Pontifical Catholic University of Rio Grande do Sul (PUCRS),
Faculty of Informatics (FACIN), Computer Science Graduate Program (PPGCC)
Parallel Application Modeling Group (GMAP)

A Unified MapReduce Domain-Specific Language for Distributed and Shared Memory Architectures

Daniel Adornes, Dalvan Griebler, Cleverson Ledur, Luiz Gustavo Fernandes

SEKE 2015

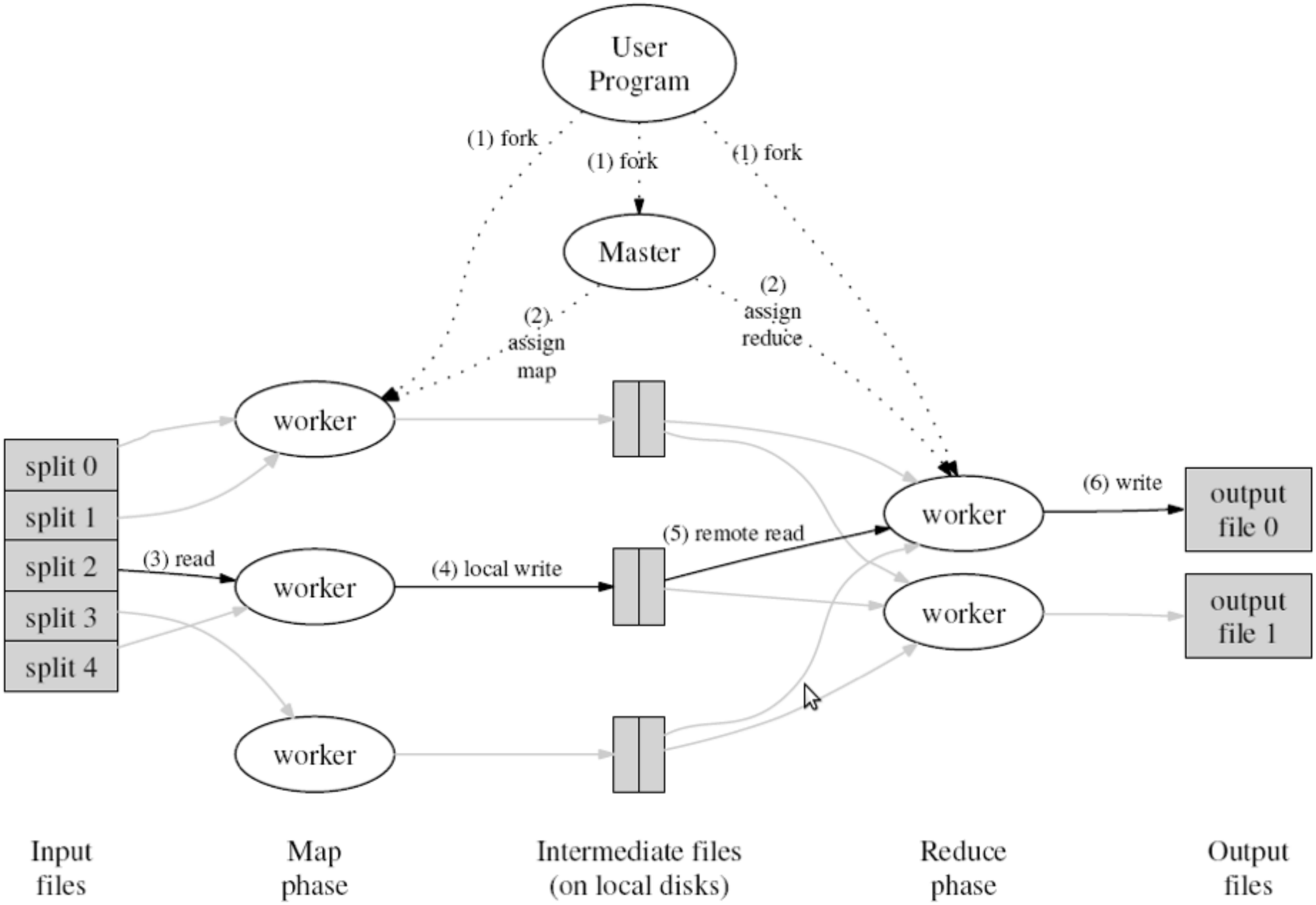
Wyndham Pittsburgh University Center

Introduction

- New challenges for software engineers and developers
- Instead of being faster, computer architectures are more parallel
- Depending on the amount of data to be processed, local memory is not enough and distributed systems become a necessity
- Programming interfaces become prone to excessive complexity

