Performance and Cost Optimization for Distributed Cloud-Native Systems

Ashraf Mahgoub

▶ To cite this version:

Ashraf Mahgoub. Performance and Cost Optimization for Distributed Cloud-Native Systems. Distributed, Parallel, and Cluster Computing [cs.DC]. Purdue University, 2022. English. NNT : . tel-03792194

HAL Id: tel-03792194
https://hal.science/tel-03792194

Submitted on 14 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

PERFORMANCE AND COST OPTIMIZATION FOR DISTRIBUTED CLOUD-NATIVE SYSTEMS

by

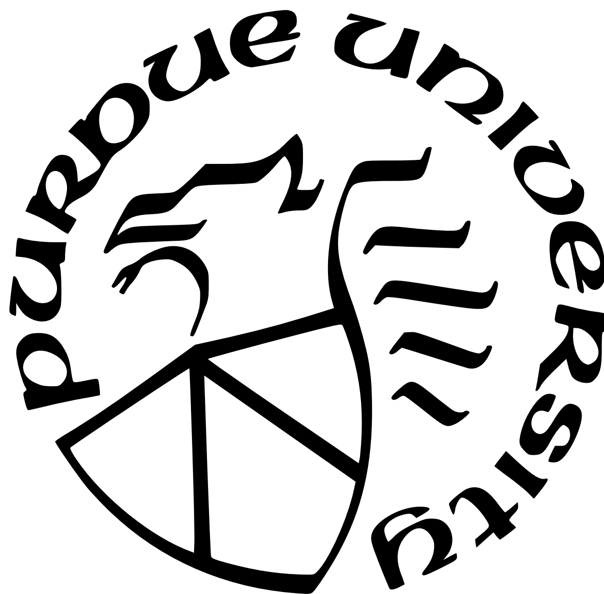
Ashraf Y Mahgoub

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



Department of Computer Science

West Lafayette, Indiana

August 2022

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Saurabh Bagchi, Co-Chair

Elmore Family School of Electrical and Computer Engineering

Dr. Ananth Grama, Co-Chair

Department of Computer Science

Dr. Ninghui Li

Department of Computer Science

Dr. Changhee Jung

Department of Computer Science

Dr. Somali Chaterji

Colleges of Engineering and Agriculture

Dr. Sonia Fahmy

Department of Computer Science

Approved by:

Dr. Sunil Prabhakar

To my mother and siblings for their continuous support;
to my lovely wife and sons for keeping me sane in difficult times;
to my advisors for giving me the opportunity to explore and learn.

TABLE OF CONTENTS

LIST OF TABLES 11

LIST OF FIGURES 12

ABSTRACT 20

1 INTRODUCTION AND PROPOSAL SUMMARY 24

1.1 Thesis Proposal Summary 24

2 RAFIKI: A MIDDLEWARE FOR PARAMETER TUNING OF NOSQL DATASTORES FOR DYNAMIC METAGENOMICS WORKLOADS 27

2.1 Abstract 27

2.2 Introduction 28

2.3 Background 32

2.3.1 NoSQL datastores 33

2.3.2 Cassandra Overview: Key Features 34

Write Workflow 34

Compaction 35

2.3.3 Design Distinctions in ScyllaDB 36

2.3.4 Performance metrics: Throughput and Latency 36

2.3.5 Genomics Workloads 37

Workload Dynamism 37

Processing Pipeline 37

2.4 Methodology 38

2.4.1 Rafiki Workflow 39

2.4.2 Notation 40

2.4.3 Workload Characterization 40

2.4.4 Important Parameter Identification 41

Key Parameters 41

2.4.5 Data Collection for Training Models 43

2.4.6 Performance Prediction Model . . . 44
Surrogate Performance Model . . . 44
Creation of the Performance Model . . . 44
2.4.7 Configuration Optimization . . . 46
Optimal Configuration Search . . . 46
Optimization using Genetic Algorithms . . . 47
2.4.8 DBA level of intervention . . . 48
2.5 Experiments and Results . . . 48
2.5.1 Workload Driver and Hardware . . . 49
2.5.2 Data Collection . . . 49
2.5.3 Surrogate Model and Training . . . 50
2.5.4 Cassandra's Sensitivity to Dynamic Workloads . . . 50
2.5.5 Key Parameters Selection . . . 51
2.5.6 Effectiveness of Selected Configuration Parameters . . . 53
2.5.7 Performance Prediction . . . 54
Training the Prediction Model . . . 55
Validating Model Performance . . . 55
2.5.8 Performance Improvement through Optimal Configuration . . . 57
2.5.9 Performance Improvement for Multiple Cassandra Instances . . . 58
2.5.10 ScyllaDB performance tuning . . . 58
2.6 Conclusion . . . 59
3 SOPHIA: ONLINE RECONFIGURATION OF CLUSTERED NOSQL DATABASES FOR TIME-VARYING WORKLOADS . . . 61
3.1 Abstract . . . 61
3.2 Introduction . . . 61
3.3 Design of SOPHIA . . . 66
3.3.1 Workload Modeling and Forecasting: . . . 66
3.3.2 Adapting a Static Configuration Tuner for SOPHIA: . . . 67
3.3.3 Dynamic Configuration Optimization: . . . 69

3.3.4 Finding Optimal Reconfiguration Plan with Genetic Algorithms: . . 71
3.3.5 Distributed Protocol for Online Reconfiguration: 72
3.3.6 DBA level of Intervention: 74
3.4 Datasets 74
3.5 Experimental Results 76

4 OPTIMUSCLOUD: HETEROGENEOUS CONFIGURATION OPTIMIZATION FOR DISTRIBUTED DATABASES IN THE CLOUD 90
4.1 Abstract 90
4.2 Introduction 90
4.3 Background and Rationale 97
4.3.1 Cassandra 97
4.3.2 Redis 97
4.3.3 Example Rationale for Heterogeneous Configurations 98
4.4 Design 99
4.4.1 Workload Representation and Prediction 99
4.4.2 Performance Prediction 100
4.4.3 Selection of Servers to Reconfigure 101
4.4.4 Selecting the Reconfiguration Plan 104
Objective Function Optimization 104
Cost-Benefit Analysis 105
4.4.5 Distinctions from Closest Prior Work 106
4.5 Experimental Setup and Results 107
4.5.1 Applications 108
4.5.2 Baselines 110
4.5.3 End-to-end System Evaluation 111
4.5.4 Sensitive Parameter Identification 114
4.5.5 Single Server Performance Prediction 114
4.5.6 Cluster Performance Prediction 116
4.5.7 Evaluation with Diverse Workloads 117

4.5.8 Evaluation with Redis . . . 119
4.5.9 Tolerance to Prediction Errors . . . 120

5 SONIC : APPLICATION-AWARE DATA PASSING FOR CHAINED SERVERLESS APPLICATIONS . . . 122
5.1 Abstract . . . 122
5.2 Introduction . . . 122
5.3 Rationale and Overview . . . 127
5.3.1 data passing Methods . . . 128
5.3.2 Dynamic Data Passing Method Selection . . . 129
5.4 Design . . . 131
5.4.1 Usage Model . . . 131
5.4.2 Online Profiling and Model Training . . . 132
5.4.3 Minimizing End-to-End Execution Time . . . 133
5.4.4 Online VM and Data Passing Selection . . . 135
5.4.5 Further Design Considerations . . . 136
5.5 Implementation . . . 137
5.6 Evaluation . . . 138
5.6.1 Performance Metrics . . . 138
5.6.2 Baselines and Methodology . . . 140
5.6.3 Applications . . . 142
5.6.4 End-to-End Evaluation . . . 142
5.6.5 Scalability . . . 144
Varying Degree of Concurrency . . . 145
Varying Intermediate Data Size . . . 146
5.6.6 Microbenchmarks . . . 147
Prediction Accuracy with Online Refinement . . . 147
Sonic's Performance Improvement Root Causes . . . 149
Sensitivity to Input Content . . . 150
Tolerance to Prediction Noise . . . 152

Varying Cold-Start Overheads 153
5.7 Related Work 154
5.8 Conclusion 155

6 ORION AND THE THREE RIGHTS: SIZING, BUNDLING, AND PREWARMING FOR SERVERLESS DAGS 157
6.1 Abstract 157
6.2 Introduction 157
6.3 Motivation 161
6.3.1 Workload Characterization 161
6.3.2 Performance Modeling 165
6.4 Design 166
6.4.1 Modeling E2E Latency Distribution 166
6.4.2 Allocating the Right Resources 169
6.4.3 Bundling Parallel Invocations 172
6.4.4 Pre-warming to Mitigate Cold Starts 174
6.4.5 Further Design Considerations 175
6.5 Implementation 176
6.6 Experimental Evaluation 177
6.6.1 Serverless DAG Applications 177
6.6.2 Orion and Competing Approaches 179
6.6.3 End-to-End Evaluation 181
Evaluation of Performance Model 183
Optimizing Resources for a Target E2E Latency 186
Impact of Varying Bundle Size 187
6.6.4 Generalizability to Microsoft Azure 188
6.7 Pre-warming Policy Simulator 189
6.8 Related Work 191
6.9 Discussion 193
6.10 Conclusion 195

7 WISEFUSE: WORKLOAD CHARACTERIZATION AND DAG TRANSFORMATION FOR SERVERLESS WORKFLOWS . . . 196
7.1 Introduction . . . 196
7.2 Workload Characterization . . . 200
7.2.1 General DAG Characterization . . . 201
7.2.2 Data Transfer Between Stages . . . 204
7.2.3 Skew among Parallel Workers . . . 206
7.2.4 Impact of DAG Skew and Intermediate Data Size on Latency . . . 206
7.3 Design . . . 208
7.3.1 Overview . . . 208
7.3.2 Workflow and Usage Model . . . 210
7.3.3 Per-function Performance Model . . . 211
7.3.4 Estimating the Impact of Fusion . . . 212
7.3.5 Estimating the Impact of Bundling . . . 213
7.3.6 Execution Plan Optimization . . . 215
7.3.7 Further Design Considerations . . . 219
7.4 Implementation . . . 220
7.5 Evaluation . . . 221
7.5.1 Baselines and Competing Approaches . . . 221
7.5.2 Applications . . . 222
7.5.3 E2E Evaluation . . . 224
7.5.4 Microbenchmarks . . . 229
Per-function Modeling Accuracy . . . 229
Estimating Latency After Fusion and Bundling . . . 231
Varying Input Data Size . . . 232
7.5.5 WiseFuse on Google Cloud . . . 233
7.6 Related Work . . . 235
7.7 Discussion . . . 237
7.8 Conclusion . . . 239

8 Conclusion 240

REFERENCES 242

LIST OF TABLES

2.1 Cassandra maximum, minimum, and default throughputs as the key-configuration parameters are varied 53

2.2 Prediction Model Performance for Cassandra 56

2.3 Cassandra: Performance improvement of optimal configuration selected by Rafiki vs. Default configuration performance for single-server and two-server setups. . 58

2.4 ScyllaDB: Performance of Rafiki selected configurations vs. Grid search 58

3.1 RMSE for predicting MG-RAST and Bus-Tracking workloads. 79

4.1 OptimusCloud's key features vs. existing systems. 94

4.2 (MC stands for *Markov Chain*). Workload prediction RMSE for MG-RAST and Bus-tracking workloads with different lookahead periods. 109

