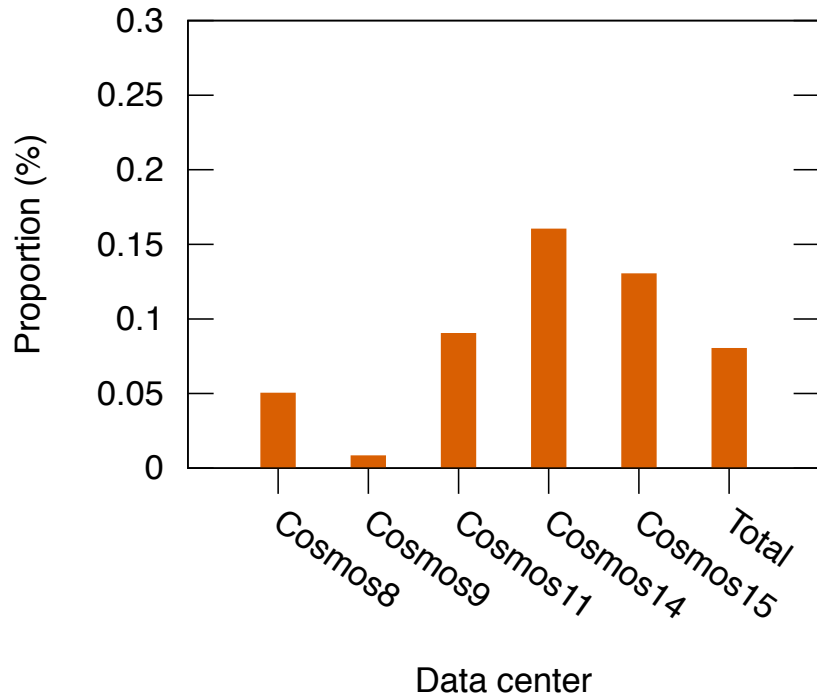
Inlineable methods:

- Only calls to intrinsics
- No loops
- No try-catch blocks
- No new and cast operations
- No arguments passed by reference



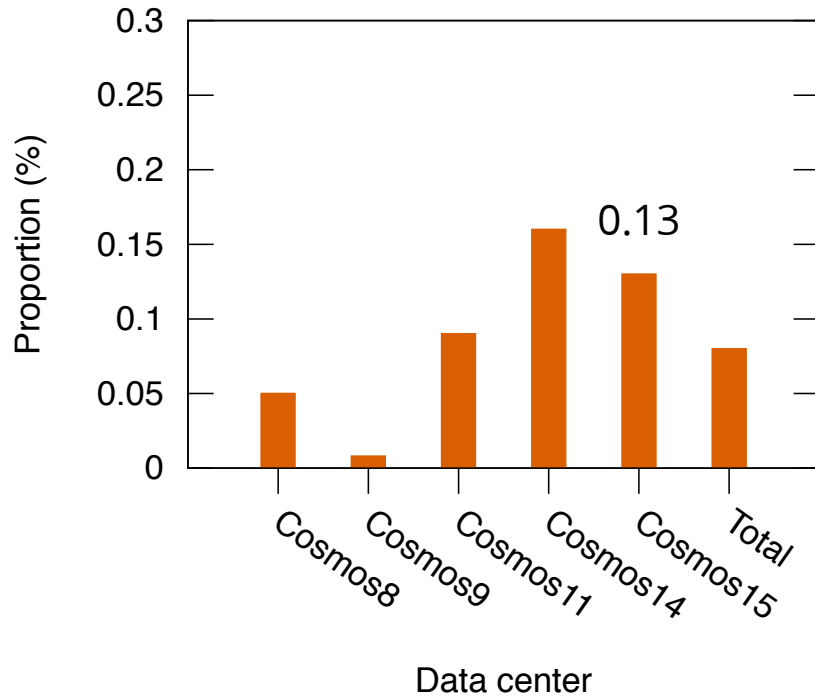
Static Analysis: Evaluation

Optimizable vertices:



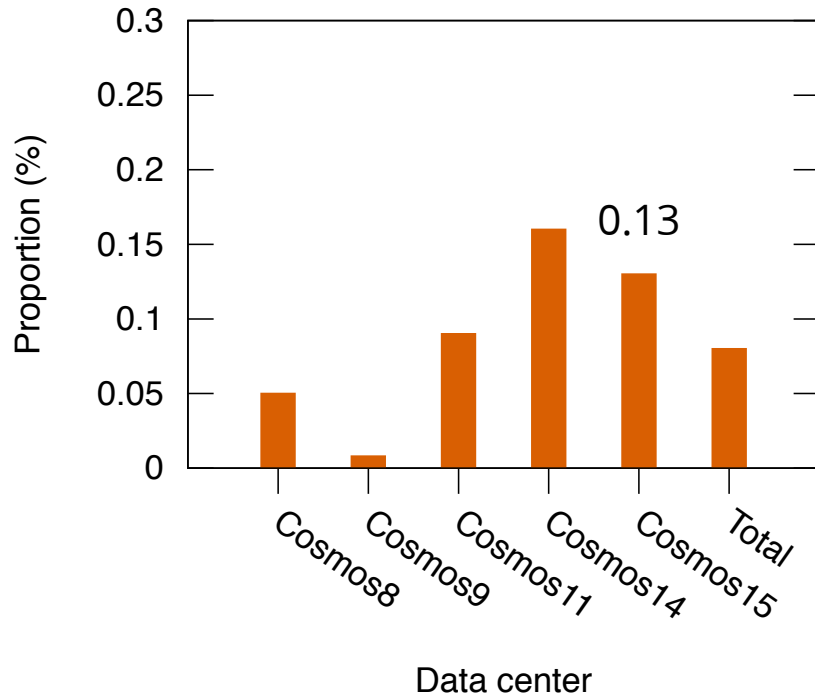
Static Analysis: Evaluation

Optimizable vertices:



Static Analysis: Evaluation

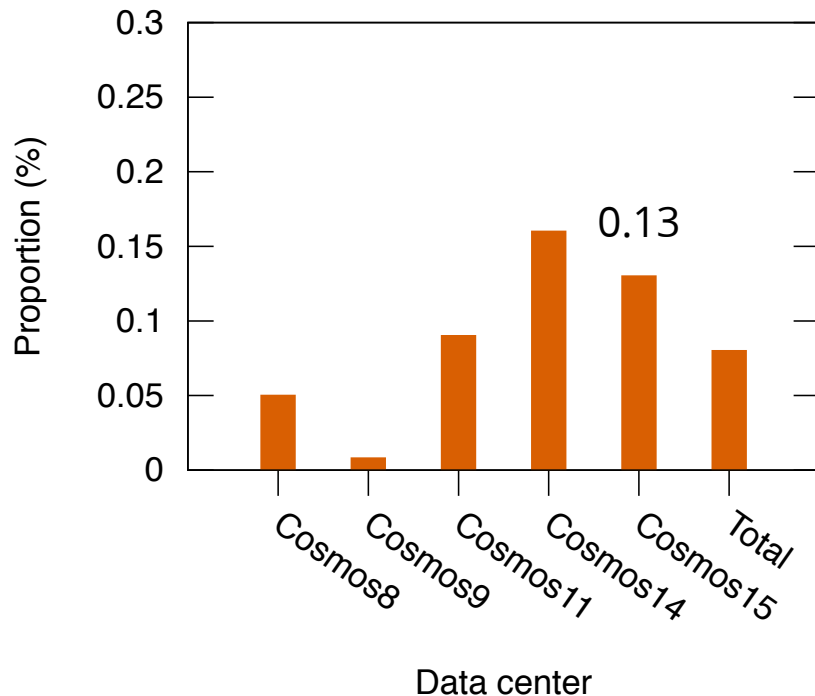
Optimizable vertices:



0.13% ~ 40,000 hours

Static Analysis: Evaluation

Optimizable vertices:



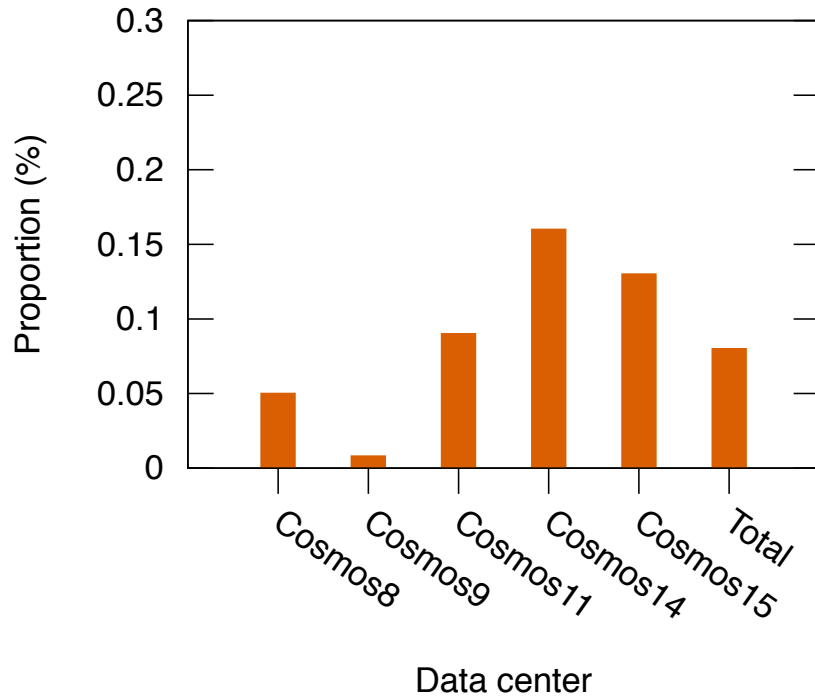
Case studies:

Job	CPU Time	Throughput
A	23%	30%
B	no change	no change
C	25%	38%
E	4.7%	5%
F	no change	115%

0.13% ~ 40,000 hours

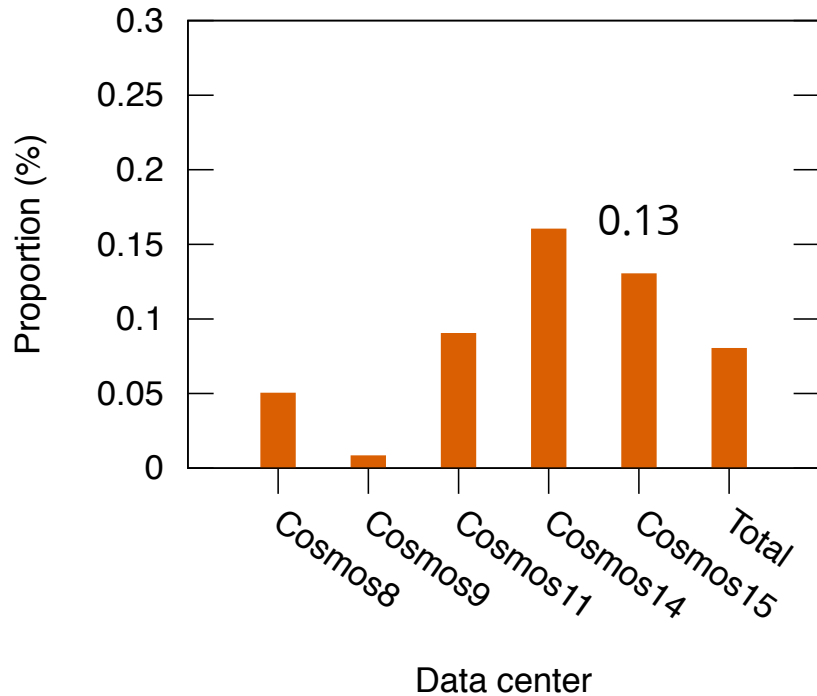
Static Analysis: Evaluation

Optimizable vertices:



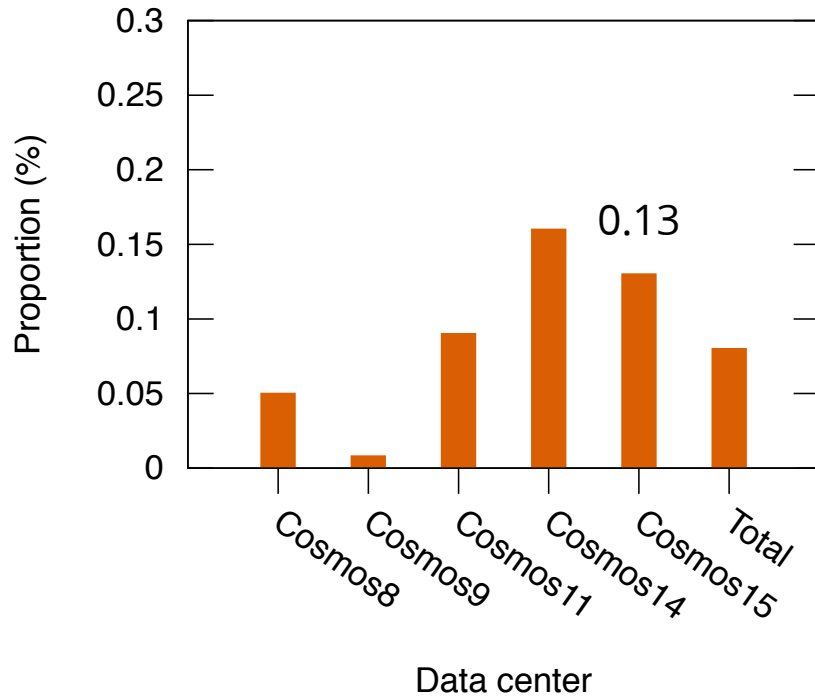
Static Analysis: Evaluation

Optimizable vertices:



Static Analysis: Evaluation

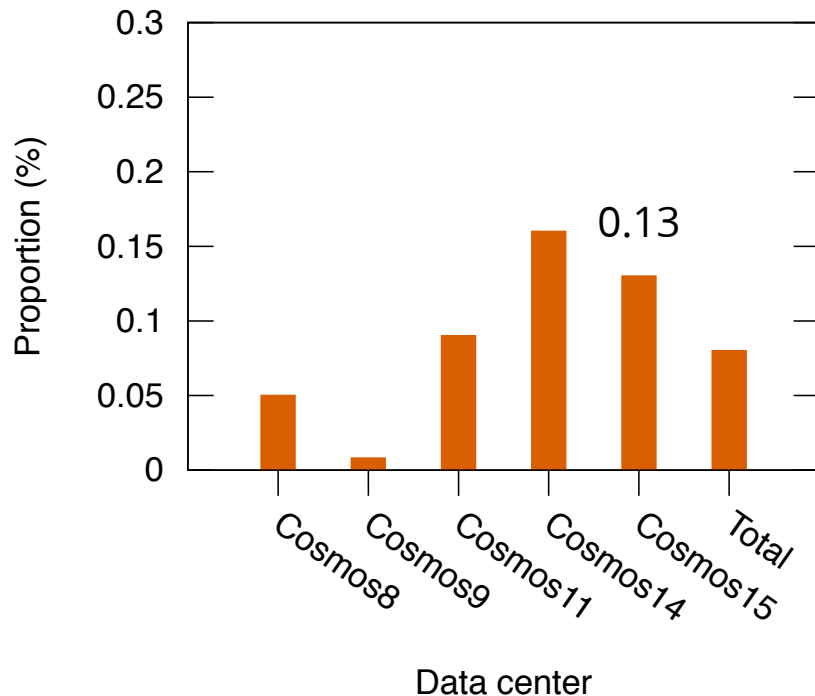
Optimizable vertices:



0.13% ~ 40,000 hours

Static Analysis: Evaluation

Optimizable vertices:



Case studies:

Job	CPU Time	Throughput
A	23%	30%
B	no change	no change
C	25%	38%
E	4.7%	5%
F	no change	115%

0.13% ~ 40,000 hours

Case Study in SCOPE: Example

```
//optimization opportunity
```

```
SELECT url_id, url, t_url_id,  
        CommonMethod.GetTargetUrl(url,t_url) AS t_url  
FROM SSTREAM @VLPMMapSS;
```

```
//optimized code
```

```
SELECT url_id, url, t_url_id,  
        string.IsNullOrEmpty(t_url)?url:t_url AS t_url  
FROM SSTREAM @VLPMMapSS;
```

Vertex level improvement: 42%

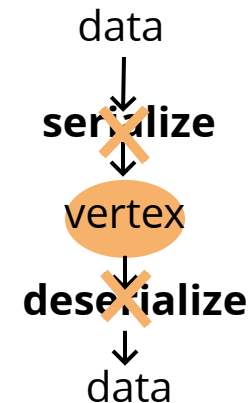
Job level improvement: 25%

Cross-language Optimizations in Big Data Systems: Summary

- Static analysis for method inlining opportunities
- Only optimizations that reduce cross-runtime interactions
- Large scale evaluation

Conclusions

- Actionable performance analyses
- Easy to exploit classes of optimizations
- Future work: automatically inferring optimization patterns across and within *different domains*



Huawei Dresden Research Center

Research topics:

- Design and development of embedded systems
- Program analysis and formal verification
- Software testing: fuzzing and automated test case generation



Huawei Dresden Research Center

Research topics:

- Design and development of embedded systems
- Program analysis and formal verification
- Software testing: fuzzing and automated test case generation

We are hiring!