MapReduce abstract model

- 2004 - Google introduced the MapReduce abstract model, based on two operations, *map* and *reduce*, originally from functional programming languages
- Simplicity and scalability for developing software to process large datasets
- Aimed, but **not limited**, to distributed environments



MapReduce job execution flow - Dean and Ghemawat (2004, p. 3)

MapReduce abstract model

"Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines"

Dean and Ghemawat (2004, p. 3)

MapReduce implementations

- **2004** - MapReduce original publication
- **2005** - Hadoop
- **2007** - Phoenix
- **2009** - Phoenix Rebirth
- **2010** - Tiled-MapReduce
- **2011** - Phoenix++

MapReduce implementations

- **2004** - MapReduce original publication
- **2005** - Hadoop
- **2007** - Phoenix
- **2009** - Phoenix Rebirth
- **2010** - Tiled-MapReduce
- **2011** - Phoenix++

Hadoop interface components

- Language: Java
- Mapper and Reducer
- Writable
- InputFormatReader
- RecordReader

Phoenix++

- Language: C++
- Efficient key-value storage
  - Modular storage options: ***Containers***
- Effective ***combiner*** stage
  - Aggressively call combiner after every map emit

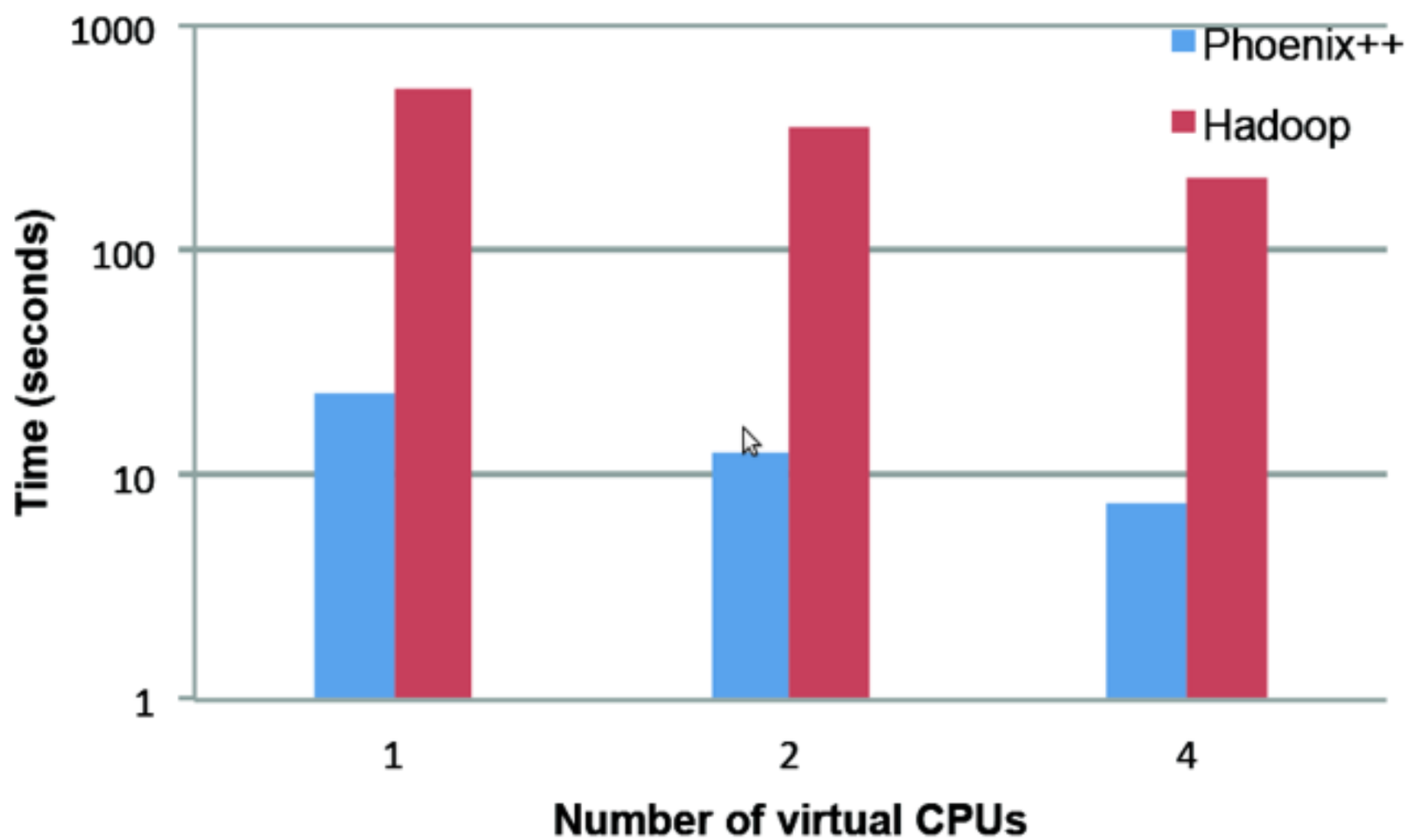
Phoenix++

- Modular storage options
  - Specialized *Container* types

Key Distribution	Sample applications	*Container* type
:	*Word Count*	variable-size hash table
***:k**	*Histogram, Linear Regression, K-means, String Match*	array with fixed mapping
1:1	*Matrix Multiplication, PCA*	shared array

