Automatic Synthesis of Parallel and Distributed Unix Commands and Pipelines

Nikos Vasilakis   Jiasi Shen   Shivan Handa   Martin Rinard
MIT, CSAIL

Abstract

We present KumQuat, a system for automatically synthesizing parallel and distributed versions of Unix shell commands and pipelines. KumQuat follows a divide-and-conquer approach, decomposing commands into (i) a parallel mapper applying the original command to produce partial results, and (ii) an ordered merger that combines the partial results into the final output. KumQuat synthesizes the merger by applying repeated rounds of exploration; at each round, it compares the results of the synthesized program with those from the sequential program to discard invalid candidates. A series of refinements improve the performance of both the synthesis component and the resulting synthesized programs. Applied to a variety of POSIX, GNU, and third-party commands, KumQuat achieves up to an order-of magnitude performance improvements without any developer effort.

1 Introduction

Distributed and parallel systems can offer significant benefits over their sequential counterparts. For example, they can speed up time consuming computations or process large data sets that would not fit in any single computer. Despite these benefits, their development remains different from and significantly more difficult than standard sequential systems, often involving code that is difficult, laborious, and error-prone to develop [19,20,26,39,49,50]. Consider the following Unix pipeline for calculating term frequencies [4]:

```
cat * | tr A-Z a-z | tr -cs a-z '\n' |      (p1)
  sort | uniq -c | sort -rn | head 5 > out
```

With standard approaches, the distributed version can require significant manual effort to develop, even with support such as distributed programming languages [22,25,41,51,53] and distributed operating systems [24,29,32,33,52]. The effort can be considerable even for individual commands such as `wc` and `uniq` that fit well into popular distributed computing frameworks [10,30,31,56] or languages [2,6,28]. The effort is further compounded by the fact that commands often have dozens of flags, may be written in different programming languages, or may even be available only in binary form.

We present KumQuat, a new system for automatically synthesizing parallel and distributed versions of shell commands and pipelines. KumQuat uses *active learning* [8]: it repeatedly feeds the computation selected inputs, observes the resulting outputs, then repeats the process to infer the behavior of the computation and eventually gain the ability to automatically regenerate a parallel and/or distributed version.

KumQuat works with commands and pipelines that can be expressed as divide-and-conquer computations with two phases:[1] the first executes the original, unmodified command or pipeline on parts of the input; the second merges the partial results from the first phase to obtain the final output. To automate the regeneration of parallel and/or distributed versions, KumQuat automatically synthesizes the merge operators required to implement the second (merge) phase.

The resulting automatically regenerated parallel computation executes directly in the same environment and with the same program and data locations as the original sequential pipeline. This version is suitable, for example, for increasing the performance of local computations that do not require the heavyweight resources of a distributed computing infrastructure. The current KumQuat implementation also leverages Hadoop's streaming API (as well as custom data and program transformations) to produce distributed versions that contain synthesized Hadoop-ready code.

This paper makes the following contributions:

- **Algorithm:** It presents a new algorithm that uses active learning to automatically synthesize merge operators for parallel and distributed versions of Unix commands and pipelines. The resulting synthesized merge operators enable the automatic generation of parallel and distributed versions of standard sequential Unix commands and pipelines.
- **Domain-Specific Language:** It presents a domain-specific language for merge operators. This language supports both the class of merge operators relevant to this domain and the

[1]There is no requirement that the actual internal implementation of the commands or pipelines must be structured as a divide-and-conquer computation—the requirement is instead only that the computation that it implements can be expressed in this way.

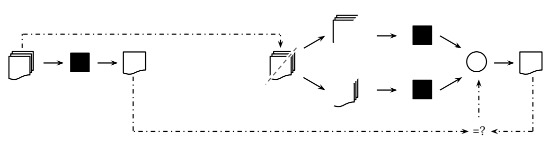


Fig. 1: High-level schematic. KUMQUAT infers and generates a distributed version ■²○ of a sequential program ■. The original sequential program ■ is treated as a black box (left). The generated distributed version ■²○ leverages ■ to implement most of the functionality, but synthesizes an adequate combiner ○ by observing the input-output behavior of both programs.

efficient synthesis algorithm for automatically generating these merge operators.

- **Transformations:** It presents program and data transformations that improve the efficiency of the synthesized parallel and distributed computations.
- **System:** It presents KUMQUAT, a system that uses the synthesized merge operators to automatically regenerate divide-and-conquer parallel and distributed versions of UNIX commands and pipelines.
- **Experimental Results:** It presents experimental results that characterize the effectiveness of KUMQUAT on a set of benchmark commands and pipelines. The results show that KUMQUAT can synthesize efficient combiners that improve the performance of popular commands by an order of magnitude in under a minute as well as the majority of stages in longer pipelines.

The paper is structured as follows. It starts with an example illustrating how KUMQUAT automatically generates a distributed version of the above UNIX pipeline p_1 (§2). It then presents the core KUMQUAT design (§3), and several performance and correctness refinements that enhance its effectiveness (§4). It then presents KUMQUAT's implementation (§5) and evaluation (§6), compares with previous work (§7), and concludes (§8).

2 Programmer Perspective

This section outlines KUMQUAT's key elements from the perspective of a user tasked with developing a distributed version of pipeline p_1. We first focus on the regeneration of only one of p_1's commands (§2), then on the regeneration of the full pipeline (§2).

A Single Command: The first processing stage, command `tr A-Z a-z`, transliterates upper- to lower-case characters:

```
kumquat tr A-Z a-z
```

KUMQUAT treats `tr A-Z a-z` as a black box. Its only interaction involves feeding the command inputs and observing its outputs. For now, we refer to `tr A-Z a-z` (and, more generally, the targets of active learning) as ■, to hide its contents and signal its position at different parts of the pipeline.

KUMQUAT starts by generating and feeding some random inputs to ■. Based on these inputs, it observes the types of ■'s outputs and attempts to extract a coarse, high-level specification σ for the component's interface—for example, that the result contains numerical elements.

Based on σ, KUMQUAT creates a lightweight scaffolding infrastructure such as the one presented in Fig. 1. The infrastructure is structured in a way that places two ■ next to each other, ready to receive input and execute in parallel. A final combiner function, marked as ○, collects their results and attempts to merge them in a meaningful way. The ○ is based on ■'s coarsely-inferred specification, with KUMQUAT focusing on collapsing the search space as quickly as possible.

With the starting sequential command ■ and the current parallel implementation ■²○, KUMQUAT infers the combiner function by choosing an input, feeding it to both versions ■ and ■²○ of the command, and comparing the outputs. It gives the sequential version the entire input and each ■ in the parallel version half the input.

KUMQUAT attempts several candidate operators for the combiner function—*e.g.*, list concatenation `(++)`, integer addition `(+)`—as well as combinations of these operators. Comparing the outputs from the sequential ■ and parallel ■²○, KUMQUAT refines its selection of both the combiner function as well as the subsequent inputs. If the results do not match, KUMQUAT attempts a different operator—a choice guided by several criteria, including the result's type signature. If the results match, KUMQUAT chooses inputs that are larger and that can potentially reveal any issues with the selected merge operator. When KUMQUAT has obtained enough information from these executions, it regenerates the parallel and distributed versions.

Parallel Version: The regenerated parallel version executes without any modifications to the surrounding environment nor any additional runtime support. It executes with the same shell, within the same environment, and with the same program and data locations as the original command. It is therefore suitable for increasing the performance of commands and pipelines that operate on inputs that fit within the user's computer and do not require the heavyweight resources of a large distributed computing infrastructure. Because the parallel version operates with no distributed processing overheads (such as startup overheads and overheads associated with moving inputs and outputs into and out of the distributed infrastructure), the KUMQUAT active learning/merge operation synthesis phase (which works with small inputs) works with the parallel version.

The regenerated parallel command is written in `tr.par.sh`, a script whose contents amount to:[2]

```
$R <(cat $IN1 | $M) <(cat $IN2 | $M)
```

This particular regeneration works with two processors. KUMQUAT launches two ■ each processing half of the input in parallel, merging the results by applying the synthesized merge function `R`. The variables `IN{1,2}` point to the two input halves, `M` points to `tr A-Z a-z`, and the synthesized merge command `R` implements `(++)`.

[2]We have simplified the script for clarity, but not significantly.

In practice, the regenerated program also depends on several parameters such as the number of available processors and the input file sizes. A larger number of available processors leads to a corresponding number of `<(...)` constructs and associated input files. If the original `tr A-Z a-z` command received its input from a file, then KUMQUAT splits the input and populates the `IN*` variables; otherwise, these variables point to named pipes (FIFOs) in the file-system. When executed, the new script reads the results of `tr A-Z a-z` from the two input streams and emits results to the output stream. Finally, POSIX-compliant, temporary named pipes (FIFOs) (§5) replace process substitution, expressed as `<(...)`.