Questions?

DecisionProf: Static Preprocessing

```
startDecision;  
  
startCheck: a(); endCheck();  
startCheck: b(); endCheck();  
startCheck: c(); endCheck();  
  
endDecision;  
  
if (a() && b() && c()) {  
    ...  
}
```

- Hoists leaf expressions
- Beginning and end of each decision
- Beginning and end of each checks

DecisionProf: Safe Check Evaluation

Collect and undo all writes to variables and object properties that may affect code after check evaluation

```
var x = 0;

function a () {
  x++;
  var y=1;
  .....
}

startCheck: a();
startCheck: b();

//reset all side effects
if (a () && b()) ...
```

DecisionProf: Safe Check Evaluation

Collect and undo all writes to variables and object properties that may affect code after check evaluation

```
var x = 0;

function a () {
  x++;
  var y=1;
  .....
}

startCheck: a();
startCheck: b();

//reset all side effects
if (a () && b()) ...
```

← *write to x affects
program state*

DecisionProf: Safe Check Evaluation

Collect and undo all writes to variables and object properties that may affect code after check evaluation

```
var x = 0;

function a () {
  x++;
  var y=1;
  .....
}

startCheck: a();
startCheck: b();

//reset all side effects
if (a () && b()) ...
```

← *write to x affects
program state*

← *program state is changed
outside normal execution*

DecisionProf: Safe Check Evaluation

Collect and undo all writes to variables and object properties that may affect code after check evaluation

```
var x = 0;

function a () {
  x++;
  var y=1;
  .....
}

startCheck: a();
startCheck: b();

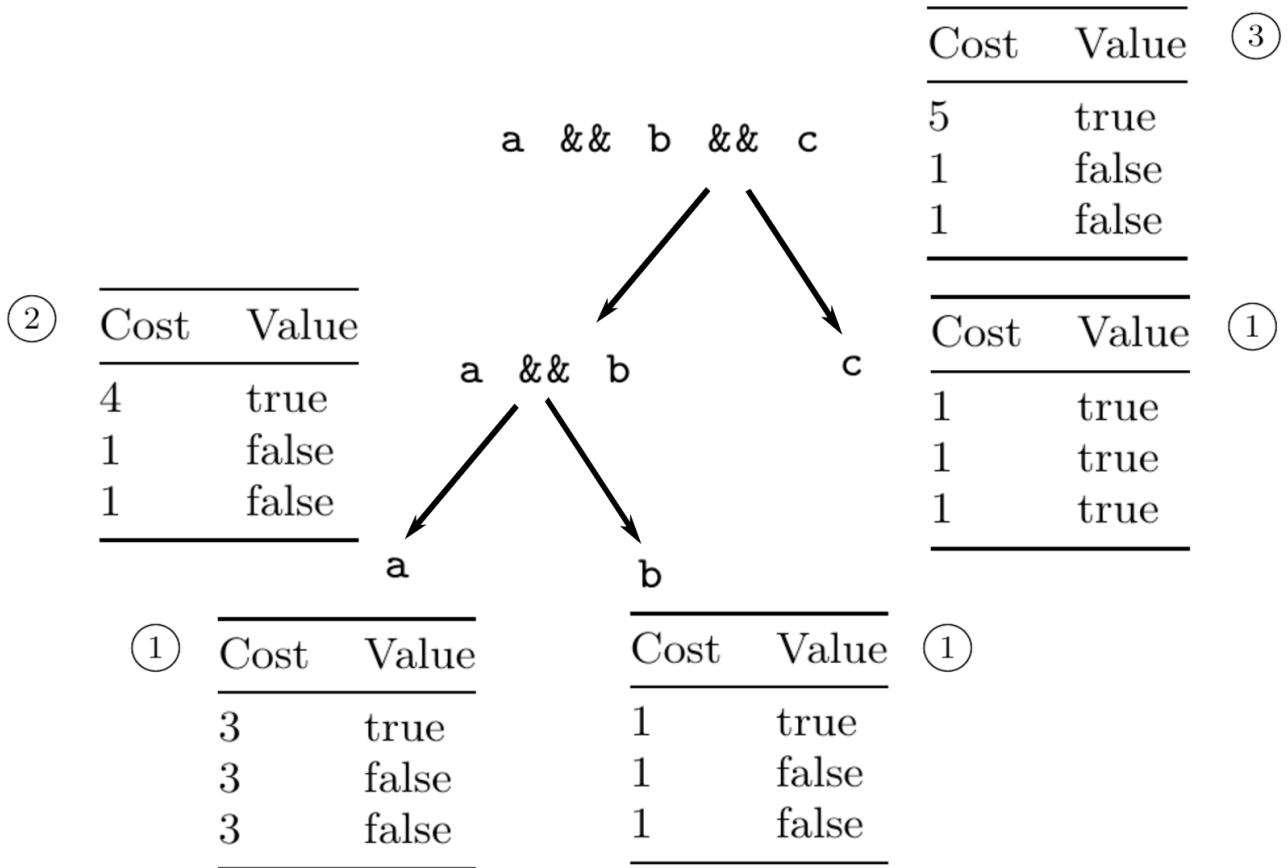
//reset all side effects
if (a () && b()) ...
```