Performance comparison

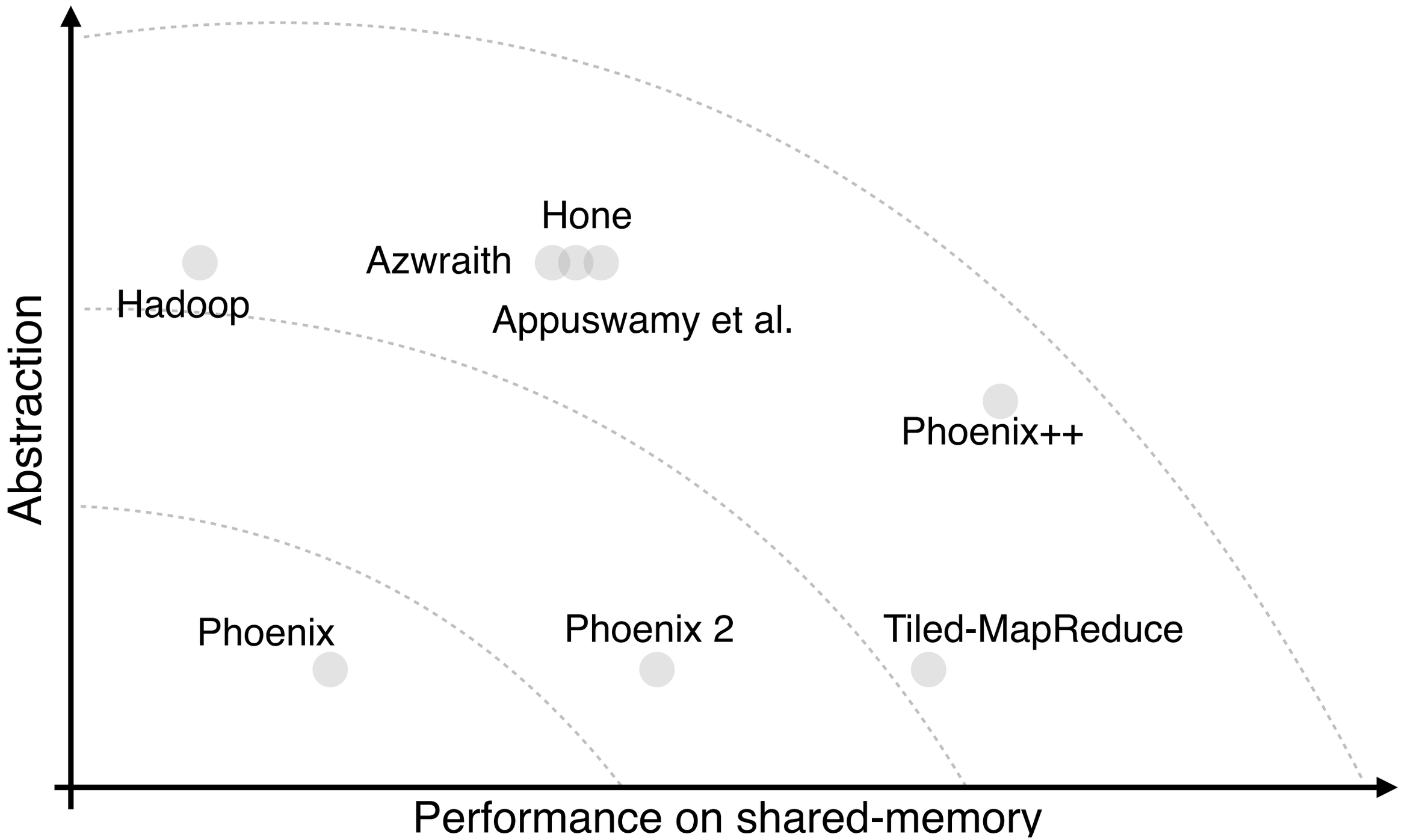
Hadoop vs Phoenix++



Experiment of 1 GB word count using Phoenix++ and Hadoop on a multi-core architecture. The y-axis is in a **logarithmic** scale.

Related work

- Important researches on improving Hadoop for high performance at the single-node level.
- No research was found on building a unified MapReduce programming interface.


Hone
Azwraith
Hadoop
Appuswamy et al.
Phoenix++
Phoenix
Phoenix 2
Tiled-MapReduce
Abstraction
Performance on shared-memory

Unified MapReduce programming interface

- One single programming interface
- Transformation rules for Hadoop and Phoenix++ programming interfaces
  - Shared-memory and distributed state-of-the-art solutions

Unified MapReduce programming interface

- Focus on MapReduce logic
- Abstraction capable of keeping key performance components
- Able to be hereafter extended to comprehend new solutions and architectures (e.g., GPGPUs)

Unified MapReduce programming interface

```
@Type name(attr_name: attr_type, …)

@MapReduce<NAME, K_IN , V_IN , K_OUT , V_OUT , K_DIST > {

    @Map(key, value){
        // Map code logic
    }

    @SumReducer
}
```

Unified MapReduce programming interface

```
@MapReduce<NAME, K_IN , V_IN , K_OUT , V_OUT , K_DIST > {

    …

    @Reduce(key, values){
        double product = 1

        for(int i=0; i < length(values); i++)
            product *= values [ i ]

        emit(key, product)
    }
}
```

Transformation process

Stage	Elements
First	*imports/includes*
Second	*@MapReduce @Map @Reduce* *@Type* *global variables*
Third	*unsolved keywords*
Fourth	*variable types*
Fifth	*functions*

Unified interface - *Histogram*

```
@type pixel(r: ushort, g: ushort, b: ushort)

@MapReduce<HistogramMR, long, pixel, int, ulonglong, "*:768">

  @Map(key, p)
      emit(p.b, 1)
      emit(p.g+256, 1)
      emit(p.r+512, 1)

  @SumReducer
```

Hadoop interface - *Histogram*

```
public class HistogramMR {
   public static class Map extends Mapper<LongWritable, Pixel,
                                          IntWritable, LongWritable> {

      private final static LongWritable one = new LongWritable(1);

      @Override
      public void map(LongWritable key, Pixel p, Context context)
           throws IOException, InterruptedException {
         context.write(new IntWritable(p.getR()), one);
         context.write(new IntWritable(p.getG() + 256), one);
         context.write(new IntWritable(p.getB() + 512), one);
      }
   }
}
```