Distributed Version: KUMQUAT also generates a distributed, fault-tolerant version—the current implementation synthesizes Hadoop-ready code. It leverages Hadoop's streaming API as well as custom data and program transformations. The streaming API allows writing `mapper` and `reducer` functions in languages other than Java; KUMQUAT's transformations retrofit order on Hadoop's unordered reducer invocation by introducing additional metadata and value wrapping-unwrapping pairs. The generated code looks like this:

```
hadoop jar $HLib/hadoop-streaming-*.jar
  -input $IN -output $OUT
  -mapper $M -combiner $C -reducer $R
```

Variables point to the same values as in the parallel case, except `C` which points to a custom Hadoop combiner (§5). Hadoop takes care of partitioning and replicating the input files and intermediary results, as well as the communication between different parts of the program.

The map and the reduce functions are identical for both versions. Where they differ is the scaffolding code not shown here: the distributed program is significantly simpler than the parallel version. This is because input partitioning, scheduling, and communication have to be explicitly provided by KUMQUAT in the parallel version, whereas these concerns are taken care of by Hadoop in the distributed version.

The Rest of the Pipeline: The pipeline p_1 features several other components than just `tr A-Z a-z`. The goal is to synthesize a distributed replacement for the entire pipeline. Rather than invoking KUMQUAT for every component in the pipeline, the developer simply passes the script containing the complete pipeline p_1 to KUMQUAT:

```
kq -t 200 -p p1.sh
```

KUMQUAT statically parses, analyzes, and rewrites p_1 to inject active-learning monitors at the boundaries of each command, prefixing each command with a `kq-io` monitoring command that interposes on each command's input-output pairs. The interposition monitors then regenerate a distributed version of each stage according to the aforementioned process. See Tab. 1 (Section (§6)) for the synthesized combiners in p_1 along with their learning and regeneration times.

3 Core System Design

We next present the core system design of KUMQUAT.

3.1 Foundations: Commands and Streams

In UNIX, shell commands are abstract *processes* that operate on data *streams*. Streams, in combination with optional arguments and carefully designed default values informed by practical use, are partly responsible for much of the ability of shell commands and pipelines to express powerful computations with little code. Many expressions are simplified because commands and pipelines are designed to operate on streams while maintaining minimal, if any, state on the side. Commands often operate on a single element (line) or on pairs of adjacent lines.

KUMQUAT views streams as ordered and finite, and processes as deterministic and monotonic. These assumptions simplify KUMQUAT's underlying theoretical model. A stream can be represented as a finite sequence of elements—an inductive datatype like the higher-order datatype `List a` (`[a]`).

The UNIX stream abstraction (and, more generally, the language of the shell) is weakly uni-typed, representing all data as lines of text—including numbers, booleans, and other primitive types. Thus, at the core, KUMQUAT models the base type `a` of each element as `String`, the newline character as the element separator, and the EOF condition as the base `nil` element. While the UNIX shell does not explicitly specify many other types, individual elements or results can be encoded using a variety of data-types. For example, `wc`'s result `729 5435 42860` counting lines, words, and bytes can be viewed as a 3-tuple of the form (`Int,Int,Int`).

Processes can be thought of as pure functions operating on such lists. For example, the `tr` command, transliterating characters, can be viewed as a function with type `[String]` $\rightarrow$ `[String]`, whereas `wc -l` can be viewed as a function with type `[String]` $\rightarrow$ `Int`. Some commands are higher-order functions: `xargs -n`ν is a command that takes a command and provides it as input ν elements from the input stream; `time` takes a command and wraps its results with timing metadata.

3.2 Decomposition: Command + Combiner

KUMQUAT's underlying idea is to decompose the distributed version of a program p into two parts. The first part is comprised of parallel applications of a mapping function m (the mapper) followed by the second part, the application of a reduction function g (the reducer). Applications of m (and, often, g) are commutative and associative, which means that their applications on partial input can execute in parallel (this does not mean, however, that m and g can be interleaved as the two cannot be assumed to be commutative or associative between each other). One key enabler behind KUMQUAT is to assume (then check) that the mapper m is p itself.

For clarity of exposition, the following discussion focuses on 2-ary reducers—*i.e.*, ones with two input arguments—but the analysis generalizes straightforwardly to reducers of any arity (§6). For 2-ary combiners, the goal is to identify g such that: the result of a command f over a stream $(\alpha{+}{+}\beta)$ that can be broken into two parts α and β is the same as independent piecewise applications of f over α and β whose results are combined by a function g. That is, the following holds:

$$f(\alpha{+}{+}\beta) = g(f(\alpha), f(\beta)) \quad (1)$$

For classes where this transformation is possible, the goal becomes to identify and synthesize the function g.

Commands that are part of a pipeline are pure functions over their input stream and do not feature context-dependent semantics. KUMQUAT's target data intensive UNIX commands and pipelines, operating as pure functions over streams carrying large amounts of data, have this property. KUMQUAT does not target commands that may mutate the pipeline context (which is naturally rare in Unix's task-parallel pipelines).

An important invariant is that the type of the outputs of f and g must be the same. If $\Gamma \vdash f : T_1 \to T_2$ then, by construction, $\Gamma \vdash g : T_2 \to T_2 \to T_2$; that is, g is a special type of reducer whose input and output types are the same. When operating on UNIX's weakly uni-typed streams of type `[String]`, this invariant aids structural decomposition. Examples of type information include that `wc` has spaces between its results and that its values are numerical. As seen next, this property is used by the definition of the language (§3.3) describing combiners: all constructs are designed to produce outputs that have the same formatting as inputs.

3.3 Broad Classes of Reductions

To capture the space of possible reductions, KUMQUAT defines and uses a domain-specific language (DSL) presented in Fig. 2. Figure 3 presents the big-step execution semantics for KUMQUAT's DSL for describing combiners. The DSL describes all possible reduction programs that can be constructed, broken down into a few broad classes. More complex classes often incorporate the combination of less complex ones.

A program in KUMQUAT's DSL is an expression, which is a binary operation with parameter variables x_1 and x_2. An environment σ maps variable names to their values. In our usage scenarios, variables `a` and `b` hold the outputs of executing command f on the first and the second input streams, respectively. In other words, given σ with information of the input streams α and β, σ(`a`) evaluates to $f(\alpha)$ and σ(`b`) evaluates to $f(\beta)$.

The transition function ⇒ maps an expression within our DSL to its output value. For example, the operator **fuse** accepts two strings v_1 and v_2, tokenizes them using delimiter d, and pairwise applies binary operator b on the tokens produced. KUMQUAT's DSL contains a significant number of operators, which allow it to capture a large class of functionality required to implement shell commands.

The broad classes of combiners are presented below, ordered by level of complexity:

Simple Merge This construction captures a broad class of reductions that apply a straightforward merge on their inputs.

$$
\begin{array}{rcl}
e \in \mathsf{Expr} & := & b\ x_1\ x_2 \\
b, b_1, b_2 \in \mathsf{BinaryOp} & := & \textbf{num}\ n \mid \textbf{ifeq}\ u\ b_1\ b_2 \\
& \mid & \textbf{fuse}\ d\ b \mid \textbf{fuse-2}\ d\ b_1\ b_2 \\
& \mid & \textbf{unwrap-front}\ d\ b \\
& \mid & \textbf{unwrap-back}\ d\ b \\
& \mid & \textbf{offset}\ d_1\ d_2 \mid \textbf{stitch}\ d\ b \\
& \mid & \textbf{first} \mid \textbf{rerun} \\
n \in \mathsf{NumOp} & := & + \mid - \mid \times \mid / \mid \% \\
u \in \mathsf{UnaryOp} & := & \textbf{id} \mid \textbf{split}\ d\ o \\
o \in \mathsf{ElemOp} & = & \textbf{pick-front} \mid \textbf{pick-back} \\
& \mid & \textbf{drop-front} \mid \textbf{drop-back} \\
x_1, x_2 \in \mathsf{Variable} & := & \mathtt{a} \mid \mathtt{b} \\
d, d_1, d_2 \in \mathsf{Delim} & := & \text{`}\backslash n\text{`} \mid \text{`}\backslash t\text{`} \mid \text{` `} \mid \text{`,`} \mid \text{`\_`} \mid \text{`:`}
\end{array}
$$

Fig. 2: KUMQUAT's synthesis DSL. The DSL captures the space of synthesizable combiners.

Instances of this class include list concatenation, arithmetic operators, binary operators *etc.* For example, when m is `tr -cs`, g is `(++)` as the results of applying `tr -cs` to different streams can be simply concatenated. Operators of this class in KUMQUAT's DSL include **id**, **num**, **concat**, **unwrap-front**, **unwrap-back**, and **split**.