← *write to x affects
program state*

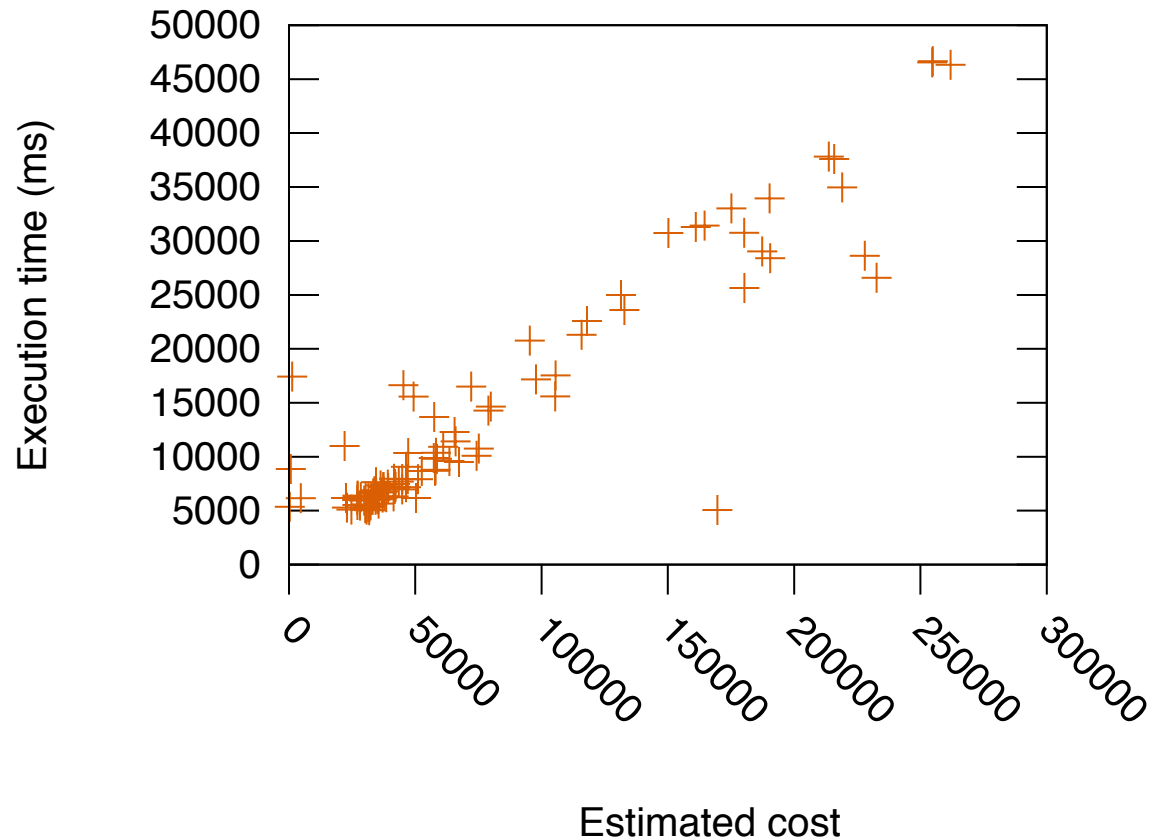
← *program state is changed
outside normal execution*