4.3 Comparison of different Single-server prediction techniques. OptimusCloud achieves better performance in terms of R^2 and $RMSE$ over all baselines. . . . 115

6.1 Correlation between execution times of functions in the Video Analytics DAG. In-series correlation is low but in-parallel correlation is high. 185

6.2 Video Analytics: Error rates for Orion's E2E latency estimation. Abbreviations: S→Split, E→ Extract, and C→Classify 185

6.3 Orion's E2E latency-optimized VM sizes. Orion meets the latency objective with a low error rate in the range of [-2.75%, 4.93%] 188

7.1 RMSE in latency estimates for each function in our evaluation applications. . . 230

LIST OF FIGURES

2.1 A high-level overview schematic of our proposed system Rafiki that searches through the configuration parameters' space of NoSQL datastores to achieve close-to-optimal performance. 27

2.2 Write workflow overview 34

2.3 Patterns of workload for MG-RAST. The workload read/write ratios are shown for 15 minutes intervals. 38

2.4 Performance of Cassandra with optimal configuration selected by Rafiki vs. Default configuration. Also three points are shown for the theoretically optimal performance using exhaustive searching. 51

2.5 ANOVA analysis for Cassandra to identify the key parameters, *i.e.*, the ones that most significantly control its performance. 52

2.6 Interdependency among selected parameters necessitating use of non-greedy search strategy. 54

2.7 Prediction error for Cassandra using the surrogate performance model with Neural Network, as a function of the number of training samples 54

2.8 Cassandra: Distribution of performance prediction errors for unseen configurations. The average absolute error is 7.5% with most projections lying in the | 5 |% range. 55

2.9 Cassandra: Distribution of performance prediction errors for unseen workloads. The average absolute error is 5.6% with most projections lying in the | 5 |% range 56

2.10 Average throughput for Cassandra and ScyllaDB under a 70% reads workload (collected every 10 seconds). Since Cassandra has a more stable performance compared to ScyllaDB, throughput prediction for Cassandra is more accurate. . 60

3.1 Workflow of Sophia with offline model building and the online operation, plus the new components of our system. It also shows the interactions with the NoSQL cluster and a static configuration tuner, which comes from prior work. 63

3.2 The effect of reconfiguration on the performance of the system. Sophia uses the workload duration information to estimate the cost and benefit of each reconfiguration step and generates plans that are globally beneficial. 63

3.3 Improvement for four different 30-minute test windows from MG-RAST real traces over the baseline solutions. 80

3.4 Gain of applying Sophia to the bus-tracking application. We use 8 Cassandra servers with RF=3, CL=1. A 100% on the Y-axis represents the theoretical best performance. Sophia achieves improvements of 24.5% over default, 21.5% over Static-Opt, and 28.5% over naïve. 81

3.5 Improvement for HPC data analytics workload with different levels of concurrency. We notice that Sophia achieves higher average throughput over all baselines 82

3.6 Improvement with scale using HPC workload with 5 jobs with RF=3 and CL=1. Sophia provides consistent gains across scale because the cost of reconfiguration does not change with scale (for the same RF and CL). The higher gains for 16 and 32 servers is due to the use of M5 instances, which can be exploited by Sophia better than Static-Opt. 82

3.7 Effect of varying RF and CL on system throughput. We use a cluster of 8 nodes and compare the performance of Sophia to Default, Static-Opt, and naïve. Sophia outperforms the static baselines and approaches the theoretical best as RF–CL increases. 85

3.8 Effect on increasing data volume per node. We use a cluster of 4 servers and compare the performance to the static optimized. The results show that Sophia's gain is consistent with increasing data volumes per node. 85

3.9 Effect of noise in workload prediction on the performance of Sophia on the data analytics workload with level of concurrency = 5. The percentage represents the amount of noise added to the predicted workload pattern. 85

3.10 Impact of tuning Redis' VM configuration parameters with Sophia with the data analytics workload. The percentage improvement of Sophia is shown on each bar and the right Y-axis is for the 2M jobs. A missing bar represents a failed job. We notice that the current Redis fails for large workloads (2M), while Sophia achieves the best of both worlds 88

4.1 Violin plot showing performance (throughput) of Best, Default, and Worst database configurations across different EC2 VM types. 92

4.2 Overview of OptimusCloud's workflow. First, a workload predictor is trained with historical traces from the database to be tuned. Second, a single server performance predictor is trained to map workload description, VM specs, and NoSQL application configuration to throughput. Third, a cluster-level performance predictor is used to estimate the throughput of the heterogeneous cluster of servers. In the online phase, our optimizer uses this predictor to evaluate the fitness of different VM/application configurations and provides the best performance within a given budget. 93

4.3 Importance of creating heterogeneous clusters (OptimusCloud Full) over homogeneous clusters (both Static and Dynamic) for Bus-Tracking application. Tuning both application and cloud configurations (OptimusCloud Full) has benefit over tuning only the VM configuration (3rd bar from left). The percentage value on the top of each bar denotes how much OptimusCloud improves over that particular scheme. 96

4.4 Change in Perf/$ for the write (solid) and read throughput (dotted) as we reconfigure the nodes from C4.large to R4.xlarge. 98

4.5 (RF=2, CL=1) Cluster performance depends not just on the configuration of each server, but also on the relative positions of the instances on the token ring. Cluster1 achieves 7× reads Ops/s over Cluster2 with the same VM types and sizes. 101

4.6 Replication examples with 3 different cluster sizes with RF=3. Cluster C1 has 3 nodes and each node has a complete copy of the data, therefore each node is a *Complete-Set*. Cluster C2 has 6 nodes and has the following *Complete-Set*s: [1,4] , [2,5] & [3,6]. Cluster C3 has 5 nodes (not divisible by RF=3), therefore it has two *Complete-Set*s: [1,3 (or 4)], [2,4 (or 5)]. 103

4.7 Importance of various parameters, including pairwise combinations. Parameters with black solid bars are w.r.t. the right Y-axis. EC2 configuration, the workload, and top 5 Cassandra parameters describe 81% of data variance, after which there is a significant drop in importance, denoted by the red dotted line. Top parameters are: file_cache_size (FCS), memtable_cleanup_threshold (MCT), memtable_heap_space (MHS), compaction_throughput (CT), and compaction_method (CM) 111

4.8 Evaluation of MG-RAST traces in Cassandra using OptimusCloud vs state-of-the-art tuning systems. The primary Y-axis represents the ratio of the normalized Ops/s/$ achieved by each system to the theoretical-best performance. OptimusCloud achieves the highest Perf/$ and lowest P99 latency. 112

4.9 Impact of noisy predictions on OptimusCloud's improvement over best Homogeneous-Static configurations 115

4.10 Performance prediction error histogram for heterogeneous clusters. We notice that OptimusCloud's error percentage is within -15% to +15% for 70% of the test points, with R^2 value of 0.91 and RMSE of 7.7 KOps/s. On the other hand, the best strawman shows poor performance (RMSE 36 KOps/s) while Selecta performs better (RMSE 21 KOps/s). 116

4.11 HPC workload evaluation with 10 concurrent jobs, and varying (RF,CL) requirements 116

4.12 HPC workload evaluation with 10 concurrent jobs, and varying (RF,CL) requirements 117

4.13 Bus-Tracking workload pattern with RF=3, CL=1 117

4.14 Evaluation on Redis for HPC workload with a cluster of 6 servers 118

4.15 Evaluation on Redis for HPC workload with a cluster of 12 servers 118

5.1 DAG overview (DAG definition provided by the user and our profiled DAG parameters) for Video Analytics application. 123

5.2 Execution time comparison with *Remote storage*, *VM storage*, and *Direct-Passing* for the LightGBM application with Fanout = 1, 3, 12. The best data passing method differs in every case. 125

5.3 Workflow of Sonic: Users provide a DAG definition and input data files for profiling. Sonic's online profiler executes the DAG and generates regression models that map the input size to the DAG's parameters. For every new input, the online manager uses the regression models to identify the best placement for every λ and best data passing method for every pair of sending\receiving λs in the DAG . . . 126

5.4 Sonic's interaction with an existing Resource Manager in the system. . . . 126

5.5 The three data passing options between two lambdas (λ_1 and λ_2). *VM-Storage* forces λ_2 to run on the same VM as λ_1 and avoids copying data. *Direct-Passing* stores the output file of λ_1 in the source VM's storage, and then copies the file directly to the receiving VM's storage. *Remote storage* uploads and downloads data through a remote storage (*e.g.* S3). . . . 127

5.6 Comparison between *Remote-Storage* and *Direct-Passing* for LightGBM workload with varying fanout degrees. Typically, beyond a certain fanout, Remote has lower execution time than Direct-Passing. A more well-provisioned VM (m5.xlarge) will shift the crossover point to the right. . . . 129

5.7 Performance of Sonic and the baselines for our three applications (memory-sized). Two performance metrics are shown: Perf/$ (bars; left axis) and end-to-end execution time (lines; right axis). Relative improvements in Perf/$ due to Sonic are at the top of each bar for the corresponding baseline. . . . 139

5.8 Performance of Sonic and the baselines for our three applications with the latency-optimized configuration. . . . 140

5.9 Scalability of Sonic, OpenLambda+S3, and SAND with equal portions of our three apps running concurrently. Sonic maintains its improvement over the entire scale, in both Perf/$/job and raw latency. . . . 145

5.10 Error in parameter estimation for Video Analytics application. Convergence point is reached with *e.g.* 35 jobs. Split_mem, Extract_mem, and Classify_mem represent memory footprints for Split, Extract and Classify functions respectively. All parameters are predicted using polynomial regression models, which take the application's input size in MBs as input. . . . 148

5.11 Example showing the benefits of Sonic over baselines. The table at the bottom shows possible λ placements for each stage in the LightGBM application and their corresponding latency and cost. We highlight the best sequence of decisions that achieves the best Perf/$ for the entire DAG. . . . 149

5.12 Normalized Perf/$ of Sonic vs SAND and OpenLambda+S3 with varying input sizes. We fix the Fanout-degree=6 and change the number of images used in training the Random-Forest model. . . . 150

5.13 Effect of training SONIC on YouTube video categories similar or dissimilar to test. % over the bars represent the SONIC's (second bar from left) gain over that baseline. 151

5.14 Impact of noise in SONIC's memory footprint predictions. Errors of less than ± 10% have small effect and over-prediction has higher effect than under-prediction. The values over the bars are w.r.t. SONIC with zero error (rightmost). 151

5.15 Effect of varying ratios of cold to hot execution times on SONIC, SAND, and OpenLambda+S3. Normalized Perf/$ is calculated by dividing the Perf/$ by the max across the three techniques. 153

6.1 ORION Overview. ORION profiles the DAG, estimates E2E latency CDF using *CONV* and *MAX*, and performs three system optimizations — right-sizing, bundling, and right pre-warming. 159

6.2 Illustration of DAG Depth (*i.e.* number of in-series stages) and Width (*i.e.* Maximum fanout degree). 159

6.3 ORION improves both latency and cost of Video Analytics DAG. Executions with min VM size (*i.e.* Min-Cost) and max VM size (*i.e.* Min-Latency) are for reference, and all latencies are for warm executions. 161