Sequence Zip (Fusion) This construction captures a broad class of reductions where a simple merge is applied piecewise on elements that have been extracted from a complex intermediate result. To achieve extraction, KUMQUAT needs to first identify a *delimiter* upon which it splits the intermediate results. For example, the delimiter for `wc -lw` is a space character (applied multiple times). After the application of the simple-merge operator, the result are combined using the same delimiter used for extraction. KUMQUAT's DSL supports such reductions with operators **fuse** and **fuse-2**.

Stateful Merge This construction captures the class of reductions where a merge is applied after a stateful preprocessing step. Such stateful pre-processing step attempts to capture and carry some state—*e.g.*, a counter, a sum, or a max—across partial results. The stateful component is usually very simple, but often combined with an operator of type fusion: one part of each partial result is stateful (and is combined with stateful merge) and the rest is stateless (and is combined with simple merge). The most common instance of stateful-merge reductions involves maintaining counters—extraction and placement operators are of type fusion, with the added complication of maintaining the sum across chunks. A simple example is the `nl` command: for every non-empty element in its input stream, `nl` returns a tuple whose first element is the element's index from the start of the stream. KUMQUAT's DSL supports such reductions with the **offset** operator.

Condition Check This construction captures the class of reductions where a condition has to be ensured or applied at the boundaries of each partial result. Invariants that should

$$\frac{}{\sigma \vdash x \Longrightarrow \sigma(x)} \qquad \frac{\sigma \vdash x_1 \Longrightarrow v_1 \quad \sigma \vdash x_2 \Longrightarrow v_2 \quad b\ v_1\ v_2 \Longrightarrow_e v}{\sigma \vdash b\ x_1\ x_2 \Longrightarrow v} \qquad \frac{}{\mathbf{id}\ v \Longrightarrow_e v} \qquad \frac{v_l = \mathsf{splitBy}\ d\ v \quad o\ v_l \Longrightarrow_e v' \quad v'' = \mathsf{combineBy}\ d\ v'}{(\mathbf{split}\ d\ o)\ v \Longrightarrow_e v''}$$

$$\frac{}{(\mathbf{fuse}\ d\ b)\ []\ [] \Longrightarrow_e []} \qquad \frac{\begin{array}{c} vh_1, vt_1 = \mathsf{splitFirst}\ d\ v_1 \quad vh_2, vt_2 = \mathsf{splitFirst}\ d\ v_2 \\ b\ vh_1\ vh_2 \Longrightarrow_e v \quad (\mathbf{fuse}\ d\ b)\ vt_1\ vt_2 \Longrightarrow_e v' \end{array}}{(\mathbf{fuse}\ d\ b)\ v_1\ v_2 \Longrightarrow_e v +\!+\ d +\!+\ v'} \qquad \frac{\begin{array}{c} vh_1, vt_1 = \mathsf{splitFirst}\ d\ v_1 \quad vh_2, vt_2 = \mathsf{splitFirst}\ d\ v_2 \\ b_1\ vh_1\ vh_2 \Longrightarrow_e v \quad (\mathbf{fuse}\ d\ b_2)\ vt_1\ vt_2 \Longrightarrow_e v' \end{array}}{(\mathbf{fuse\text{-}2}\ d\ b_1\ b_2)\ v_1\ v_2 \Longrightarrow_e v +\!+\ d +\!+\ v'}$$

$$\frac{}{(\mathbf{num}\ n)\ v_1\ v_2 \Longrightarrow_e \mathsf{intToStr}(n(\mathsf{strToInt}(v_1)\ \ \mathsf{strToInt}(v_2)))} \qquad \frac{}{(\mathbf{concat})\ v_1\ v_2 \Longrightarrow_e v_1 +\!+\ v_2} \qquad \frac{}{(\mathbf{concat}\ d)\ v_1\ v_2 \Longrightarrow_e v_1 +\!+\ d +\!+\ v_2}$$

$$\frac{\begin{array}{c} v_n = \mathsf{strToInt}(\mathsf{splitBy}\ d_2\ (\mathsf{splitBy}\ d_1\ v_1)[-1])[0] \\ f = \lambda.v_x\ ([\mathsf{intToStr}((\mathsf{strToInt}\ v_x[0]) + n)] +\!+\ v_x[1:]) \\ g = \lambda.v_x\ (\mathsf{combineBy}\ d_2\ (f\ (\mathsf{splitBy}\ d_2\ v_x))) \\ v = \mathsf{combineBy}\ d_1\ (\mathsf{map}\ g\ (\mathsf{splitBy}\ d_1\ v_2)) \end{array}}{(\mathbf{offset}\ d_1\ d_2)\ v_1\ v_2 \Longrightarrow_e v} \qquad \frac{\begin{array}{c} v'_1 = \mathsf{splitBy}\ d\ v_1 \\ v'_2 = \mathsf{splitBy}\ d\ v_2 \\ b\ v'_1[-1]\ v'_2[0] \Longrightarrow_e v_m \\ v' = v'_1[:-1] +\!+\ [v_m] +\!+\ v'_2[1:] \\ v = \mathsf{combineBy}\ d\ v' \end{array}}{(\mathbf{stitch}\ d\ b)\ v_1\ v_2 \Longrightarrow_e\ v} \qquad \begin{array}{c} \frac{}{\mathbf{first}\ v_1\ v_2 \Longrightarrow_e v_1} \\ \\ \frac{}{\mathbf{rerun}\ v_1\ v_2 \Longrightarrow_e \sim (v_1 +\!+\ v_2)} \end{array}$$

$$\frac{\begin{array}{c} u\ v_1 \Longrightarrow_e v'_1 \quad u\ v_2 \Longrightarrow_e v'_2 \\ v'_1 = v'_2 \quad b_1\ v_1\ v_2 \Longrightarrow_e v \end{array}}{(\mathbf{ifeq}\ u\ b_1\ b_2)\ v_1\ v_2 \Longrightarrow_e v} \qquad \frac{\begin{array}{c} v'_1 = \mathsf{removeLeading}\ d\ v_1 \\ v'_2 = \mathsf{removeLeading}\ d\ v_2 \end{array}}{(\mathbf{unwrap\text{-}front}\ d\ b)\ v_1\ v_2 \Longrightarrow_e d +\!+\ (b\ v'_1\ v'_2)} \qquad \begin{array}{c} \frac{}{\mathbf{pick\text{-}front}\ []\ \Longrightarrow_e []} \\ \\ \frac{}{\mathbf{pick\text{-}front}\ v\ \Longrightarrow_e [v[0]]} \end{array}$$

$$\frac{\begin{array}{c} u\ v_1 \Longrightarrow_e v'_1 \quad u\ v_2 \Longrightarrow_e v'_2 \\ v'_1 \neq v'_2 \quad b_2\ v_1\ v_2 \Longrightarrow_e v \end{array}}{(\mathbf{ifeq}\ u\ b_1\ b_2)\ v_1\ v_2 \Longrightarrow_e v} \qquad \frac{\begin{array}{c} v'_1 = \mathsf{removeTrailing}\ d\ v_1 \\ v'_2 = \mathsf{removeTrailing}\ d\ v_2 \end{array}}{(\mathbf{unwrap\text{-}back}\ d\ b)\ v_1\ v_2 \Longrightarrow_e (b\ v'_1\ v'_2) +\!+\ d} \qquad \begin{array}{c} \frac{}{\mathbf{pick\text{-}back}\ [] \Longrightarrow_e\ []} \\ \\ \frac{}{\mathbf{pick\text{-}back}\ v\ \Longrightarrow_e [v[-1]]} \end{array}$$

$$\frac{}{\mathbf{drop\text{-}front}\ []\ \Longrightarrow_e []} \qquad \frac{}{\mathbf{drop\text{-}front}\ v\ \Longrightarrow_e v[1:]} \qquad \frac{}{\mathbf{drop\text{-}back}\ []\ \Longrightarrow_e []} \qquad \frac{}{\mathbf{drop\text{-}back}\ v\ \Longrightarrow_e\ v[:-1]}$$

Fig. 3: DSL Semantics. The semantics of KUMQUAT's synthesis DSL, describing all the synthesizable classes of combiners.

hold for certain windows of streams (rather than a single element) are likely to break by splitting and merging streams at arbitrary points. KUMQUAT can merge partial results only after it checks, confirms, or (re-)applies the condition at the boundaries of partial results. Most such invariants that hold beyond a single line focus on pairs of lines—*i.e.*, most UNIX commands operate on pairs of adjacent lines. Thus, it is usually enough to re-apply the original command locally on the meeting point of the two partial results. As an example, the `uniq` utility identifies pairs of identical consecutive lines. A combiner may extract the tail of the first partial and the head of the second, apply the original command on that input, and then merge the result. KUMQUAT supports such reductions with operators **stitch**, **ifeq**, and **rerun**.