Phoenix++ interface - *Histogram*

```
class HistogramMR : public MapReduceSort<HistogramMR, pixel,
                                         intptr_t, uint64_t,
        array_container<intptr_t, uint64_t, sum_combiner, 768
#ifdef TBB
    , tbb::scalable_allocator
#endif
 > >
{
public:
    void map(data_type const& value, map_container& out) const {
        emit_intermediate(out, value.b, 1);
        emit_intermediate(out, value.g+256, 1);
        emit_intermediate(out, value.r+512, 1);
    }
};
```

Unified interface - *WordCount*

```
@MapReduce<WordCountMR, long, text, string, int>

  @Map(key, value)
    toupper(value)
    tokenize(value)
      emit(token, 1)

  @SumReducer
```

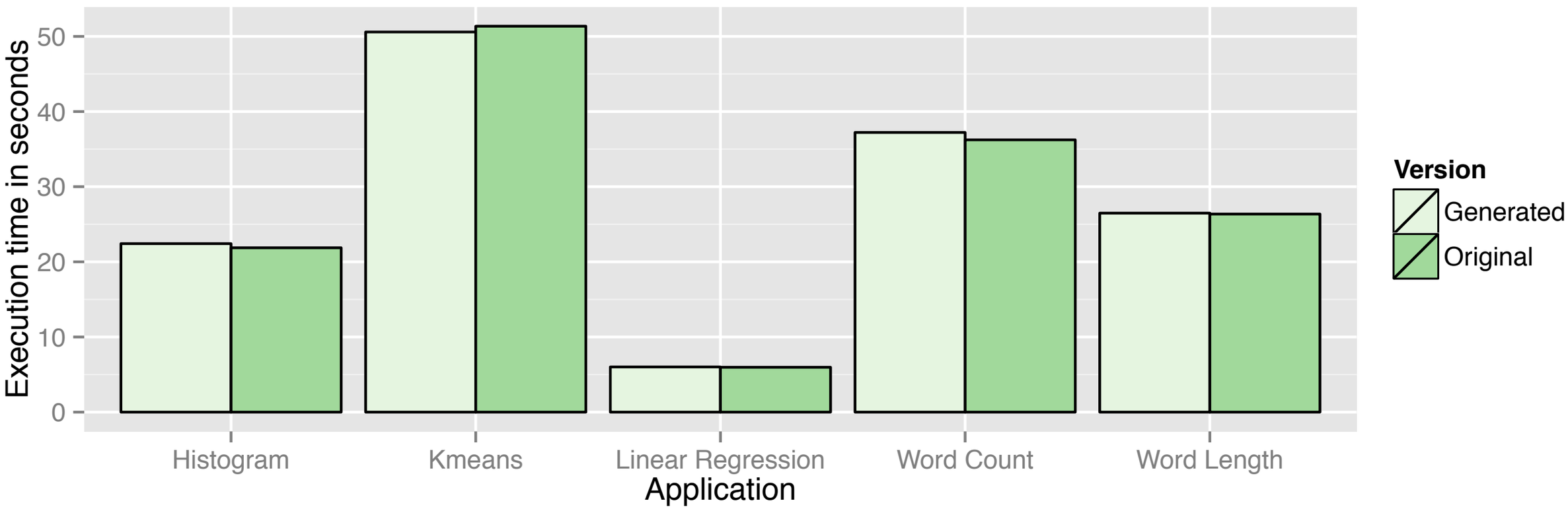
Hadoop interface - *WordCount*

```
public class WordCountMR {

    public static class Map extends Mapper<LongWritable, Text,
                                           Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        @Override
        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        } …
```