6.4 Characterization of depth (number of stages), width (degree of parallelism), and total number of nodes. Depth is low (P50 = 3; P95 = 8). 65% of DAGs are linear chains (no fanout) and width reaches 37 at the 95th percentile. 162

6.5 Latency distributions for the top-5 most frequent applications executed by *Azure Durable Functions* over a period of 1 week. We notice that the execution time varies significantly across different invocations of the same application. 163

6.6 Skew CDF for stages with different width ranges. 98.2% of the DAGs have a skew ≥ 2X, and skew increases for wider DAGs. 164

6.7 DAG invocation frequency (Blue solid) and its impact on the percentage of cold starts (Red dashed). DAGs with low invocation rate (*e.g.* once per day) always experience a cold start, whereas very frequent DAGs (*e.g.* ≥ 500 invocations per day) have very rare cold starts. 165

6.8 Video Analytics DAG. `Split` function downloads input video and splits it into chunks. Each chunk is passed to an instance of `Extract` function — extracting a representative frame, sent either to `Classify` function (Variant #1), or sent to *Pre-process* function and then `Classify` (Variant #2). The steps of estimating E2E latency distribution for Variant #1 are on the right. 167

6.9 Algorithm 1 170

6.10 **(Left) Separate VMs**: workers #2 & #4 finish early, while workers #1 & #3 take longer. **(Right) Bundling**: after workers #2 & #4 finish, workers #3 & #4 get more resources reducing stage latency. 173

6.11 Impact of different pre-warming decisions on the E2E latency and utilization for a chain of two in-series functions. Without pre-warming, the E2E latency increases due to added initialization time of function F2. Both early and late pre-warming are not desirable. 173

6.12 Skew is varied by changing the detection probability threshold as: 2%, 10%, and 15%, with lower values resulting in more detected objects and higher skews [192]. 179

6.13 Skew is varied by changing the maximum value for each hyper-parameter, *e.g.* we vary the max number of trees as: 50, 100, and 200, and these map to 2.6×, 4.4×, and 10× skews. 179

6.14 Skew is altered to 1×, 2×, and 4× by changing the number of training epochs as: 100, 500, and 200. 180

6.15 E2E evaluation with cold starts. Orion achieves the lowest latency and cost compared to all baselines. 180

6.16 Impact of pre-warming on latency and utilization. We use VM and bundle sizes selected by Orion and compare different execution strategies w.r.t. cold starts. Percentages over the bars of Orion show the improvements in P95 Latency (over Orion Cold-Start) and Utilization (over Orion Zero-Delay pre-warming). . . . 184

6.17 Video Analytics: Impact of varying bundle sizes. No-bunling has high latency due to computation skew. The optimal bundle size here is 6, and using a bundle size of ≥ 10 causes contention and the latency increases. 186

6.18 Orion's estimated latency CDF vs Actual CDF for Video Analytics application deployed in Azure Functions. Ignoring in-parallel correlation leads to higher errors for the Correlation-Agnostic baseline. 187

6.19 Orion's error with varying number of stages. More stages increase the error for the tail, while the median stays stable. 190

6.20 Simulation of an Oracle pre-warming policy where utilization improves with the width of distribution from which the pre-warming delays are chosen. Orion's strategy corresponds to the 0% variability, *i.e.* deterministic delay. 190

7.2 Example serverless DAG of Video Analytics application (7.1a) and the corresponding optimized execution plan generated by WiseFuse (7.2a). The gains in latency and cost over user-defined DAG, using either minimum or maximum VM size configurations (7.2b). 198

7.3 Overview of WiseFuse design. WiseFuse optimizes the execution plan for the user-defined DAG, which includes combinations of Fusion (*vertical coupling of in-series stages*), Bundling (*horizontal coupling of parallel workers in a stage*) actions, the size of each VM host a function or a group of DAG functions. . . . 201

7.4 Daily DAG invocations over 2 weeks. Weekend days are in yellow. 202

7.5 Daily invocations per DAG vs DAG rank (by frequency). Top 5% DAGs have 94.6% of invocations. 202

7.6 Distribution of DAG depth (*i.e.* stages), width (*i.e.* degree of parallelism) and total nodes (left for all DAGs, right for top 5%). 202

7.7 CDFs for P95, P50, min of execution time per DAG (left for all DAGs, right for top 5%). P50 of all DAGs is 5.6 sec, whereas it is 3.1 sec for the top 5%. . . . 202

7.8 For 50% of DAGs, the median invocation consumes more than 210 MB-sec. For top 5% most frequent DAGs, this number is lower, at 230 MB-sec. 204

7.9 CDF of total intermediate data files passed across the entire DAG. Higher intermediate data sizes are observed with the top 5% most frequent DAGs. 204

7.10 Data exchange latency on AWS Lambda, Google, and Azure comparing remote storage to local communication. Markers represent the median, and error bars represent Min to P95 latency range. 205

7.11 CDF of DAG max skew for all DAGs, top 5%, for DAGs with fanout ≥ 32 & ≥ 128. 207

7.12 CDFs of DAG latency per max skew (right) and intermediate data size (left). Longer E2E latencies are observed for DAGs with higher max skew and intermediate data size. 207

7.13 Without Bundling, latency is dominated by the straggler (X_2) and is equal to t_2. With Bundling, X_2 gets more resources after X_1 executes, decreasing the stage's latency to t'_2. 209

7.14 Coupling between Fusion and Bundling. Fusion reduces data exchange latency but prevents shuffling, which is essential to break the locality of stragglers and reduce the E2E latency. 209

7.15 Pseudo code for calculating the latency CDF for any sized bundle of functions in one stage 213

7.16 Speed-up in `Classify` when executed on a 6-core VM vs a 1-core VM. 215

7.17 WiseFuse's comparison in latency and $ cost. % over the bars show WiseFuse's gains in E2E latency (1.5× configuration) over this particular scheme. We set the latency target to (1.5×, 2.5×, and 5×) of the best theoretical latency (total processing time using an arbitrarily large VM and zero communication latency). 224

7.18 WiseFuse's comparison in latency and $ cost. % over the bars show WiseFuse's gains in E2E latency (1.5× configuration) over this particular scheme. We set the latency target to (1.5×, 2.5×, and 5×) of the best theoretical latency (total processing time using an arbitrarily large VM and zero communication latency). 225

7.19 Comparison between WiseFuse (-Full) and its two variants that do Bundling only, or Fusion only. 228

7.20 Comparison between WiseFuse and Grid Search. 230

7.21 Accuracy of WiseFuse's CDF estimation for a fused chain for our Video Analytics application. Without considering correlation, errors can be up to 12.5%. With correlation, errors come down to $\leq$ 3.4%. 232

7.22 Accuracy of WiseFuse's CDF estimation for a bundle of 4 `Classify` workers. WiseFuse achieves errors in the range of [-6.4%,7%] compared to the actual CDF. 232

7.23 Video Analytics: Comparison between WiseFuse's optimized plan and user-defined DAG using min and max VM sizes. We show how the latency and cost are impacted by varying input sizes. 234

7.24 Average latency for compute and communication on Google Cloud Functions with varying memory size. The network bandwidth continues to scale unlike in AWS Lambda. 234

7.25 Skew for compute and communication on Google Cloud Functions. While network bandwidth scales with memory size, the communication skew grows. . . . 234

ABSTRACT

We investigate the problem of performance and cost optimization for two types of cloud-native distributed systems: NoSQL data-stores and Serverless DAG applications.

First, NoSQL data-stores provide a set of features that is demanded by high performance computing (HPC) applications such as scalability, availability and schema flexibility. High performance computing (HPC) applications, such as metagenomics and other big data systems, need to store and analyze huge volumes of semi-structured data. Such applications often rely on NoSQL-based datastores, and optimizing these databases is a challenging endeavor, with over 50 configuration parameters in Cassandra alone. As the application executes, database workloads can change rapidly over time (e.g. from read-heavy to write-heavy), and a system tuned for one phase of the workload becomes suboptimal when the workload changes.

We present a method and a system for optimizing NoSQL configurations for Cassandra and ScyllaDB when running HPC and metagenomics workloads. First, we identify the significance of configuration parameters using ANOVA. Next, we apply neural networks prediction using the most significant parameters and their workload-dependent mapping to predict database throughput, as a surrogate model. Afterwards, we optimize the configuration using genetic algorithms on the surrogate to maximize the workload dependent performance. Using the proposed methodology in our first framework (RAFIKI), we can predict the throughput for unseen workloads and configuration values with an error of 7.5% for Cassandra and 6.9-7.8% for ScyllaDB. Searching the configuration spaces using the trained surrogate models, we achieve performance improvements of 41% for Cassandra and 9% for ScyllaDB over the default configuration with respect to a read-heavy workload, and also significant improvement for mixed workloads. In terms of searching speed, RAFIKI, using only 1/10,000-th of the searching time of exhaustive search, and reaches within 15% and 9.5% of the theoretically best achievable performances for Cassandra and ScyllaDB, respectively—supporting optimizations for highly dynamic workloads.

Next, we consider the problem of reconfiguring NoSQL databases under changing workload patterns. This is challenging because of the large configuration parameter search space

with complex interdependencies among the parameters. While state-of-the-art systems can automatically identify close-to-optimal configurations for static orkloads, they suffer for dynamic workloads as they overlook three fundamental challenges: (1) Estimating performance degradation during the reconfiguration process (such as due to database restart). (2) Predicting how transient the new workload pattern will be. (3) Respecting the application's availability requirements during reconfiguration. Our second framework, Sophia, addresses all these shortcomings using an optimization technique that combines workload prediction with a cost-benefit analyzer. Sophia computes the relative cost and benefit of each reconfiguration step, and determines an optimal reconfiguration for a future time window. This plan specifies when to change configurations and to what values, to achieve the best performance without degrading data availability. We demonstrate its effectiveness for three different workloads: a multi-tenant, global-scale metagenomics repository (MG-RAST), a bus-tracking application (Tiramisu), and an HPC data-analytics system, all with varying levels of workload complexity and demonstrating dynamic workload changes. We compare Sophia 's performance in throughput and tail-latency over various baselines for two popular NoSQL databases, Cassandra and Redis.

Afterwards, we focus on achieving cost and performance efficiency for cloud-hosted databases. This is challenging as it requires exploring a large configuration space, including the parameters exposed by the database along with the variety of VM configurations available in the cloud. Even small deviations from an optimal configuration have significant consequences on performance and cost. Existing systems that automate cloud deployment configuration can select near-optimal instance types for homogeneous clusters of virtual machines and for stateless, recurrent data analytics workloads. We show that to find optimal performance-per-$ cloud deployments for NoSQL database applications, it is important to (1) consider heterogeneous cluster configurations, (2) jointly optimize database and cloud VM configurations, and (3) dynamically adjust configuration as workload behavior changes. Therefore, we present our third framework, OptimusCloud, an online reconfiguration system that can efficiently perform such joint and heterogeneous configuration for dynamic workloads. We evaluate OptimusCloud with two clustered NoSQL systems: Cassandra and Redis, using three representative workloads and show that OptimusCloud provides

40% higher throughput/$ and 4.5× lower 99-percentile latency on average compared to state-of-the-art prior systems, CherryPick, Selecta, and our second framework Sophia.