Interleaved Merge This construction captures the class of reductions where the combiner has to interleave elements from multiple partial results. Similar to the previous class, condition checks, the reducer can (and does) take advantage of the original command to do the merging. It picks elements in a FIFO fashion, but repeatedly applies the original command to choose the next element. The typical command that requires interleaved merge is `sort`. KUMQUAT's DSL does not currently support such reductions. It is straightforward to extend KUMQUAT with operators that specifically handle the reductions for `sort`.

3.4 Combiner Synthesis

Given a command f, KUMQUAT synthesizes a combiner g such that equation 1,

$$f(\alpha{+\!+}\beta) = g(f(\alpha), f(\beta)),$$

holds for all of the observed input streams α and β. A key property of the synthesis algorithm is that it actively generates inputs α, β during synthesis to effectively eliminate incorrect candidate combiners.

Alg. 1 outlines KUMQUAT's combiner synthesis algorithm, procedure *synthesize*. It takes a command, f, and a user-specified maximum size of the abstract syntax tree (AST) of each candidate combiner, n. The algorithm starts by invoking procedure *generateInputs*, which generates a set of random input streams (α, β above), and storing them in I. The algorithm then invokes procedure *getCommandOutputs* to run the command f on the inputs I and obtain the command outputs O. Specifically, O contains the values for $f(\alpha)$, $f(\beta)$, and $f(\alpha{+\!+}\beta)$ for each pair of input streams α, β in I. The algorithm next invokes procedure *observeOutputType* to collect the type for the values in O. Specifically, the observed type t indicates whether the values in O contain numbers or any of the potential delimiters.

The algorithm then prepares the search space of candidate combiners. It first invokes procedure *allCombiners* to obtain

```
Data: Command $f$, max combiner size $n$
Result: Synthesized combiner $g$
$I \leftarrow generateInputs()$
$O \leftarrow getCommandOutputs(f, I)$
$t \leftarrow observeOutputType(O)$
$G \leftarrow allCombiners(n, t)$
$G' \leftarrow filterSatCombiners(G, O)$
$G'' \leftarrow generateInputsToFilterSatCombiners(f, G')$
if $G'' = \emptyset$ then
    return Nil
end
else
    return $getBest(G'')$
end
```

Algorithm 1: Procedure *synthesize*, which implements KumQuat's core synthesis algorithm. The procedure takes a command and synthesizes a combiner for the command, satisfying its (actively generated) input-output behavior.

all possible combiners whose AST size is not greater than n and whose type is t. These candidate combiners are stored in G. The algorithm then invokes *filterSatCombiners* to eliminate candidate combiners in G whose input-output behavior does not conform to I, O. Specifically, G' contains a combiner g in G if equation 1 holds for each tuple of values $f(\alpha)$, $f(\beta)$, and $f(\alpha{+}{+}\beta)$ in O.

The algorithm passes the search space G' to procedure *generateInputsToFilterSatCombiners*, which actively generates more input-output behavior examples for f and returns a set of satisfying candidate combiners G''. We present this procedure in the next section (§3.5). The algorithm finally ranks the candidate combiners in G'' by invoking *getBest*, which returns the best combiner based on likelihood and performance. When there are no satisfying combiners in the search space, the algorithm returns Nil.

3.5 Input Generation

Recall (§3.4) that KumQuat generates input streams α, β for the command f and executes it to obtain the input-output behavior examples (values $f(\alpha)$, $f(\beta)$, and $f(\alpha{+}{+}\beta)$) for the combiner. KumQuat executes f with carefully chosen input streams that enable f to produce a variety of output values that define the combiner behavior.

A key observation is that command arguments affect the resulting reduction, and as such they need to be incorporated in the input generation phase. Even with a simple command such as `cat -s` as an example, its argument `-s` squeezes empty lines. To trigger this behavior, the synthesizer needs to generate inputs that feature adjacent empty lines. Other inputs are more difficult to generate—strings that match certain regular expressions for `grep` and `sed` can be particularly challenging. KumQuat addresses this by taking such arguments into account when generating inputs. Note that, different from flags that can affect the control of a command, such data-oriented values can be changed without affecting the reducer.

```
Data: Command $f$, potential candidate combiners $G$
Result: Satisfying combiners $G'$
$s \leftarrow$ Initial input shape specification
$G'' \leftarrow G$
while $true$ do
    $G_m \leftarrow \emptyset$
    foreach potential mutation $m$ do
        $s' \leftarrow mutate(s, m)$
        $I \leftarrow generateInputs(s')$
        $O \leftarrow getCommandOutputs(f, I)$
        $G_m[m] \leftarrow filterSatCombiners(G'', O)$
    end
    $G' \leftarrow filterAlwaysSatCombiners(G_m)$
    if $G' \neq G''$ then
        $m' \leftarrow getMostEffectiveMutation(G'', G_m)$
        $s \leftarrow mutate(s, m')$
        $G'' \leftarrow G'$
    end
    else
        return $G'$
    end
end
```

Algorithm 2: Procedure *generateInputsToFilterSatCombiners*, which implements KumQuat's input generation algorithm. The procedure generates inputs while actively mutating the input specifications to effectively eliminate incorrect candidates combiners.

Recall that the command f is a black box. In other words, KumQuat does not know beforehand about what input streams are the most appropriate for inferring the behavior of f. KumQuat adopts an active learning algorithm to generate input streams for f.

Alg. 2 presents KumQuat's input generation algorithm, procedure *generateInputsToFilterSatCombiners*. It takes a command, f, and a set of candidate combiners, G. The algorithm maintains a specification of the shape of input streams, s. Specifically, s specifies the ranges for the numbers of lines, the numbers of spaces, the lengths of words, and the homogeneity of the contents for the input streams. Invoking procedure *generateInputs* with s produces a set of random inputs that each conforms to the ranges specified in s.

The structure of the input generation algorithm is inspired by gradient descent. The algorithm starts with an initial input shape specification s, then iteratively mutates s and uses it to filter candidate combiners. In each iteration, the algorithm tries mutating s in various directions. Example mutations include increasing the number of lines or decreasing the homogeneity of words. For each potential mutation m, the mutated input shape specification s' is used to generate input streams I, to run the command f, and to obtain command outputs O, resulting in a set of satisfying combiners $G_m[m]$. After trying all of these mutations, the algorithm checks if any candidates are eliminated in the current iteration. If so, the algorithm invokes *getMostEffectivMutation* to identify the mutations that are the most effective at eliminating candidates. The algorithm then applies the best mutations to the input shape specification s

and enters the next round of iteration. If no candidates are eliminated in the current iteration, the algorithm terminates and returns the set of remaining candidate combiners.

4 KumQuat Refinements

This section describes a series of refinements that aim at improving the efficiency and effectiveness of the synthesizer as well as that of the synthesized programs.

4.1 Synthesis Refinements

A few extensions on the core synthesis algorithm have the potential to improve KUMQUAT's performance significantly.

Type Guidance: Recall (§3.4) that the synthesis algorithm uses a set of initial executions of a command to collect the type for the command's outputs. KUMQUAT leverages likely type information to guide its choice of DSL terms. The current implementation focuses on two types: delimiters and values.

If the initial set of command outputs does not contain a specific delimiter, *e.g.*, `':'`, KUMQUAT notes this information in the type of the synthesized command to allow the synthesizer to discard candidate combiners that use this delimiter. Similarly, if the initial set of command outputs does no contain Arabic numbers, KUMQUAT notes this information to discard candidate combiners that involve numerical operations. These refinements allow KUMQUAT to prune the synthesis search space significantly.

A complication with delimiters is that that they may contain duplicates of varying length, depending on the command's padding strategy. Notably, some commands insert a fixed number of spaces between values, while others insert a flexible number of consecutive spaces as visual padding. KUMQUAT first attempts to synthesize a combiner with the type guidance described above. If this synthesis fails, KUMQUAT retries after rewriting each group of consecutive spaces in the command outputs into the `'\t'` delimiter.

Operator Weights: Different (classes of) combiners have a different likelihood of appearing. For example, many combiners use **concat**, followed closely by **concat** *d*, in turn followed by **num**.

KUMQUAT uses such likelihood information to guide synthesis. Specifically, it uses operator likelihood to rank the satisfying candidate combiners at the end of Alg. 1. The operator likelihood information is often sufficient for ranking the most succinct and correct candidate combiner as the first one.