Phoenix++ interface - *WordCount*

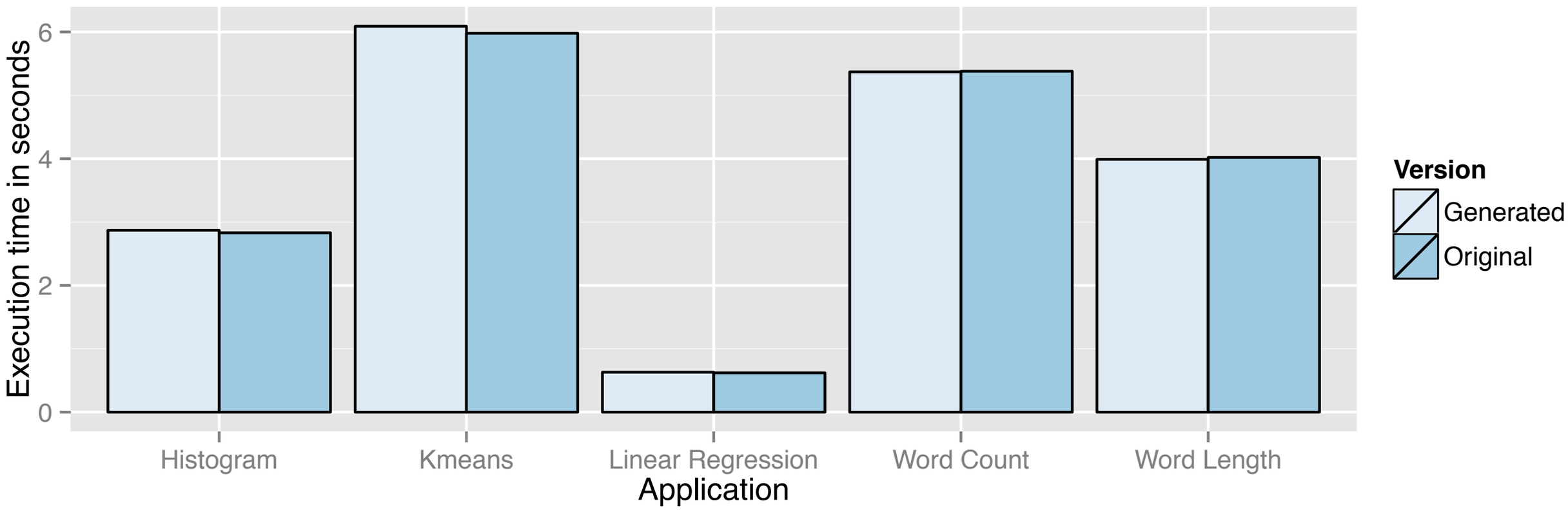
- **C++ includes** **± 6 lines**
- **MapReduce blocks** **± 25 lines**
- **Custom split** **± 24 lines**
- **Custom types** - C++ struct **± 34 lines**
- **TOTAL** **89 lines**

Performance evaluation - Hadoop



Mean execution time in seconds for original and generated Hadoop code (30 executions)

Performance evaluation - Phoenix++



Mean execution time in seconds for original and generated Phoenix++ code (30 executions)