Second, serverless platforms are becoming increasingly popular for their ease-of-use and pay-as-you-go billing. The structure of serverless applications is usually composed of multiple functions that are chained together to form a directed acyclic graph (DAG). The current approach of exchanging intermediate (ephemeral) data between functions is through a remote storage (such as S3), which introduces significant performance overhead. We compare three data-passing methods, which we call *VM-Storage*, *Direct-Passing*, and state-of-practice *Remote-Storage*. Crucially, we show that no single data-passing method prevails under all scenarios and the optimal choice depends on dynamic factors such as the size of input data, the size of intermediate data, the application's degree of parallelism, and network bandwidth. The first result of this line of work is Sonic, a data-passing manager that optimizes application performance and cost, by transparently selecting the optimal data-passing method for each edge of a serverless workflow DAG and implementing communication-aware function placement. Sonic monitors application parameters and uses simple regression models to adapt its hybrid data passing accordingly. We integrate Sonic with OpenLambda and evaluate the system on Amazon EC2 with three analytics applications, popular in the serverless environment. Sonic provides lower latency (raw performance) and higher performance/$ across diverse conditions, compared to four baselines: SAND, vanilla OpenLambda, OpenLambda with Pocket, and AWS Lambda.

Next, we target the problem of modeling the performance and cost of serverless DAGs, which enables the cloud provider to support service level performance objectives. To address this problem, we propose Orion, a performance modeling framework for serverless DAGs. We first analyze traces from a production FaaS infrastructure to identify three characteristics of serverless DAGs. We use these to motivate and design three features. The first is a performance model that accounts for runtime variabilities and dependencies among functions in a DAG. The second is a method for co-locating multiple parallel invocations within a single VM thus mitigating content-based skew among these invocations. The third is a method for pre-warming VMs for subsequent functions in a DAG with the right look-ahead time. We integrate these three innovations and evaluate Orion on AWS Lambda with three serverless

DAG applications. Our evaluation shows that compared to three competing approaches, ORION achieves up to 90% lower P95 latency without increasing $ cost, or up to 53% lower $ cost without increasing P95 latency.

Finally, we investigate the possibility of transforming serverless DAGs to cost and performance optimized variants with lower communication latency and execution skew. For this objective we propose WISEFUSE, an automated approach to generate an optimized execution plan for serverless DAGs for a user-specified latency objective (SLO) or cost budget. We introduce three optimizations: (1) **Fusion** combines in-series functions together in a single VM to reduce the communication overhead between cascaded functions. (2) **Bundling** executes a group of parallel invocations of a function in one VM to improve resource sharing among the parallel invocations to reduce skew. (3) **Resource Allocation** assigns the right size to each VM hosting a function or a group of functions to reduce the latency and cost of invoking the serverless DAG. We implement WISEFUSE and evaluate it experimentally using three serverless applications, namely, Video Analytics, Approximate SVD, and ML Analytics, which span different DAG structures, memory footprints, and intermediate data sizes. In comparison to competing approaches, WISEFUSE shows significant improvements in E2E latency and cost. Specifically, for the ML pipeline, WISEFUSE achieves a P95 latency that is 67% lower than Photons [SoCC-20], 39% lower than Faastlane [USENIX ATC-21], and 90% lower than [USENIX ATC-21], without increasing the $ cost.

1. INTRODUCTION AND PROPOSAL SUMMARY

In this chapter, we introduce the problem of automated configuration tuning for cloud-hosted distributed systems. We highlight the importance of the problem, the key challenges, and the proposed solutions. We provide an overview for three frameworks that target optimizing the performance and cost of NoSQL data-stores: Rafiki, Sophia, and OptimusCloud. We also provide an overview for three frameworks that target performance and cost optimization of Serverless DAGs: Sonic, Orion, and WiseFuse. First, we give a summary of our thesis proposal. Second, we give an overview of every framework's design and evaluation. Finally, we list all the contributions and conclude all the insights learnt from the evaluation of all frameworks.

1.1 Thesis Proposal Summary

NoSQL data-stores are essential systems that are becoming increasingly in HPC environments. These data-stores typically provide a large configuration space that control their performance and cost. Although these configurations usually come with a default setting, this default setting can rarely achieve the best performance for a given application due to the lack of knowledge about this application. Therefore, reconfiguration is usually performed by the system admins to customize the data-store behavior so that it matches the workload specific characteristics. For fast changing workloads, this problem becomes very challenging since human admins will need to re-profile the database to understand which configurations to use when the workload changes. Moreover, configuration parameters can have significant dependencies among themselves and the optimal value of one parameter can be impacted by the values of the other parameters.

Therefore, we propose a solution that automates the selection of configuration parameters for dynamic workloads. Our solution is represented in three frameworks. In the first framework, Rafiki, we show how we build a performance prediction model that can be used as a surrogate for the actual data-store instance. The model takes the workload characterization and a candidate configuration setting as inputs and predicts the performance of the database instance (in Ops/sec). This surrogate model training is essential in reducing

the searching time for new configurations, which is a key requirement to adapt with fast changing workloads. In the second framework, SOPHIA, we show that identifying the best configuration setting for a particular workload phase is not sufficient to provide globally optimized performance. The reason is that changing the configurations has a cost due to the experienced downtime observed by the servers while being reconfigured. Accordingly, changing the configurations *greedily* for every workload change can significantly reduce the overall performance over time. Therefore, we propose SOPHIA, which performs workload prediction along with Cost-Benefit analysis to identify when to perform the reconfiguration operations and achieve globally optimized performance. In the third framework, OPTIMUS-CLOUD, look into the problem of tuning the cloud configurations (i.e. VM architecture) *jointly* with the database configurations to optimize both performance and $ cost. Our analysis shows the importance of this joint optimization due to the high dependency between the architecture and the database parameters. However, it also shows that considering both configurations increases the search space significantly, which prohibits the ability to search the space fast enough to adapt with dynamic workloads. Therefore, we propose the concept of *Complete-Sets*, which OPTIMUSCLOUD relies on to partition the cluster into sub-clusters. All instances within a sub-cluster (a *Complete-Set*) share the same configurations (i.e. homogeneous), whereas different *Complete-Sets* can have different configurations. Our evaluation shows that our proposed heterogeneous configurations outperform those that are found by several prior works such as CherryPick, Selecta.

In the second line of work, we investigate optimizing the performance and cost of applications running in serverless frameworks. Such applications are often represented as Directed acyclic graphs (DAGs). Nodes in the DAG represent serverless functions, and edges in the DAG represent data-dependencies between the functions. We analyze traces from a production FaaS infrastructure to identify three characteristics of serverless DAGs. We use these to motivate and design three frameworks. (1) SONIC: targets minimizing communication latency between in-series functions. This is achieved by selecting between three data passing methods dynamically as the input size changes. (2) ORION: captures the latency distribution of the DAG, which enables providing performance SLOs. This is achieved by modeling the distribution of each function while considering the correlation between in-series

and in-parallel functions. (3) WiseFuse: provides an optimized execution plan for the user-defined DAG after performing two important optimizations: Fusion for in-series functions, and Bundling for in-parallel invocations.

2. RAFIKI: A MIDDLEWARE FOR PARAMETER TUNING OF NOSQL DATASTORES FOR DYNAMIC METAGENOMICS WORKLOADS

2.1 Abstract

High performance computing (HPC) applications, such as metagenomics and other big data systems, need to store and analyze huge volumes of semi-structured data. Such applications often rely on NoSQL-based datastores, and optimizing these databases is a challenging endeavor, with over 50 configuration parameters in Cassandra alone. As the application executes, database workloads can change rapidly from read-heavy to write-heavy ones, and a system tuned with a read-optimized configuration becomes suboptimal when the workload becomes write-heavy.

In this paper, we present a method and a system for optimizing NoSQL configurations for Cassandra and ScyllaDB when running HPC and metagenomics workloads. First, we identify the significance of configuration parameters using ANOVA. Next, we apply neural networks using the most significant parameters and their workload-dependent mapping to predict database throughput, as a surrogate model. Then, we optimize the configuration using genetic algorithms on the surrogate to maximize the workload-dependent performance. Using the proposed methodology in our system (Rafiki), we can predict the throughput for unseen

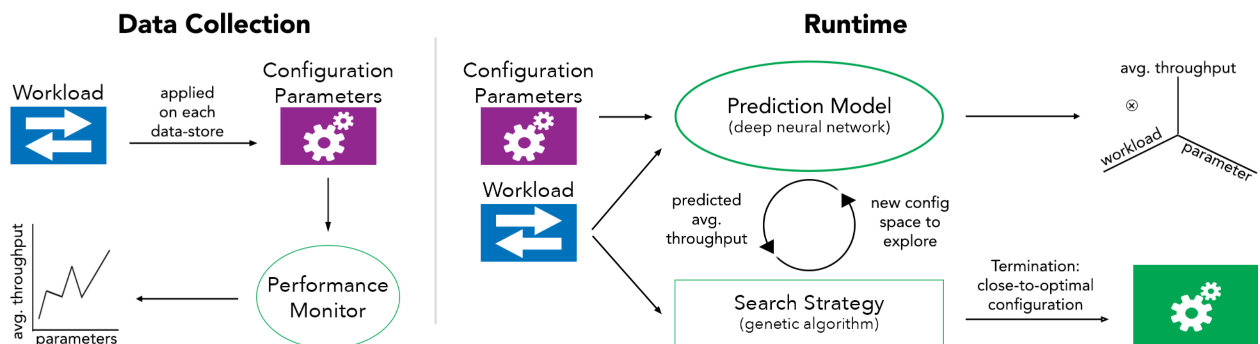


Figure 2.1. A high-level overview schematic of our proposed system Rafiki that searches through the configuration parameters' space of NoSQL datastores to achieve close-to-optimal performance.

workloads and configuration values with an error of 7.5% for Cassandra and 6.9-7.8% for ScyllaDB. Searching the configuration spaces using the trained surrogate models, we achieve performance improvements of 41% for Cassandra and 9% for ScyllaDB over the default configuration with respect to a read-heavy workload, and also significant improvement for mixed workloads. In terms of searching speed, RAFIKI, using only 1/10000-th of the searching time of exhaustive search, reaches within 15% and 9.5% of the theoretically best achievable performances for Cassandra and ScyllaDB, respectively—supporting optimizations for highly dynamic workloads

2.2 Introduction

Metagenomics applications, poised alongside other big data systems, have seen explosive data growth, placing immense pressure on overall datacenter I/O [1]. Automatic database tuning, a cornerstone of I/O performance optimization, remains a challenging goal in modern database systems [2]–[4]. For example, the NoSQL database engine Cassandra offers 50+ configuration parameter, and each parameter value can impact overall performance in different ways. We demonstrate that the performance difference between the best and worst configuration files for Cassandra can be as high as 102.5% of throughput for a read-heavy workload. Further, the optimal configuration setting for one type of workload is suboptimal for another and this results in as much as 42.9% degradation in database performance in the absence of optimized parameter versions, such as in our system RAFIKI. In this paper, we present RAFIKI[1], an analysis technique and statistical model for optimizing database configuration parameters to alleviate I/O pressures, and we test it using Cassandra, when handling dynamic metagenomics workloads.