An interesting special case is for commands that can serve as their own combiners. That is, the combiner g satisfies $f(\alpha{+}{+}\beta) = g(f(\alpha), f(\beta)) = f(f(\alpha){+}{+}f(\beta))$ for all input streams α, β. A trivial DSL combiner would use **rerun**. However, combiners that use **rerun** may require re-executing the original command f over many pairs of elements in α and β. This can result in significant overheads due to constant costs (*e.g.*, process fork *etc.*) as well as repeating work, foregoing many of the parallelism gains. KUMQUAT hence deprioritizes such candidate combiners due to poor performance. In our experiments with the synthesizer (§6), candidate combiners that use **rerun** often take two orders of magnitude longer to execute than combiners that do not.

Parallel Synthesis: KUMQUAT's synthesis features ample opportunities for parallelization. One opportunity occurs in candidate generation, in which different worker replicas can explore disjoint subsets of the candidate space. Another opportunity occurs in input generation and testing—*i.e.*, calling the same synthesized candidate on multiple inputs. Both are particularly useful, as KUMQUAT's synthesis domain can involve significant operating system overheads—-*i.e.*, spawning a new processes for every input-output text, which is in turn communicated through interprocess communication mechanisms.

As scaling out involves constant overheads for process spawning and interprocess communication, scaling out makes sense only after constant costs are negligible relative to synthesis. This is achieved by having KUMQUAT scale out after a few AST levels have been explored.

4.2 Distribution Refinements

We now turn to refinements related to distributed execution.

Mapping onto MapReduce: KUMQUAT's map-combiner decomposition is subtly different from the one in the distributed MapReduce paradigm [10]. By default, MapReduce does not maintain input order: inputs to map are key-value pairs and outputs are grouped by key before handed off to the reducer. To leverage a MapReduce implementation, KUMQUAT must first graft ordering atop MapReduce.

KUMQUAT therefore augments MapReduce operations, starting with a custom *wrap-unwrap* pair that operates at a few crucial points (Fig. 4). *Wrap* prepends (tags) stream elements with ordered identifiers—*e.g.*, adds line-numbers by piping through `nl`. As these identifiers need to respect global order, this operation needs to happen before the start of the MapReduce job—*e.g.*, when the stream is in transit to the distributed file system.

Identifiers can be placed in-band, as long as the KUMQUAT-provided map and reduce functions take care of wrapping and unwrapping so that the command and synthesized combiner operates on the raw elements. KUMQUAT's map thus wraps first applies *unwrap*, then calls the command, and finally applies *wrap*. On ingress, *unwrap* splits the stream into two parallel streams—*i.e.*, identifiers and raw data values—and on egress *wrap* merges the two streams into one.

These identifiers are not only ordered but also unique, to prohibit MapReduce from grouping elements together: each stream is processed by the reducer in due order. To complete the picture, KUMQUAT provides a custom partitioner that during the shuffling phase sorts (rather than hash) identifiers.

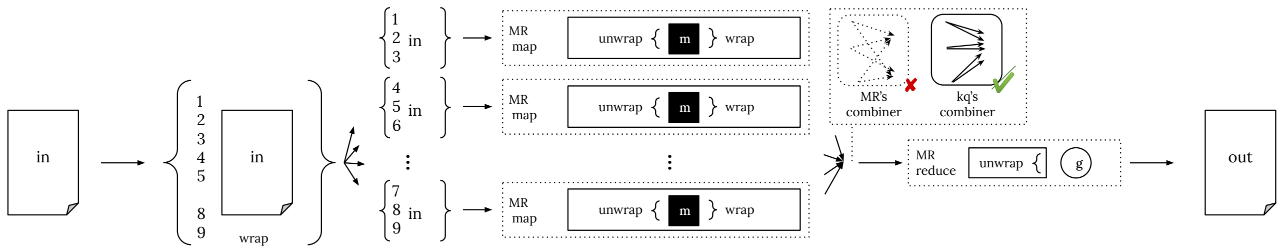
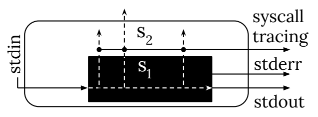


Fig. 4: KUMQUAT's distributed implementation over MapReduce's core abstractions. To be able to use the abstractions made available by a MapReduce implementation, KUMQUAT wraps stream elements with ordered identifiers. Wrappers around map and reduce take care of wrapping and unwrapping elements and a custom partitioner takes care of partitioning data by its identifiers.

Side Effects: Side-effectful commands are ones that modify some state outside their output streams (the main effect). Broadly, such state falls into two classes, both of which pose significant challenges for distributed systems.

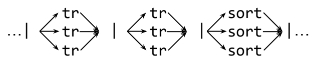
The first class comprises side-effects that fall outside the command's main memory—for example, writes to the file system, environment variables, or requests over the network. These side-effects are particularly challenging for the synthesizer to deal with, exactly because they are outside the monitored interface—even if two commands result into the same `stdout` and `stderr` streams for the same inputs, they might encode different computations.



To address this challenge, KUMQUAT wraps a command with system-call tracing infrastructure that reports all system calls during its execution. Applied online on every input, this wrapping could significantly decelerate the synthesizer. To avoid such slowdown, KUMQUAT wraps only a single invocation of a command, which completes concurrently (and always before) the synthesis: if a command has side-effects outside memory under normal operation, it will have them on any non-exceptional input.

The second class of side-effects are in-memory ones—for example, the fact that `sort` maintains a sorted list of inputs and that `wc` maintains a few counters. Fortunately, such side-effects are still encoded within `stdout` and `stderr` streams in KUMQUAT's domain: server or daemon commands would be challenging to deal with, but are not typically found as intermediate phases of data-processing pipelines.

Pipeline Rewrites: KUMQUAT's primary focus is individual commands. However, there are cases when scaling out a set of command invocations might still fail to accelerate a pipeline. This is due to KUMQUAT's fork-join parallelism: each pipeline stage will have to combine mapper results before proceeding to the next.



To address this challenge, KUMQUAT performs two transformations on the resulting parallel program. These transformations attempt to maintain parallel execution without any need for combination for as long as possible. Intuitively, they achieve this by pushing expensive combiners as far to the right of the pipeline as possible. More formally, the two transformations can be described over a pair p of pipeline stages $(m_1.g_1, m_2.g_2)$ as follows.

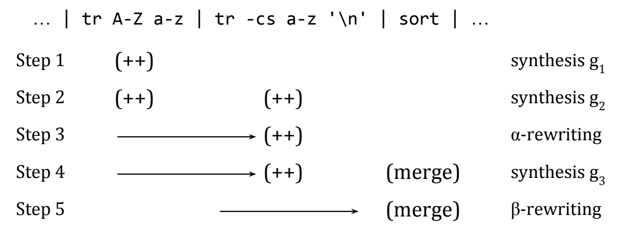


Fig. 5: KUMQUAT's combiner rewriting on part of p_1. KUMQUAT performs iterative rewriting to push combiners to the right and defer their application for as long as possible, effectively prolonging data parallelism.

If two stages have the same combiner function, then the pair of map (and combine) phases can be grouped together—*i.e.*, the following equation holds:

$$\forall m_1, g_1, m_2, g_2 : g_1 = g_2 \Rightarrow (m_1.g_1, m_2.g_2) = (m_1.m_2, g_1.g_2) \quad (\alpha\text{-rewriting})$$

If the combiner for the first stage is **concat** function, then it is only needed to apply the second combiner function (*i.e.*, concat performs as an identity combiner).

$$\forall m_1, g_1, m_2, g_2 : g_1 = (++) \Rightarrow (m_1.g_1, m_2.g_2) = (m_1.m_2, g_2) \quad (\beta\text{-rewriting})$$

Their effect on a small fragment from p_1 is shown in Fig. 5.

5 Implementation

This section provides KUMQUAT's implementation details.

Core System: KUMQUAT combines several components written in various languages. Both the active learning and regeneration components are written in Python, implementing the algorithms and DSLs presented in this paper. The base set of operators as well as the resulting reducers are all in Python. Synthesized combiners link against a 80-line utility library that provides runtime support—*e.g.*, for parsing stream descriptors provided as arguments (§2), reading streams with partial results, and applying the combiner.