SLOCCount

- Source Lines of Code counting
- Effort estimate based on COCOMO model

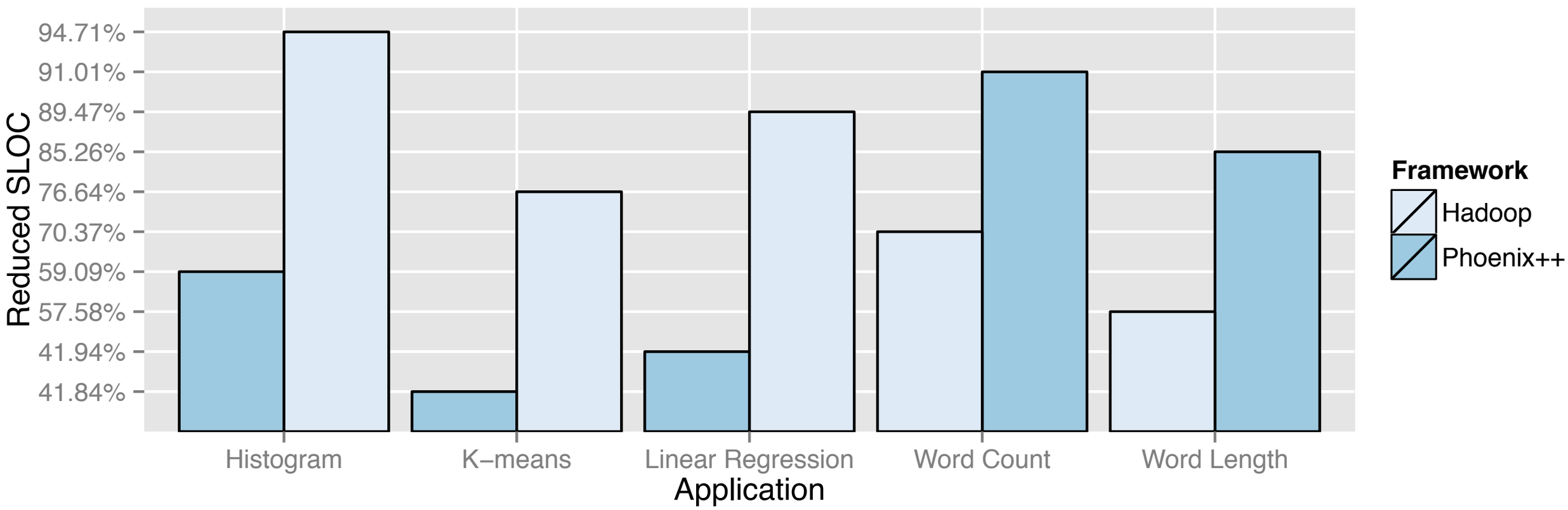
SLOC count and reduction

Application	Phoenix++	Hadoop	Unified Interface	Reduction compared to Phoenix++	Reduction compared to Hadoop
WordCount	*89*	*27*	*8*	*91.01%*	*70.37%*
WordLength	*95*	*33*	*14*	*85.26%*	*57.58%*
Histogram	*22*	*170*	*9*	*59.09%*	*94.71%*
K-means	*98*	*244*	*57*	*41.84%*	*76.64%*
Linear Regression	*31*	*171*	*18*	*41.94%*	*89.47%*