Prior work and the state-of-practice have left several gaps in database configuration optimization that RAFIKI addresses. First, most approaches rely on expert knowledge for significant parameter selection or simply include all parameters for tuning. For example, [5] singles out a parameter from prior work surveys (over-simplification), while [6], [7] include most, if not all, parameters, with consequent time-complexity issues. In RAFIKI, we utilize

[1]↑Just as RAFIKI was the wise monkey from "The Lion King" who always knew how to avoid dead ends, we wish our system to avoid poor-performing dead-spots with its sagacity.

analysis of variance (ANOVA [8]) to identify *key parameters* from the set of all parameters for further analysis, thus, reducing the computational complexity and data collection overheads for model training. Second, many techniques utilize optimization approaches that are vulnerable to local maxima, by making linear assumptions about the performance response of each tuning parameter [9], or rely on online tuning techniques. In practice, this results in suboptimal performance and very long wall-clock convergence times due to the overhead of performance metric collection (minutes per trial). RAFIKI utilizes *trained surrogate models* for performance to enable rapid searching via stochastic models to mitigate local maxima concerns. Finally, most approaches do not account for *dynamic workload changes* when tuning. For example, [10], [11] require over 30 minutes to adapt to new workloads. RAFIKI includes workload characteristics directly in its surrogate model so that large step changes in workloads are rapidly met with large step changes in configuration parameters. We find that this last piece is crucial in big data applications, such as in metagenomics, where key reuse distance is large, putting large pressures on the hard disk, and the ratio of read-to-write queries changes rapidly during different phases of its data manipulation pipeline.

We apply RAFIKI to database traces from Argonne National Lab's MG-RAST[2] system, the most popular metagenomics portal and analysis pipeline. The field of metagenomics encompasses the sequencing and analysis of community microbial DNA, sampled directly from the environment. In recent years, DNA sequencing has become significantly more affordable and widespread, being able to sequence metagenomic samples more efficiently, affording the exploration of microbiomes in different ecosystems, including in the human gut glasner2017finding in different clinical manifestations, strongly motivating research in metagenomics blow2008metagenomics. Consequently, metagenomics analysis have come to the fore in the realm of big data systems for personalized medicine applications. Big data systems must deal with ever-growing data volumes and access patterns that are rising fast and exhibit vast changes, respectively. In MG-RAST, for example, user counts and data volumes have grown consistently over the past 9 years, and the repository today has roughly 280k total metagenomes, of which 40,696 are public, containing over a trillion sequences and 131.28 Terabasepairs (Tbp) [12], using about 600 TB resources. Such growth has placed significant

[2]↑http://metagenomics.anl.gov

pressure on I/O [13]. The system allows multiple users to insert new metagenomes, analyze existing ones, and create new metadata about the metagenomes—operations that mirror many big data applications where new data is inserted and existing data is analyzed continuously. Such processes, especially in a multi-user system like MG-RAST, result in highly dynamic database workloads with extended periods of mixed read-write activity, punctuated by bursty writes, and a dynamic mix of reads and writes during the longest periods. Such accesses are atypical of the archetypal web workloads that are used for benchmarking NoSQL datastores, and consequently, the default configurations woefully under-perform RAFIKI's optimized solutions. For example, due to the volume of data and the typical access patterns in MG-RAST, key re-use distance is very large and this puts immense pressure on the disk, while relieving pressure on caches.

We, therefore, felt the need for the design of the RAFIKI middleware that can be utilized by NoSQL datastores for tuning their configuration parameters at runtime. Our solution RAFIKI, as shown in Figure 7.3, utilizes supervised learning to select optimal configuration parameters, when faced with dynamic workloads. In the first phase, the configuration files are analyzed for parameters-of-interest selection. Of the 50+ configurations in Cassandra, for example, we find that only 5 significantly impact performance for MG-RAST. To discover this, workload-configuration sets are applied via benchmark utilities, and the resulting performance measures collected from a representative server. We determine significant configuration parameters by applying ANOVA analysis to identify performance-parameter sensitivities. Even with this restricted set of sensitive parameters, the search space is large and an exhaustive search for optimal configuration settings is infeasible. For example, for our selected set of 5 parameter settings, the search space conservatively has 25,000 points. Doing this exhaustive search means running the server for a given workload-configuration combination for a reasonable amount of time (such as, 5 minutes), thus giving a search time of 2,080 hours—clearly infeasible for online parameter tuning.

In the second phase, we train a deep neural network (DNN) to predict performance as a function of workload and configuration. For data collection, we collect benchmark data for a random subset of the possible configurations. Each configuration is run against multiple workloads so that experimentally, a workload+configuration map to a performance metric.

We collect a relatively sparse set of samples and utilize regularization techniques to prevent over-fitting of a DNN-based regression model. From this model, we create a surrogate for performance—given a new configuration and a new workload, the DNN will predict the performance of the database system. In the final phase, we use the surrogate model, in conjunction with a genetic algorithm (GA), to search for the optimal configuration.

By the domain-specific choice of the fitness function, we are able to get close-to-maximum throughput for any given workload characteristic.

Rafiki is evaluated against synthetic benchmarks that have been tailored to match the query distribution of the sample MG-RAST application. Trace information from MG-RAST, measured over a representative 4 day period, was analyzed to generate accurate representations in the benchmarking tool. Additionally, some sub-sampling of the trace was used to measure performance in a case study. A second datastore, ScyllaDB, which is based on Cassandra, is used to show the generality of the performance tuning middleware for NoSQL datastores. Rafiki's ability to outperform the ScyllaDB's internal optimizer demonstrates its improvement over the state-of-the-practice. Rafiki is able to increase Cassandra's throughput compared to the default configurations settings, by 41.4% for read-heavy workloads, 14.2% for write-heavy workloads, and 35% for mixed workloads. The DNN-based predictor can predict the performance with only 5.6% error, on average, when confronted with hitherto unseen workloads and with only 7.5% error for unseen configurations. The improvements for ScyllaDB are more modest because of its internal self-tuning feature—averaging 9% for read-heavy and read-only workloads. Finally, by comparing with an exhaustive grid search, we show that the performances achieved by our technique, using only 1/10,000-th of the searching time of exhaustive search, is within 15% and 9.5% of the theoretically best achievable performances for Cassandra and ScyllaDB, respectively.

The primary claims to novelty of our work can be summarized as follows:

1. We demonstrate that the performance of the NoSQL engines can change significantly with respect to the applied workload characteristics (*e.g.*, read-to-write proportions), due to workload-dependent procedures, such as compaction. Specifically, we identify

that for metagenomics workloads, it is important to dynamically and quickly vary a set of configuration parameters to achieve reasonable performance.

2. We create a novel technique to predict the performance of a NoSQL engine for unseen workload characteristics and unseen configurations. This is challenging due to the non-linear nature of the dependence of performance on these and due to the inter-dependent nature of various configuration parameters. This serves as a surrogate model for us to do a search through the large space of configuration parameter settings.

3. Our middleware, called Rafiki, can search efficiently, using genetic algorithms, through the parameter space to derive close-to-optimal parameter settings, while using only 0.28% of the time of an exhaustive search. Our search is agile enough that it can be quickly re-done when the workload characteristics change.

4. We identify that the chief configuration parameters are related to compaction, such as, when to combine multiple data tables on disk into one and how many levels of tables to maintain. The relation between compaction-related configuration parameters and performance is non-monotonic, while the parameter space is infinite with both continuous and integer control variables. Our ANOVA analysis backs up these claims.

The rest of the chapter is organized as follows. In Section 2.3, we cover the fundamental pieces of the optimization problem along with an introduction to MG-RAST's design and implementation. In Section 2.4, we describe the pieces of our solution (Figure 7.3). Section 2.5 contains the implementation and analysis of our approach. Section 2.6 provides the conclusion.

2.3 Background

This section covers background material useful to understanding the method by which we improve database performance.

2.3.1 NoSQL datastores

NoSQL datastores are key value stores that have begun to replace traditional, transactional database systems (MSSQL/MySQL/PostgreSQL) due to performance gains from the relaxation of transaction requirements, *i.e.*, the ACID properties [14], promising scalability beyond what is possible in transaction-based SQL systems. Since metagenomics and many other big data applications can tolerate a certain degree of lack of consistency, we can instead prioritize availability and partition-tolerance [15]. For example, having a slightly outdated copy of a genome may result in less pattern matches, but the matches that are found are still valid results.

In this paper, we focus on Cassandra, an Apache Foundation[3] project, which is one of the leading NoSQL and distributed DBMS driving many of today's modern business applications, including such popular users as Twitter, Netflix, and Cisco WebEx. We also consider a recent competitor, ScyllaDB, which claims order of magnitude performance improvements over Cassandra.

Of particular relevance to our use case of metagenomics is that Cassandra has a decentralized architecture in which any node can perform any operation, by virtue of a peer-to-peer topology in which all server instances are connected. This allows for horizontal scaling to accommodate a large number of processing servers. This is an important consideration for genomics processing because the individual processing steps can be quite computationally demanding, *e.g.*, it is computationally expensive to search a queried protein among a large number of functionally annotated proteins in a datastore but is also highly parallelizable.

Cassandra follows the principle of the *CAP theorem* [15], which states that a distributed system can provide efficiently at most two of the following three properties—consistency (C), availability (A), and partition tolerance (P). Cassandra sacrifices the consistency in-order to preserve the other two.

For this work, we will focus on Cassandra and ScyllaDB, a competitor that claims improved performance.

[3]↑Cassandra V3.7 http://cassandra.apache.org

2.3.2 Cassandra Overview: Key Features

This section describes the key features of Cassandra that become prime focus areas for performance optimization.

Write Workflow

Write (or update) requests, a key performance bottleneck in bioinformatics, are handled efficiently in Cassandra using some key in-memory data structures and efficiently arranged secondary storage data structures [16]. We describe them next. As shown in figure 2.2,

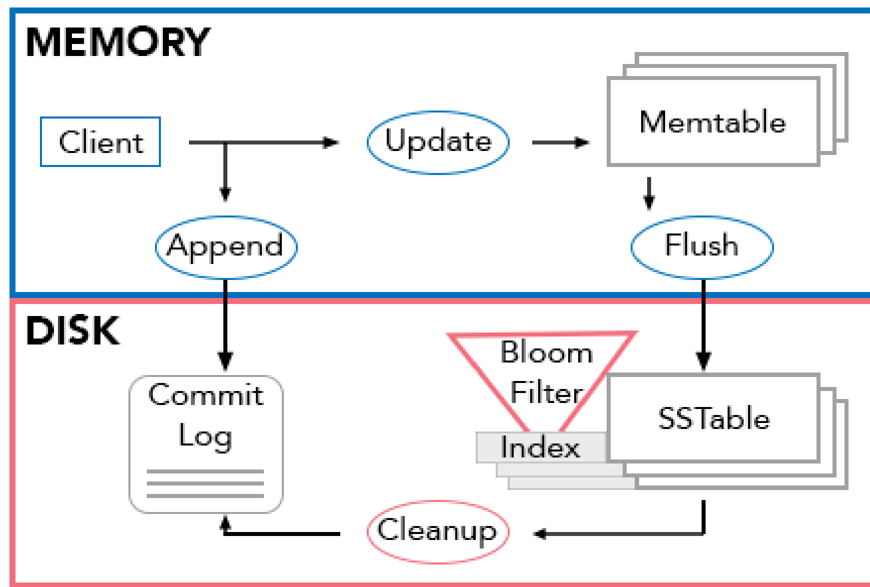


Figure 2.2. Write workflow overview

when a write request arrives, it is appended to Cassandra's CommitLog, a disk-based file where uncommitted queries are saved for recovery/replay. Then the result of the query is processed into to an in-memory data structure called the Memtable. A Memtable functions as a write-back cache of data rows that can be looked up by key – that is, unlike a write-through cache, writes are batched up in the Memtable until it is full, when it is flushed (the trigger and manner of flushing are controlled by a set of configuration parameters, which form part of our target for optimization). Each flush operation transfers these contents to the

secondary storage representation, called SSTables. SSTables are immutable and every flush task produces a new SSTable. The data for a given key value may be spread over multiple SSTables. Consequently, if a read request for a row arrives, all SSTables (in addition to the Memtable) have to be searched for portions of that row, and then the partial results combined. This is an expensive process in terms of execution time, especially since SSTables are resident in secondary storage. Over time, Cassandra may write many versions of a row in different SSTables and each version may have a unique set of columns stored with a different timestamp. Cassandra periodically merges SSTables and discards old data in a process called *compaction* to keep the read operation efficient. The compaction process merges keys, combines columns, evicts tombstones, consolidates SSTables, and creates a new index in the merged SSTable.