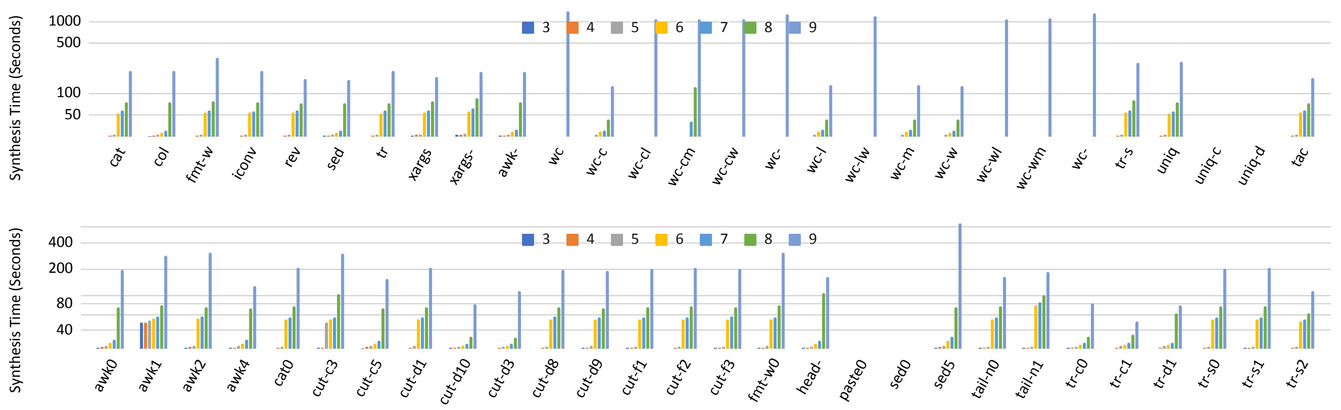


Fig. 6: Time to synthesize vs. the user-provided maximum AST size. The plots present the number of seconds used for the synthesizer to terminate and identify a combiner from the search space of all DSL expressions under the specified tree size. A value of zero indicates that no satisfying candidates exist within the search space. The plots show subsets of the commands collected from GitHub (top) and the Unix50 pipeline game (bottom) (*Cf.*§6.1).

For pipelines that incorporate multiple commands, KumQuat first calls into Smoosh's OCaml bindings built around LibDash [16]. Smoosh passes the script's AST as JSON to KumQuat, which performs light transformations that prefix each command with a KumQuat-provided `io` command. This command interposes at the boundaries of individual commands to extract input-output pairs. KumQuat then feeds the modified AST back to Smoosh, which converts it back to a POSIX-compliant string.

To check for side effects outside main memory, KumQuat prefixes commands with a call to `strace` [27], which reports calls to a pre-defined set of system calls.

Parallel and Distributed Drivers: For the parallel version, a driver orchestrates execution by splitting input into multiple chunks, creating the necessary FIFOs, passing partial inputs to maps and redirecting their output to the FIFOs, and passing FIFOs as arguments to the combiner.

For the distributed version, the driver reads from `stdin` and writes to `stdout`. The `wrap` function is a call to `nl` with custom delimiters. The `unwrap` function is a combination of `tee` with two `sed`s, each applied to one of `tee`'s resulting streams to demultiplex the identifier stream from the data stream. For convenience, HDFS's `put` and `get` are modified to call `wrap` and `unwrap` on the stream to/from HDFS.

6 Evaluation

This section uses a series of POSIX, GNU, and third-party commands to evaluate the performance of KumQuat as well as that of the synthesized parallel and distributed programs.

Highlights: For most benchmarks, KumQuat produces a synthesized combiner in under 5 minutes—with KumQuat checking candidate combiners at a rate of 1.3K expressions per second. Compared to KumQuat's baseline synthesis, refinements lead to significant improvements: $> 57\times$ from type refinements, $> 30\times$ from parallelism, and $> 11\times$ from refinements targeting input generation.

In the cases where KumQuat accelerates a command, the speedup is between $1.1 - 22\times$—and for a few commands such as `grep` and `xargs -n2` the speedup is near-linear. In some cases, *e.g.*, `head` and `sed 3q`, KumQuat naturally does not accelerate a command as its execution lasts for less than a second; in these cases, the synthesized parallel execution still remains within a second.

The distributed execution is much more varied, incurring a slowdown for many I/O-bound commands. We report on these results, and consider them acceptable, because (i) in some cases the goal is to obtain a parallel, not a distributed, version and (ii) in some other cases the goal of the distributed execution is to support data sizes that do not fit in a single computer, and not necessarily to accelerate the execution.

Setup: KumQuat and all parallel programs run on a server with 2TB of memory and 24 physical (48 virtual) $\times$ 3.40GHz Intel Xeon E7-8893, Ubuntu 19.10, GNU Coreutils 8.30-3.U2, Python 3.7.5, and OCaml 4.05.0. Distributed experiments run on 10$\times$i3.large AWS EC2 nodes, with Hadoop 3.2.1 running on Oracle's JDK 1.8.0_251; the rest of the software setup was identical to the server used for parallel experiments. Commands are evaluated on the Wikipedia corpus (150GB), with a 654K-word `american-insane` used as a dictionary for commands such as `comm`.

6.1 Individual Commands

Commands: We use two disjoint groups of command instances to evaluate KumQuat, where an *instance* is a command paired with a specific set of flags—*e.g.*, `grep -in` and `grep -v` are different instances of `grep`. We do not distinguish between string arguments (often implicitly) supplied as flag: `grep '1'` and `grep '2'` are considered the same instance. The first group contains 35 frequent commands from GitHub's most popular Unix pipelines. The second group contains 70 commands from Unix50's mini-game [23].

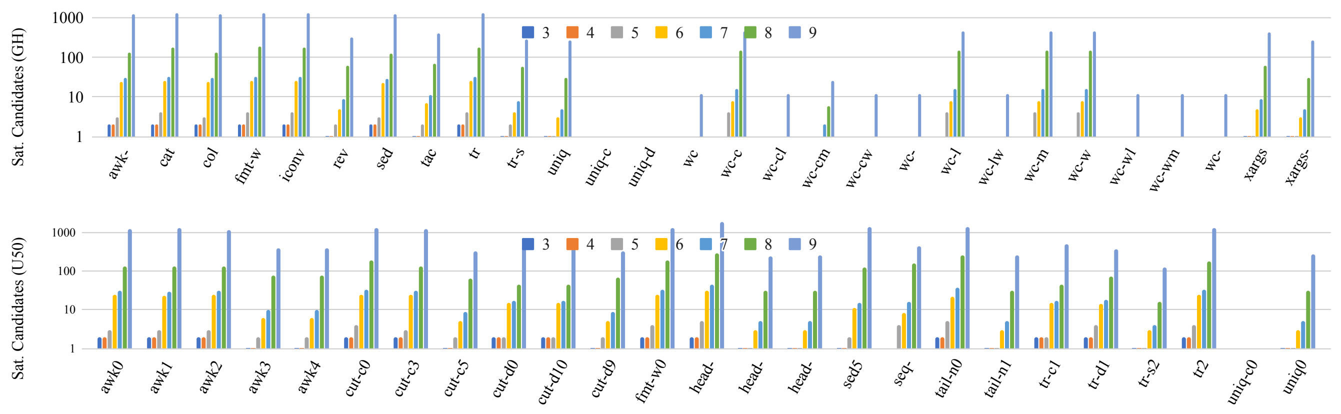


Fig. 7: Satisfying candidates vs. the user-provided maximum AST size. The plots present the number of plausible candidates that satisfy all generated I/O examples and are within the specified tree size. Top: commands collected from GitHub; bottom: commands from the Unix50 pipeline game (*Cf.*§6.1).

Synthesis: Fig. 6 and 7 show the synthesis time and the number of satisfying candidates as a function of the user-provided maximum AST size. (For clarity of exposition, we show a subset of the benchmarks.) Synthesis time indicates the number of seconds required for the synthesizer to terminate. Termination is when the synthesizer identified a combiner from the search space of all DSL expressions under the specified tree size. A value of zero indicates that no satisfying candidates were found within the search space. Satisfying candidates are all plausible candidates that satisfy all generated I/O examples and are within the specified tree size. This synthesis time is positively correlated with the size of the search space.

Combiners: Tab. 1 presents the synthesized combiners for most commands in our benchmark set. For all commands with combiners in the specified search space, KumQuat synthesized the correct combiner as the best candidate.

The vast majority of combiners need only an AST size of 3—which means they are synthesized within a few seconds. There are a few exceptions, for example `uniq`: given the freedom to explore combiners of maximum size 3, KumQuat synthesizes a combiner whose core is the term **rerun**. Given a freedom of 9 nodes, KumQuat synthesizes a longer combiner that features significantly better runtime performance.

Execution Time Fig. 8 shows the execution time of the sequential and synthesized parallel programs. For the parallel execution, the runtime is broken into several components: (i) setup, which includes the time to set up and tear down FIFO pipes and (worker) processes, (ii) splitting, which is the time it takes to split the input. (iii) parallel execution, which includes only the parallel applications of the command on the partial inputs, and (iv) combining, which is the time taken to execute the combiner on the outputs. Two long-running commands `grep` and `xargs -n2` (noted with $*$ and $**$ respectively) take smaller inputs: 1/10th for `grep` and 1/100th for `xargs`.

Speedups on the parallel component average 32.2×. For individual commands, adding the combiner lowers the average to 28×—but the result is highly dependent on both the sequential time of a command as well as the complexity of the combiner. For entire pipelines, the speedup depends on how far to the right KumQuat's transformations can push the combiner, discussed next.