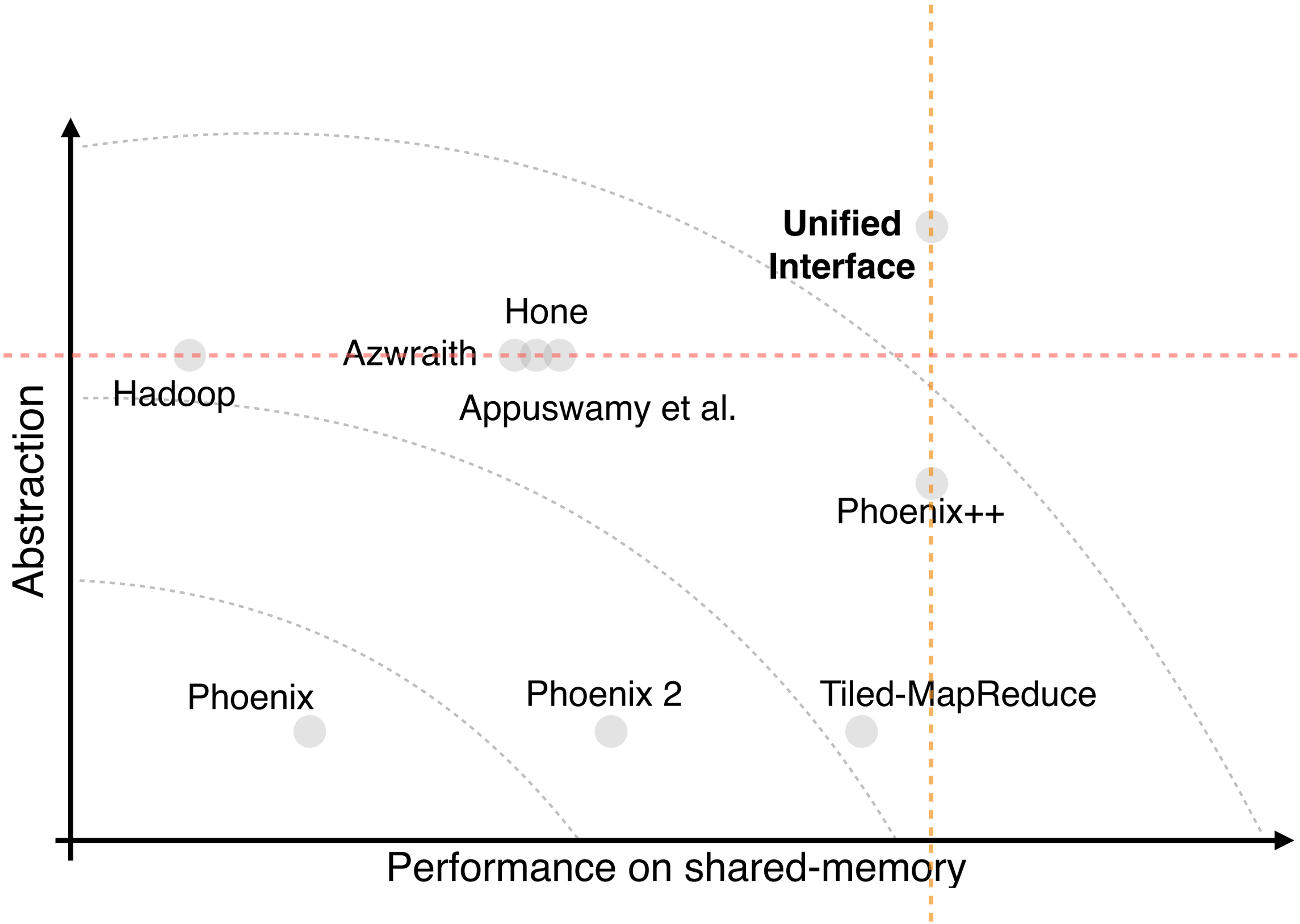
SLOC count and reduction

Application	Phoenix++	Hadoop	Unified Interface	Reduction compared to Phoenix++	Reduction compared to Hadoop
WordCount	**89**	27	8	**91.01%**	70.37%
WordLength	**95**	33	14	**85.26%**	57.58%
Histogram	22	**170**	9	59.09%	**94.71%**
K-means	98	**244**	57	41.84%	**76.64%**
Linear Regression	31	**171**	18	41.94%	**89.47%**

SLOC reduction



SLOC reduction for the interface version with curly braces


Unified Interface
Hone
Azwraith
Hadoop
Appuswamy et al.
Phoenix++
Phoenix
Phoenix 2
Tiled-MapReduce
Abstraction
Performance on shared-memory

Conclusions

- MapReduce implementations for lower level architectures, particularly, lose MapReduce's originally aimed abstraction
- Through a comprehensive set of transformation rules it is possible to effectively cover the components of Phoenix++ and Hadoop’s programming interfaces

Conclusions

- Performance evaluation shows less than 3% of variance from original and generated versions for all sample applications
- A SLOC and effort reduction from 41.84% and up to 96.48% is achieved

Conclusions

- Code written with the proposed unified interface can be reused for addressing different architectures
- Phoenix++ provides some optimizations for NUMA architectures, which are not supported by the transformation rules

Future work

- The effective construction of the compiler and code generator based on the proposed transformation rules
- The extension of transformation rules for compatibility with MapReduce solutions for different architectures (e.g., GPGPUs)

Pontifical Catholic University of Rio Grande do Sul (PUCRS),
Faculty of Informatics (FACIN), Computer Science Graduate Program (PPGCC)
Parallel Application Modeling Group (GMAP)

A Unified MapReduce Domain-Specific Language for Distributed and Shared Memory Architectures

Daniel Adornes, Dalvan Griebler, Cleverson Ledur, Luiz Gustavo Fernandes

SEKE 2015

Wyndham Pittsburgh University Center