Compaction

Cassandra provides two compaction strategies, that can be configured on the table level. The default compaction strategy "Size-Tiered Compaction" triggers a new compaction process whenever a number of similar sized SSTables exist, while "Leveled Compaction" divides the SSTables into hierarchical levels. **Size-Tiered Compaction:** This compaction strategy activates whenever a set number of SSTables exist on the disk. Cassandra uses a default number of 4 similarly sized SSTables as the compaction trigger, whereas ScyllaDB triggers a compaction process with respect to each flush operation. At read time, overlapping SSTables might exist and thus the maximum number of SSTables searches for a given row can be equal to the total number of existing SSTables. While this strategy works well with a write-intensive workload, it makes reads slower because the merge-by-size process does not group data by rows. This makes it more likely that versions of a particular row may be spread over many SSTables. Also, it does not evict deleted data until a compaction is triggered. **Leveled Compaction:** The second compaction strategy divides the SSTables into hierarchical levels, say L0, L1, and so on, where L0 is the one where flushes go first. Each level contains a number of equal-sized SSTables that are guaranteed to be non overlapping, and each level contains a number of keys equal to 10X the number of keys at the previous

level, thus L1 has 10X the number of keys as in L0. While the keys are non-overlapping within one level, the data corresponding to the same key may be present in multiple levels. Hence, the maximum number of SSTable searches for a given key (equivalently, row to be read) is limited to the number of levels, which is significantly lower than the number of possible searches using the size-tiered compaction strategy. One drawback of the leveled compaction strategy is that the compaction is triggered each time a MEMTable flush occurs, which requires more processing and disk I/O operations to guarantee that the SSTables in each level are non overlapping. Moreover, flushing and compaction at any one level may cause the maximum number of SSTables allowed at that level (say Li) to be reached, leading to a spillover to Level L(i+1). Qualitatively it is known [17] that size-tiered compaction is a better fit for write-heavy workloads, where searching many SSTables for a read request is not a frequent operation.

2.3.3 Design Distinctions in ScyllaDB

ScyllaDB has the following modifications to Cassandra that are relevant to our work. First, ScyllaDB is written in C++, while Cassandra is written in Java. This is meant to overcome the side effects of running inside the Java Virtual Machine (JVM) such as garbage-collection, which could encounter stop-the-world pauses and impact overall performance. Second, ScyllaDB uses a different row caching methodology, which allows for caching arbitrary rows. In contrast, Cassandra's row caching (disabled by default) allows for caching sequential rows only. However, Cassandra depends mainly on caching keys-to-partition maps to save secondary storage seeks, in a structure called the *key cache*, where ScyllaDB caches actual row data. Finally, ScyllaDB distributes its operations among several engines, while each engine has its own isolated resources. Such isolation aims to avoid heavy-duty locking and unlocking and is geared toward large multi-core machines.

2.3.4 Performance metrics: Throughput and Latency

In this paper, we use mean throughput to measure the performance of the datastore. Mean throughput represents the average number of operations the system can perform per

second and this is meant to demonstrate the capacity of the database system to support MG-RAST workflow. Some database systems focus on latency metrics that represent the system's response time as observed to the client, but our workload is not latency sensitive, but rather is throughput sensitive. Therefore, our methodology will consider the optimum mean throughput.

2.3.5 Genomics Workloads

MG-RAST is the world's leading metagenomics sequence data analysis platform developed and maintained at Argonne National Laboratory [12]. Since its inception, MG-RAST has seen growing numbers of datasets and users, with the current version hosting 40,696 public and 279,663 total metagenomes, containing over a trillion sequences amounting to 131.28 Tbp. The derived data products, i.e., results of running the MG-RAST pipeline, are about 10 times the size of the submitted original data, leading to a total of 1.5 PB in a specialized object store and a 250 TB subset in an actively used datastore.

Workload Dynamism

Figure 2.3 shows the relative ratio of read to write (and update) queries for 4 days of MG-RAST workload. Qualitatively, we observe that there are periods of read heavy, write heavy, and a few mixed during the observed period. More importantly, the transition between these periods is not smooth and often occurs abruptly and lasts for 15 minutes or less. These frequent oscillations make it difficult for online systems to adapt because the transients are very sharp and clearly, static schemes, such as staying with the default configuration settings, will lead to poor performance.

Processing Pipeline

MG-RAST accepts raw sequence data submission from freely-registered users and pipelines a series of bioinformatics tools to process, analyze, and interpret the data before returning analysis results to users. Bioinformatics tools in MG-RAST are categorized into: filtering and quality control, data transformation and reduction (such as, gene prediction and RNA

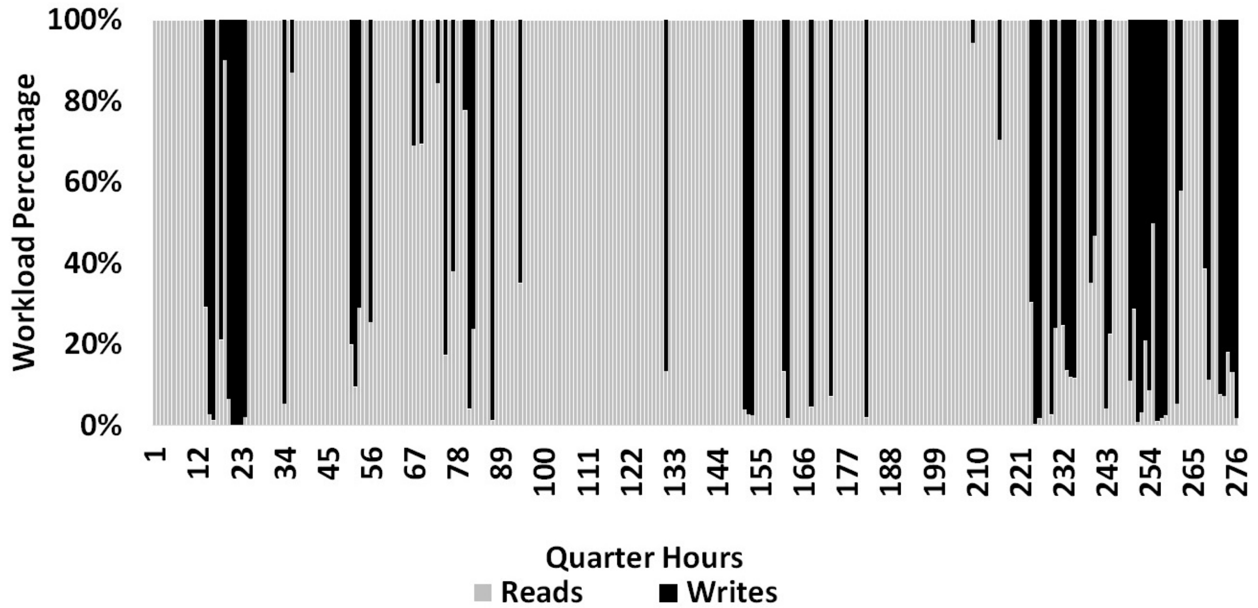


Figure 2.3. Patterns of workload for MG-RAST. The workload read/write ratios are shown for 15 minutes intervals.

detection), and data analysis and interpretation (such as, protein identification and annotation). Many independent users load, process, and store data through these steps, and each user's task may run concurrently with others in the system. Each step accesses elements of a common (genetic) community dataset.

Some pipeline stages break DNA sequences into many overlapping subsequences that are re-inserted into the datastore, increasing processed outputs by a factor of 10 over initial datasets. Moreover, metagenomics often involves decisions that cannot be made until late-stage pipeline passes, creating persistently large working sets. Thus, in this analysis of NoSQL systems, we are driven by the pressing needs of metagenomics platforms, such as MG-RAST.

2.4 Methodology

This section describes our proposed methodology for creating the middleware for configuration tuning of NoSQL datastores. We start off with the end-to-end workflow, which incorporates both the offline training and the online optimal configuration parameter choice in response to workload changes. Then, we describe the details of each step of the workflow.

2.4.1 Rafiki Workflow

The Rafiki middleware has the following workflow:

1. **Workload Characterization**: The database workloads are parametrized and expected workloads are injected into the system during data collection. The output metrics, in response to these injected workloads, are measured. In our domain, throughput is the metric of interest.

2. **Important Parameter Identification**: All of the performance related control parameters are identified, and each control parameter is independently varied, measured, and ranked in order of importance using the ANOVA technique. The top ranking parameters are designated as "key parameters" and used for subsequent stages of the workflow.

3. **Data Collection**: Now targeted training runs are carried out by varying the values of only the key parameters so that their interdependent impact on the output metric can be collected for further analysis.

4. **Surrogate Modeling**: The effect of the configuration parameters on the output metrics of interest for any given workload characteristic, is modeled using a Deep Neural Network (DNN).

5. **Configuration Optimization** (Online stage): This occurs while the NoSQL system is operational. In this stage, the optimal configuration parameters for the actual observed workload is identified via a Genetic Algorithm (GA). The GA, to explore a particular point in the search space, does not need to execute the application but queries the surrogate model, which is much faster.

Using this entire workflow, Rafiki generates optimal configuration for a specific server architecture (CPU speed, IO bandwidth, IO capacity, memory capacity, etc.). Rafiki can be used by a NoSQL datastore engine and it is agile enough to converge to new optimal settings in the face of changing workloads.