The distributed experiments indicate lower speedups (not shown), averaging about 3.8× (and peaking at 4.2×). To better test whether some of the speedup limitations are due to Hadoop's streaming interface, we compared with native implementations expressed in Java for two benchmarks: `wc` and `grep`. KumQuat's runtime is about 47% slower; overheads are due to KumQuat's ordering transformations and Hadoop's streaming interface [11].

6.2 Pipelines

We next evaluate KumQuat on pipeline programs that combine several commands.

Calculating Misspellings: This is an 8-stage pipeline that calculates a set of misspelled words on the input document [3]. KumQuat synthesizes 6/8 stages in about 2.53 minutes. The remaining stages include `comm` and `sort`. The former features side-effects, as it takes as an additional argument a file stream, which KumQuat detects through tracing. This `comm` instance is interesting, because (i) it is trivially parallelizable and (ii) its combiner is expressible by KumQuat's DSL—but does not, due to `comm`'s perceived side-effects. The `sort` command is a more fundamental limitation: KumQuat's DSL does not support it, as it cannot express its combiner.

As both these stages are at the end of the pipeline, KumQuat is able to push the combiners up near the end of the pipeline (6/8 stages). This offers a 12.2× speedup for the synthesized fraction of the pipeline. Manually applying the divide-and-conquer transformation to `comm` and `sort`, in combination with the automatic parallelization of 6/8 stages, leads to an overall 6.8× speedup.

Calculating Bi-grams: This is an 8-stage pipeline that calculates a set of bigrams in the input stream. KumQuat synthesizes 4/8 stages in about 1.66 minutes. It also identifies that

Tab. 1: Synthesized combiners. Resulting combiners synthesized by KumQuat for the GitHub (top) and Unix50 (bottom) set of commands (*Cf.*§6.1).

Command	AST	Synthesized combiner
awk-trim, cat, col, fmt-w, iconv, rev, sed, tr	3	`(concat a b)`
tac	3	`(concat b a)`
xargs, xargs-n-2, tr-s	3	`(rerun a b)`
wc, wc-cl, wc-cw, wc-cwm, wc-lw, wc-wl, wc-wm, wc-wml	9	`((unwrap_back '\n' (unwrap_front '\t' (fuse '\t' num+))) a b)`
wc-c, wc-l, wc-m, wc-w	5	`((unwrap_back '\n' num+) a b)`
wc-cm	7	`((unwrap_front '\t' (offset '\t' '\n')) a b)`
uniq	3; 9	`(rerun a b); ((stitch '\n' (ifeq id first concat('\n'))) a b)`
uniq-c, uniq-d	N/A	N/A
awk0, awk3, cat0, cut-c0, cut-c1, fmt-w0, sed5, tr0 (. . . 34 more)	3	`(concat a b)`
paste0, sed0, sed1, sed2, sed3, sed4, uniq-c0	N/A	N/A
head-n0	3	`(first a b)`
tail-n0	3	`(first b a)`
head-n1, head-n2, tail-n1, tr-s2	3	`(rerun a b)`
uniq0	3; 9	`(rerun a b); ((stitch '\n' (ifeq id first concat('\n'))) a b)`
wc-l0	5	`((unwrap_back '\n' num+) a b)`

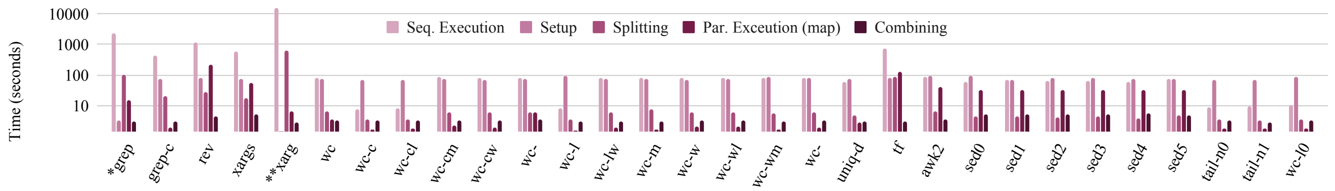


Fig. 8: Performance of Parallel Programs. The plot shows runtime performance of the parallel program broken down by phase—setting up pipes, splitting input, executing parallel mappers, and combining the results—against that of the sequential program; results are for 24× parallelism. Note log scale on *y* axis.

stages `tee` and `paste` have side-effects, requiring developer intervention. These are indeed challenging for KumQuat, as they feature multiple input/output streams. Another challenging stage is `sort`, discussed earlier. The last challenging stage is an `awk` that adds a new line at the beginning of the stream. While the combiner is, in fact, expressible by KumQuat's DSL, KumQuat cannot discover it within nine terms.

The resulting program features a 11.8× speedup for the synthesized fraction of the pipeline. Manually applying transformations to the challenging stages, combined with the 4/8 synthesized parallel stages, results in an overall 5.2× speedup.

Calculating Top N Terms: This is the 7-stage pipeline shown in the introduction, calculating a set of high-frequency terms from the input document [4]. KumQuat synthesizes 5/7 stages in about 2.08 minutes. The remaining stages include two applications of `sort` (described earlier). KumQuat is able to push the combiners up near the mid of the pipeline (6/8 stages); the resulting parallel program executes with a 9.6× speedup for the synthesized fraction of the pipeline. Applying the two `sort` stages manually, the resulting pipeline executes about 4.1× times faster.

6.3 Micro-benchmarks

Tracing: To understand the overheads of system call tracing, we apply it on successfully synthesized commands. For each command, the output written to `/dev/null` and the output of `strace` was written to `/dev/shm` (memory-mapped file-system). The average slowdown was about 1.88×; this overhead is insignificant relative to KumQuat's overall runtime performance: even for 100MB input, the highest observed was 10.10*s* for a command that without `strace` took 8.10*s*.

No-op Pipeline: To evaluate the benefits of these transformations, we perform a micro-benchmark in which we compare a 10-stage no-op pipeline with and without the transformations applied. All stages perform `cat`; while this program would not result in any practical speedups, it shows the potential for speedup gains from the transformations. The difference between the two cases is substantial: without the transformations, the pipeline results in a 8.3× slowdown. With transformations, its runtime sees only a 1.2× slowdown.

Refinements: To understand the benefits of various optimizations is challenging due to the high synthesis times. We use the synthesis of `cat`'s combiner with an AST size of 8 to measure the impact of different synthesis refinements. Without type refinements, the search space contains 335636 terms; with type refinements, these drop to 5810 (reduction: 57.7×). Parallel synthesis takes about 75.9*s*. Sequential synthesis takes jumps to 273*s* (3.64×). We note that `cat` is the simplest possible command to synthesize; thus it sees the lowest ratio of improvements from KumQuat's refinements.

7 Related Work

We discuss related work in Unix synthesis, synthesis of divide-and-conquer computations, synthesis of distributed systems, program synthesis driven by provided input/output examples, and active learning of computer programs.

UNIX Synthesis: Prior work on synthesis for UNIX shell commands [9] and pipelines [5] has been guided by examples or natural-language specifications. Instead of automatically generating parallel or distributed versions of an existing command or pipline, the goal was to synthesize the sequential command or pipeline itself given examples or a natural language description.

Divide and Conquer Decomposition: Prior work focused on decomposing programs or program fragments using divide-and-conquer techniques [12, 13, 37, 46]. The majority of this work focuses on parallelizing special constructs—*e.g.*, loops, matrices, and arrays—rather than stream-oriented primitives. In some cases [12, 13], the map phase is augmented to maintain additional metadata used by the reducer phase; this is antithetical to KUMQUAT's approach, which explicitly leaves the unmodified original command as the map phase.

Of particular relevance is the synthesis of MapReduce-style distributed programs [46]. This work focuses on synthesizing entire programs from input-output pairs rather inferring only the reducer function from an existing implementation—applied in scenarios where a programmer develops the full distributed computation by preparing examples. It also enables a much larger decomposition space that includes `flatMap`, `reduceByKey`, `filterBy`, and as a result requires a much more expressive underlying framework such as Spark [56]. KUMQUAT can, in principle, synthesize the distributed computation for any framework that follows the basic MapReduce paradigm, as long as the framework provides support for black-box streaming operations—for which Hadoop provides a simple interface. Additionally, KUMQUAT adds metadata to preserve order, which generally in non-streaming systems and specifically in [46] is not a requirement.

Synthesizing Distributed Systems: There is prior work on synthesizing features of distributed systems [7, 15, 21, 38], for example to automatically infer fences and repair programs in the context of distributed environments. KUMQUAT is different both in application and technique: it synthesizes data-parallel programs from black-box UNIX commands, as opposed to modifications or augmentations of existing (and often white-box) programs or program fragments.