← *dynamically execute x = 0;*

DecisionProf: Finding Optimal Order

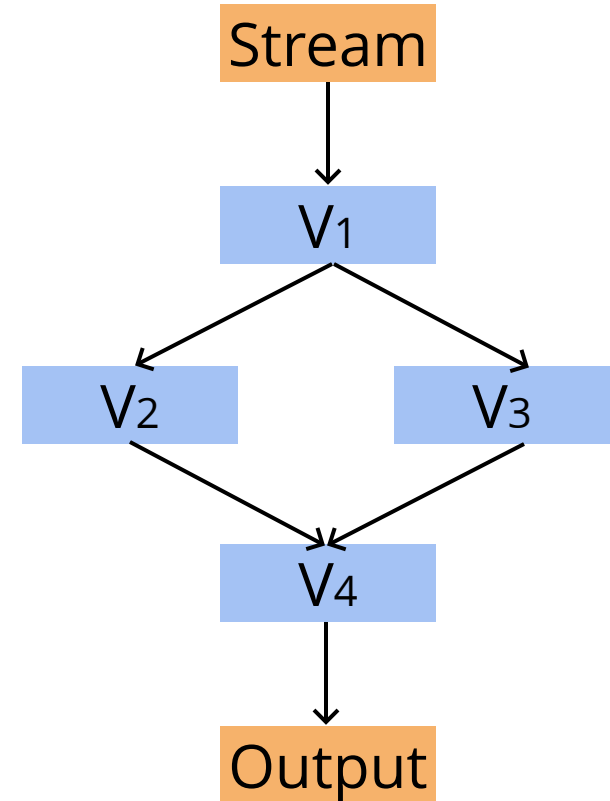


DecisionProf: Actual vs Estimated Cost

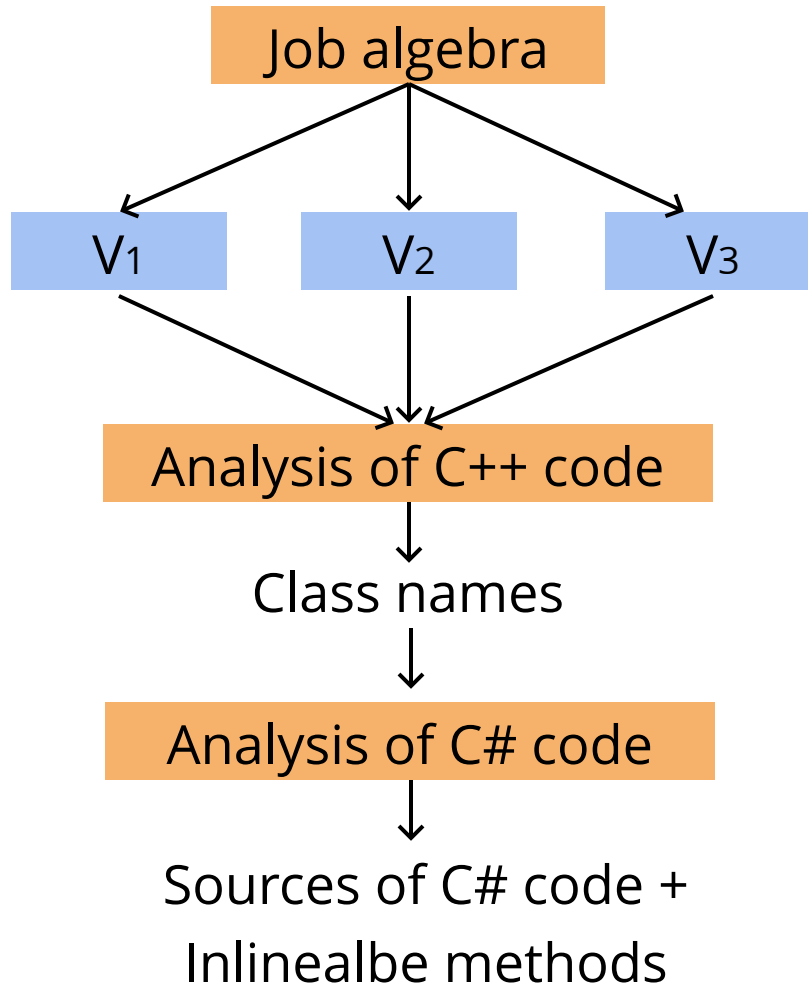


Job Artifacts

- Job Algebra
 - Vertices - processes
 - Edges - data flow
- Runtime statistics
- Script source code
- Generated C# and C++ code



Profiling Infrastructure



- .NET framework methods
- User-written methods
- Processors and reduces

Native vs Non-Native Time