2.4.2 Notation

Here, we introduce some notation that we will use for the rest of the section. Each database has a set of parameters $P = \{p_1, p_2, \cdots, p_J\}$, where J is the number of "key parameters", *i.e.*, those that impact performance to a given statistically significant level. Each parameter p has constraints on its values, either programmatically via software-defined limits, or pragmatically, via hardware or application feasibility. It also has a *default value* that is specified by the database software at distribution. The configuration is a set of parameter values $C = \{v_1, v_2, ...v_J\}$, where value v_i corresponds to parameter p_i. In shorthand, we define a configuration by the set of values that have changed from the default settings, *e.g.*, $C = \{v_1 = 5, v_3 = 9\}$ implies that v_2 has its default value. Each parameter p_i has a number of possible distinct values n_i, which may be infinite if the domain of the parameter is in the set of real numbers $\mathcal{R}$. In these cases, the value may be quantized into a finite space. The total number of possible configurations is then $\prod_{\text{i}=1}^{J} n_\text{i}$.

2.4.3 Workload Characterization

In the first step of Rafiki, we characterize the application workload in order to apply synthetic benchmarking utilities to drive our system. We use two key parameters to characterize the workload:

- *Read Ratio (RR):* the ratio between the number of read queries and the total number of queries.
- *Key Reuse Distance (KRD):* the number of queries that pass before the same key is re-accessed in a query.

The RR is crucial because the read and write workflows inside of the database take different paths with different tuning parameters that influence performance. The KRD is crucial when measuring the importance of various levels of cache since if keys are rarely reused, then caching is of limited value.

The granularity of time window over which the RR will be measured is dependent on the application. Conceptually, this time interval should be such that the RR statistic is

stationary, in an information-theoretic sense [18]. For the MG-RAST workload, we find that a 15 minute interval satisfies this property. A visual representation can be seen in Figure 2.3. For the KRD, we define a very long window of time over which the key reuse is studied. We then fit an exponential distribution to summarize this metric and use that for driving the benchmarking for the subsequent stages of the workflow. For the MG-RAST case, we use the entire 4-day period over which we have the detailed information about the queries to calculate the KRD statistic. Operationally, it is challenging to get the very detailed information about queries that is required for calculating KRD. This is because of two factors. First, the logging of the exact queries puts a strain on the operational infrastructure, especially in a production environment (MG-RAST in our case), and second, there are privacy concerns with this information, especially in the genomics domain. Hence, for practical purposes, the window of time over which the KRD is calculated, is bounded.

2.4.4 Important Parameter Identification

Cassandra[4] has over 25 performance-related configuration parameters, and in this piece of our solution approach, we identify which of these are the "key parameters". We will then include these as features in Rafiki's surrogate model of the performance of the NoSQL datastore.

Key Parameters

Both Cassandra and ScyllaDB have over 50 configuration parameters, around half of which are related to performance tuning [19]. It is not feasible to search the space of all these parameters in order to identify the optimal parameter settings and instead some pruning is needed. For this, we seek to identify the "key parameters" *i.e.*, the ones that affect performance of the application in a statistically significant manner. For this, we apply an Analysis of Variance test (Anova [8]) to each parameter individually, while fixing the rest of the parameters to their default values. For example, $C_1 = \{v_1 = 5, v_2 = \text{def}, v_3 = \text{def}\}$, $C_2 = \{v_1 = 10, v_2 = \text{def}, v_3 = \text{def}\}$, and $C_3 = \{v_1 = 15, v_2 = \text{def}, v_3 = \text{def}\}$ will be

[4]↑http://docs.datastax.com/en/cassandra/3.0/cassandra/configuration/configCassandra_yaml.html

used to collect three sample points S_1, S_2, S_3. This parameter p_1 will be scored with the $var(S_1, S_2, S_3)$, and the other two parameters will also be scored in a similar way.

Afterward, we select the top parameters with respect to variance in average throughput. We find empirically that there is a distinct drop in the variance when going from top-k to top-$(k+1)$ and we use this as a signal to select the top-k parameters as the key parameters. For Cassandra, RAFIKI identifies the following list of parameters as the key parameters:

[topsep=0pt,itemsep=0ex,partopsep=1ex,parsep=1ex]*Compaction Method (CM)*: This takes a categorical value between the available compaction strategies: Size-Tiered or Leveled[5]. Size-Tiered is recommended for write-heavy workloads, while Leveled is recommended for read-heavy ones. *Concurrent Writes (CW)*: This parameter gives the number of independent threads that will perform writes concurrently. The recommended value is 8 × number of CPU cores. *file_cache_size_in_mb (FCZ)*: This parameter controls how much memory is allocated for the buffer in memory that will hold the data read in from SSTables on disk. The recommended value for this parameter is the minimum between 1/4 of the heap size and 512 MB. *Memory table cleanup threshold (MT)*: This parameter is used to calculate the threshold upon reaching which the MEMTable will be flushed to form an SSTable on secondary storage. Thus, this controls the flushing frequency, and consequently the SSTables creation frequency. The recommended setting for this is based on a somewhat complex criteria of memory capacity and IO bandwidth and even then, it depends on the intensity of writes in the application. *Concurrent Compactors (CC)*: This determines the number of concurrent compaction processes allowed to run simultaneously on a server. The recommended setting is for the smaller of number of disks or number of cores, with a minimum of 2 and a maximum of 8 per CPU core. Simultaneous compactions help preserve read performance in a mixed read-write workload by limiting the number of small SSTables that accumulate during a single long-running compaction.

These five parameters, combined, control most of the trades between read and write performance, but there is to date no indisputable recommendation for manually setting these

[5]↑The third supported option ”Time Window Compaction Strategy” is not relevant for our workload and is thus not explored. That is only relevant for time series and expiring time-to-live (TTL) workloads.

configuration parameters nor is there any automated tool for doing this. Poring through discussion forums and advice columns for the two relevant software packages [20]–[22], we come away convinced that there is significant domain expertise that is needed to tune a server for a specific workload characteristic and a specific server architecture. Furthermore, these tunings change, and sometime quite drastically, when the workload characteristics change. These lead us to investigate an automatic and generalizable method for achieving the optimal configuration parameter values.

2.4.5 Data Collection for Training Models

In our goal to optimize the server performance, we encounter a very practical problem: there are too many search points to cover them all in building a statistical model. The five key configuration parameters, as described in the previous section, even if discretized at broad levels, represent $2 \cdot 4 \cdot 8 \cdot 10 \cdot 4 = 2,560$ configurations. If this is combined with 10 potential workloads, then, there are over 25,000 combinations for sampling. With each benchmark run executing for 5 minutes, this would require almost 3 months of compute time to calculate for a single hardware architecture. The benchmark execution time cannot be reduced much below 5 minutes so as to capture the temporal variations in the application queries and remove the startup costs.

The surrogate model in the next step is trained on only a small subset of the potential search space, from which it predicts the performance in any region of the exploration space. This eliminates the need for exhaustively collecting the performance data and allows for a much faster process through the large exploration space of configuration parameters than a typical exhaustive, grid-based search. Empirically we find that training with approximately 5% of the training space (Figure 2.7) produces accurate enough model, which corresponds to the model results presented in the evaluation results.

We capture the dynamic elements of a NoSQL system with the input feature set: workloads and server configurations. For workloads, we represent the workload feature as read ratio (RR) (write ratio is 1-RR). The KRD is used to configure the data collection, but it is not provided as an input to our model as it is found to be stationary in our target metage-

nomics domain. Second, the configuration file is represented by the five key parameters, described in Section 2.4.4. We select the configuration set $\mathbb{C}$ so that, for each parameter p_i, the minimum $\underline{v}_i$ and maximum $\overline{v}_i$ value occurs at least once in the set. The default value is also included in at least one experiment, and additional experiments are added by randomly selecting other values for the parameters, but not in a fully combinatorial way.

Performance data is collected by applying the particular workload W_i and configuration to a specific server and database engine and observing the resultant performance P_i. The space of all workloads exists as $[0, 1]$, representing RR, and it is quantized into n_w discrete values, $\{W_1, W_2, ..., W_{n_w}\}$. The particular sample is defined as $S_i = \{W_i, C_i, P_i\}$. A set of such samples defines the training data for our model.

2.4.6 Performance Prediction Model

Surrogate Performance Model

Server performance is a function of workloads, configuration parameters, and the hardware specifications on which the software is executed. The relationship between these pieces is sufficiently complex so that no simple guideline can optimize them. This is well known in the performance prediction domain [4], [13] and we show this empirically for MG-RAST in Section 2.5.6. Although we qualitatively understand the impact of most of these configuration parameters on the output metric, the quantitative relation is of interest in guiding the search toward the optimal settings. It is to be reasonably expected that such a quantitative relation is not analytically tractable. We therefore turn to a statistical model of the relationship of workloads and configurations on the end-performance, for a given hardware specification. We call this the *surrogate performance model*—surrogate because its prediction acts as an efficient stand-in for the actual performance that will be observed.

Creation of the Performance Model

We assume that the relationship between performance, workload, and the database configuration is non-linear and potentially very complex. For this reason, we rely on a sophisticated deep neural network (DNN)-based machine learning model, with multiple hidden

layers, to capture the mapping. Our choice of a DNN was driven by the fact that it does not have any assumptions of linearity for either single configuration parameters, or jointly, for multiple parameters, while being able to handle complex interdependencies in the input feature space. The control parameters include the number of layers and the degree of connectivity between the layers. In fact, our experiments validate our hypothesis that the input parameter space displays significant interdependence (Figure 2.6).

One potential drawback of DNN is the peril of overfitting to the training data. However, in our case, the amount of training data can be made large, "simply" by running the benchmarks with more combinations of input features (a very large space) and workloads (a large space). Therefore, by using different, and large-sized, training datasets and the technique of Bayesian regularization, we can minimize the likelihood of overfitting. Bayesian regularization helps with reducing the chance of overfitting and providing generalization of the learned model. On the flip side, we have to make sure that the network is trained till completion, *i.e.*, till the convergence criterion is met, and we cannot relax the criterion or take recourse to early stopping. Additionally, to improve generalizability, we initialize the same neural network using different edge weights and utilize the average across multiple (20) networks. Further, we utilize simple ensemble pruning by removing the top 30% of the networks that produce the highest reported training error. The final performance value would be an average of 14 networks in this case.

The output of the model is a function that maps {W, C} (workloads and configurations) to the average database operations per second:

$$AOPS_{general} = f_{\text{net}}(W, C) \tag{2.1}$$

$$AOPS_{Cassandra} = f_{\text{net}}(RR, CM, CW, FCZ, MT, CC) \tag{2.2}$$

Where: AOPS is the average operations per second that the server achieves, RR is the read ratio describing the workload, and the configuration parameters are CM/CW/FCZ/MT/CC.

In this paper, we utilize Bayesian regularization with back-propagation for training a feed forward neural network to serve as f_{net}. The network size itself can be selected based upon the complexity of the underlying database engine and server and increased with the size of the input dataset to better characterize the configuration-workload space. Section 2.5.3 further covers our approach for the MG-RAST/Cassandra case.

2.4.7 Configuration Optimization

Optimal Configuration Search

An exhaustive search could take three months or more of compute time to complete for a single architecture. To reduce this search time, we collect only hundreds of samples (compute hours vs. months) to train f_{net} and use this to predict performance. In doing so, we lose some precision in the configuration space, but we gain the ability to use less efficient but more robust search techniques (*e.g.* genetic algorithms) without an explosion in compute time. For example, in our experiments Rafiki utilizes 3,500 calls to the surrogate model for a single workload data point, so that a few seconds of searching represents 12 days of sampling. Using this approach, we find optimal configurations in Rafiki.