Input-Output Synthesis: KUMQUAT overlaps (and builds upon) techniques explored in the program inference and synthesis communities [1, 14, 17, 18, 34], and particularly programming-by-example [35, 45, 54]. These works assume a different usage scenarios—for example, many approaches require the user to provide a set of examples. KUMQUAT performs program inference by interacting with and observing the behavior of a full application. Rather than rely on a fixed corpus of input-output examples, the application's execution is used to define the target specification. Incorporating active learning allows these systems to successfully and efficiently expand the input-output examples automatically as needed.

Active Learning: Active learning is a classical topic in machine learning [40]. In the context of program inference, it includes learning (and regenerating) programs that interact with relational databases [42] or key/value stores [36], oracle-guided synthesis for loop-free programs [18], and techniques for pruning the search space in synthesis targeting Datalog [44]. KUMQUAT, in contrast, works with UNIX commands and pipelines to synthesize merge operations over streams produced by divide-and-conquer computations for automatic parallelization and/or distribution of the computation.

Commands and Shells: There is a series of systems that aid developers in running commands or script fragments in a parallel or distributed fashion. These range from simple UNIX utilities such as GNU `parallel` [48], SLURM [55] and `rush` [43] to parallel and distributed shells such as `rc` [33] and `dgsh` [47]. These tools require developers to modify programs to make use of the tools' APIs, in contrast to KUMQUAT, which aims to provide an automated solution that works directly on the original sequential command or pipeline.

8 Conclusion

This paper presented KUMQUAT, a system for automatically synthesizing parallel and distributed versions of commands and pipelines. KUMQUAT focuses first on commands, decomposing them into two phases: (i) a parallel mapper that applies the original command to produce partial results, and (ii) an ordered merger that combines the partial results and is synthesized by KUMQUAT. To synthesize the combiner, KUMQUAT applies program inference on the command—with the key insight being that the parallel specification to be inferred is the already-available sequential command. KUMQUAT synthesizes the merger by applying repeated rounds of exploration; at each round, it compares the results of the synthesized program with those from the sequential program to discard invalid candidates. A series of refinements improve the performance of both the inference component and the resulting synthesized programs.

KUMQUAT's implementation supports two back-ends. A parallel back-end accelerates execution on a single host without any additional runtime support—and is exploited to accelerate KUMQUAT itself. A distributed back-end offers program- and data-transformation wrappers to bolt KUMQUAT's two phases on MapReduce, offloading several distributed-systems challenges to a production-grade distributed computing runtime. Applied to a variety of POSIX, GNU, and other commands as well as UNIX pipelines, it achieves order-of-magnitude performance improvements.

Acknowledgments

We are thankful to Claudia Zhu, Konstantinos Kallas, and Konstantinos Mamouras. This work was partly supported by DARPA contract no. HR00112020013 (ALRSCC).

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2013.

[2] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[3] Jon Bentley. Programming pearls: A spelling checker. *Commun. ACM*, 28(5):456–462, May 1985.

[4] Jon Bentley, Don Knuth, and Doug McIlroy. Programming pearls: A literate program. *Commun. ACM*, 29(6):471–483, June 1986.

[5] Sanjay Bhansali and Mehdi T Harandi. Synthesis of unix programs using derivational analogy. *Machine Learning*, 10(1):7–55, 1993.

[6] Martin Biely, Pamela Delgado, Zarko Milosevic, and Andre Schiper. Distal: A framework for implementing fault-tolerant distributed algorithms. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '13, pages 1–8, Washington, DC, USA, 2013. IEEE Computer Society.

[7] Borzoo Bonakdarpour and Sandeep S. Kulkarni. Sycraft: A tool for synthesizing distributed fault-tolerant programs. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory*, pages 167–171, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[8] José P. Cambronero, Thurston H. Y. Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin C. Rinard. Active learning for software engineering. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 62–78, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Anthony Cozzie, Murph Finnicum, and Samuel T King. Macho: Programming with man pages. In *13th Workshop on Hot Topics in Operating Systems*, Napa, CA, United States, May 2011. USENIX Association.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[11] Mengwei Ding, Long Zheng, Yanchao Lu, Li Li, Song Guo, and Minyi Guo. More convenient more overhead: The performance evaluation of hadoop streaming. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, page 307–313, New York, NY, USA, 2011. Association for Computing Machinery.

[12] Azadeh Farzan and Victor Nicolet. Synthesis of divide and conquer parallelism for loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 540–555, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Azadeh Farzan and Victor Nicolet. Modular divide-and-conquer parallelization of nested loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 610–624, New York, NY, USA, 2019. Association for Computing Machinery.

[14] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM, 2015.

[15] Bernd Finkbeiner and Paul Gölz. Synthesis in distributed environments. *arXiv preprint arXiv:1710.05368*, 2017.

[16] Michael Greenberg and Austin J Blatt. Executable formal semantics for the posix shell. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019.

[17] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM Sigplan Notices*, volume 46, pages 317–330. ACM, 2011.

[18] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224. ACM, 2010.

[19] Wolfgang John, Joacim Halén, Xuejun Cai, Chunyan Fu, Torgny Holmberg, Vladimir Katardjiev, Tomas Mecklin, Mina Sedaghat, Pontus Sköldström, Daniel Turull, Vinay Yadhav, and James Kempf. Making cloud easy: Design considerations and first components of a distributed operating system for cloud. In *Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'18, pages 12–12, Berkeley, CA, USA, 2018. USENIX Association.

[20] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 445–451, New York, NY, USA, 2017. ACM.

[21] Idit Keidar. Distributed computing column 46: Synthesizing distributed and concurrent programs. *SIGACT News*, 43(2):84, June 2012.

[22] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[23] Nokia Bell Labs. The unix game—solve puzzles using unix pipes, 2019. Accessed: 2020-03-05.

[24] Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, 1988.

[25] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, Xerox Palo Alto Research Center, 1997.

[26] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. The case for the holistic language runtime system. In *Proceedings of the 1st International Workshop on Rack-scale Computing*, 2014.

[27] R McGrath and W Akkerman. Linux strace, 2004.

[28] Christopher Meiklejohn and Peter Van Roy. Lasp: a language for distributed, eventually consistent computations with crdts. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 7. ACM, 2015.

[29] Sape J Mullender, Guido Van Rossum, AS Tanenbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[30] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[31] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[32] John K Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, 1988.

[33] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*, pages 1–9, 1990.

[34] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[35] Mohammad Raza and Sumit Gulwani. Disjunctive program synthesis: A robust approach to programming by example. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[36] Martin C. Rinard, Jiasi Shen, and Varun Mangalick. Active learning for inference and regeneration of computer programs that store and retrieve data. In Elisa Gonzalez Boix and Richard P. Gabriel, editors, *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2018, Boston, MA, USA, November 7-8, 2018.*

[37] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, page 72–83, New York, NY, USA, 1999. Association for Computing Machinery.

[38] Sven Schewe. Synthesis of distributed systems. 2008.

[39] Malte Schwarzkopf, Matthew P. Grosvenor, and Steven Hand. New wine in old skins: The case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pages 9:1–9:7, New York, NY, USA, 2013. ACM.

[40] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[41] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 15–26, New York, NY, USA, 2005. ACM.

[42] Jiasi Shen and Martin Rinard. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '19. ACM, 2019.

[43] Wei Shen. A cross-platform command-line tool for executing jobs in parallel. `https://github.com/shenwei356/rush`, 2019.

[44] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 515–527, New York, NY, USA, 2018. Association for Computing Machinery.

[45] Rishabh Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment*, 9(10):816–827, 2016.

[46] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 326–340, New York, NY, USA, 2016. Association for Computing Machinery.

[47] Diomidis Spinellis and Marios Fragkoulis. Extending unix pipelines to dags. *IEEE Transactions on Computers*, 66(9):1547–1561, 2017.

[48] Ole Tange. Gnu parallel—the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[49] Blesson Varghese and Rajkumar Buyya. Next generation cloud computing. *Future Gener. Comput. Syst.*, 79(P3):849–861, February 2018.

[50] Nikos Vasilakis, Ben Karel, and Jonathan M. Smith. From lone dwarfs to giant superclusters: Rethinking operating system abstractions for the cloud. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 15–15, Berkeley, CA, USA, 2015. USENIX Association.

[51] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[52] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. *SIGOPS Oper. Syst. Rev.*, 17(5):49–70, October 1983.

[53] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with scalaloci. *Proc. ACM Program. Lang.*, 2(OOPSLA):129:1–129:30, October 2018.

[54] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. Automated migration of hierarchical data to relational tables using programming-by-example. *Proceedings of the VLDB Endowment*, 11(5):580–593, 2018.

[55] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[56] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.