The goal of the database administrator is to maximize server throughput for a given workload:

$$C_{opt} = \arg\max_{C} \text{AOPS}(W, C) \tag{2.3}$$

Where C represents the configuration parameters (CM, CW, FCZ, MT, CC) and AOPS is the performance. Using the surrogate model, AOPS can be replaced by f_{net} to create an approximate optimization problem:

$$C_{opt} = \arg\max_{\text{C}} f_{\text{net}}(\text{W}, \text{C}) \tag{2.4}$$

Where f_{net} is from Equation (2.2). This substitution allows for rapid objective function evaluation.

Optimization using Genetic Algorithms

Since we hypothesize the relationship between the configuration parameters is non-linear and non-continuous (integer parameters CM, CW, and CC), finding optima in this space will require at least some searching. For a general and powerful approach, we use a Genetic Algorithm (GA) to search this space.

We formulate the GA as follows. The fitness function values is the result of f_{net} with the workload parameter W fixed and the C parameters as the input, with higher value denoting a better configuration. The parameters are constrained by the practical limits of the configuration values and as integers where necessary. The initial population is selected to be uniformly random within these bounds. The crossover function calculates intermediate configurations within the bounds of the existing population (to enforce interpolation rather than extrapolation) by taking a random-weighted average between two points in the population. For example, if $C_1 = 3, 5, 7$,$C_2 = 2, 4, 6$, then the crossover output $C_3 = \{\frac{r_1 \cdot 3 + (1-r_1) \cdot 2}{2}, \frac{r_2 \cdot 5 + (1-r_2) \cdot 7}{2}, \frac{r_3 \cdot 7 + (1-r_3) \cdot 6}{2}\}$, where r_1, r_2, r_3 are chosen uniformly random between 0 and 1.

The fitness function is modified to ensure constraints are met, as described in [23], [24], where infeasible configuration files are scored with a penalty, and feasible ones are scored as the original fitness function (performance). For example, from above, if $r_1 = 0.3$ and p_1 is an integer parameter, then C_3's $v_1 = 1.15$. The performance of this point will be penalized by adding a penalty factor because v_1 is not an integer for C_3.

Solving our system using a GA creates a blackbox model, rather than the more desired interpretable model. It may be argued that an interpretable model is particularly important to a system administrator whereby she can estimate what some parameter change will likely do to the performance of the NoSQL datastore. With this goal we experimented with an interpretable model, the decision tree, with the node at each level having a single decision variable, one configuration parameter. We found that this was woefully inadequate in modeling the search space, giving poor performance. When each node was allowed to have a linear combination of the parameters, the performance improved and this is beginning to move toward a less interpretable model. So it appears that for this particular problem, we

have to sacrifice interpretability to gain higher expressivity and therefore higher prediction accuracy from the model.

2.4.8 DBA level of intervention

Although our proposed system provides automatic tuning for the underlying data store, some limited DBA intervention is still needed to support Rafiki with the following items:

1. Performance metrics: What application-specific performance metric should be considered for tuning (throughput, latency, etc.).
2. Performance parameters: List of performance influencing parameters with valid ranges. Excluding security, networking, and consistency related parameters.
3. Representative application trace: A workload for Rafiki to use for characterization as described in Section 2.4.3.

2.5 Experiments and Results

In this experimental section we seek to answer the following questions:

1. How effective are the selected parameters in capturing the performance (Average Throughput), and what are the interdependencies among them?
2. What is the sensitivity of the datastore to workload pattern, with the default configurations?
3. Can we accurately predict the performance of the NoSQL datastore for unknown workloads and configurations, *i.e.*, how good is our surrogate model?
4. Finally, the overall goal of Rafiki, can we efficiently search for optimal configuration parameters and improve the performance of the NoSQL datastore? Does this improvement scale to multiple servers?

2.5.1 Workload Driver and Hardware

We used a Dell PowerEdge R430 as the server machine in the single-server and single-client setup. This server has 2x Intel Xeon E5-2623 v3 3.0 GHz 4-core, 32GB RAM (2x 16GB), and 2x 1TB magnetic disk drives (PERC H330 Mini) mirrored, each drive supporting 6 Gbps. The client machine is Opteron 4386, 8 cores at 3.1 GHz each, 16 GB DDR3 ECC memory. The client and the server are directly connected through a 1 Gbps switch, ensuring that the network is never a bottleneck for our experimental setup. We use a single-server single-client setup for all of the following experiments, except for the multi-server experiment where multiple clients were used to benchmark the multiple servers that were connected in a peer-to-peer cluster.

For all the following experiments, we modified the Yahoo Cloud Serving Benchmark (**YCSB** [25]) tool for emulating MG-RAST simulated workloads to Cassandra. We use YCSB only as a harness to drive the experiments and collect metrics, while all the workload-specific details (query patterns, payload size, key reuse distance, etc.) are derived from actual MG-RAST queries. The simulated queries are based on the most frequent queries submitted to MG-RAST, selected from the MG-RAST query logs, with keys selected from used data-shards. The output is the average throughput measured in operations per second (Section 2.3.4).

2.5.2 Data Collection

We orchestrated Cassandra and ScyllaDB inside of Docker containers so that the statefulness of the database is easily maintained. Between data collection events, the server is reset to prevent any caching or persistent information from influencing subsequent benchmarks. For each data point, a fresh Docker container with Cassandra is started and a corresponding YCSB “shooter” then loads the Cassandra server. Multiple shooters are used for a particular server to ensure that it is adequately loaded.

In our experiments, we use 11 different workloads spanning 10% increments between 0% and 100% reads. The number of configurations `\mathbb{C}` $= 20$, resulting in $11 \cdot 20 = 220$ total data points. Each performance point was measured as the average throughput over a

5-minute long benchmarking period. In some cases, if an experiment fails prematurely or other activities add noise to the data collection, then the impacted samples can be removed—20 noisy/faulted samples were removed in our dataset, due to faults in the load-generating clients, thus leaving 200 total samples.

2.5.3 Surrogate Model and Training

As described in section 2.4.6, the surrogate model (DNN) was trained and tuned with a hidden layer size of $[14, 4]$ based on trial and error, and the ensemble size for the model was selected at 20 networks during all our experiments. For the final experiment where Rafiki selects the optimal parameter settings using GA, we use 100 neural nets since it is still fast enough and gives a moderate improvement. The final DNN architecture has 6 input parameters into the first hidden layer with 14 neurons, a second hidden layer with 4 neurons, and a single output layer and 1 output value. The neurons are connected in a feed-forward setup so that the output of one layer is the input to the next. All experiments utilize 75% training and 25% test unless otherwise mentioned. When we evaluate our model with unseen configurations, we group the data points by configuration, i.e., for each configuration C_{i}, there exists 11 W workloads, and similarly for unseen workloads. This division is along the configuration or workload dimension—unseen configuration means that no entries for C_{i} seen in the test set exists in the training set.

We rely on MATLAB's Neural Network Toolbox [26] to construct and train such networks. To train the model, we utilize Bayesian Regularization (`trainbr` in MATLAB). We prevent overfitting by training until convergence or 200 epochs, whichever comes first, and averaging the output of the ensemble of networks, each with different initial conditions, for each validation set. This regularization technique automatically reduces the effective number of parameters in the model at the expense of increasing the training time.

2.5.4 Cassandra's Sensitivity to Dynamic Workloads

In this experiment, we wish to see how Cassandra's baseline performance is affected by changing nature of the workloads. Figure 2.4 shows how Cassandra's default performance is

highly sensitive to its workload, demonstrating that the average throughput decreases with respect to the increase in workload's read proportion—the swing going from R=0/W=100 to R=100/W=0 is above 40%. This observed behavior is consistent with Cassandra's write path and compaction technique described in Sections 2.3.2 and 2.3.2. Specifically, Cassandra uses a size-tiered compaction strategy, which is triggered whenever a number of similar sized SSTables are created (4 by default). These are not consolidated to ensure that keys are non-overlapping among them. Hence search for a specific key through a read query causes Cassandra to search in all the SSTables causing the lowered read performance. Therefore, tuning Cassandra's performance under read-heavy workloads is essential, especially because we frequently observe read-heavy metagenomics workloads as shown in Figure 2.3.

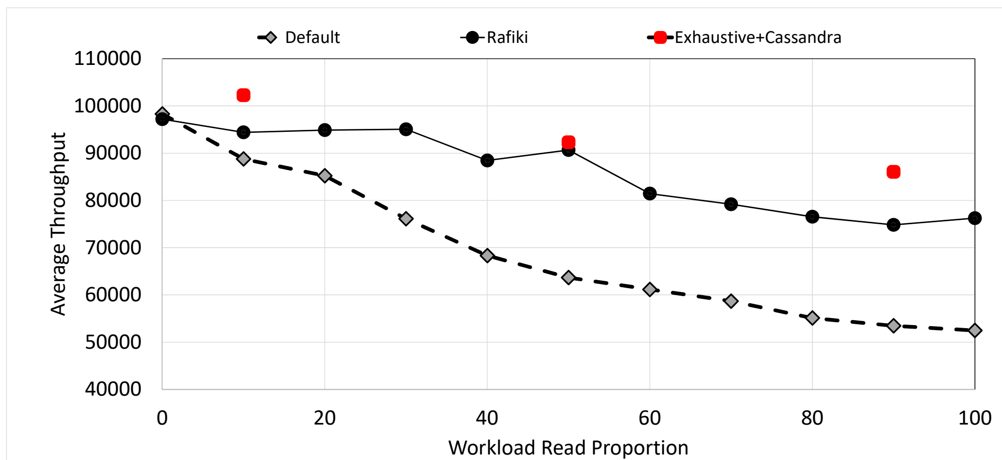


Figure 2.4. Performance of Cassandra with optimal configuration selected by RAFIKI vs. Default configuration. Also three points are shown for the theoretically optimal performance using exhaustive searching.

2.5.5 Key Parameters Selection

As described in Section 2.4.4, tuning all configuration parameters for Cassandra is impractical because of the combinatorial explosion of the number of possible configuration

sets. Therefore, we apply the ANOVA statistical technique to identify the most significant set of configuration parameters that affect Cassandra's performance, which we call the "key parameters". We vary the value of each parameter individually, while fixing the values of the rest of the parameters to their default. For categorical parameters (*e.g.*, Compaction Strategy) all possible values are tested. Whereas for numerical parameters (*e.g.*, memtable_cleanup_threshold, and concurrent_reads), a number of values (4) are tested as described in Section 2.4.4. Figure 2.5 shows the standard deviation in throughput for the top 20 configuration parameters. The most significant parameter, Compaction Strategy has standard deviation 11X that of concurrent writes, and thus removed from the figure for better visualization. From Cassandra's configuration description, we observed that the fifth configuration parameter –memtable_cleanup_threshold– controls the flushing frequency. Cassandra uses this parameter to calculate the amount of space in MB allowed for all MEMtables to grow. This amount is computed using the values of memtable_flush_writers, memtable_offheap_space_in_mb, and memtable_cleanup_threshold. Thus we skip the second and third configuration parameters and only include memtable_cleanup_threshold to control the frequency of MEMtables flushing, having 5 different parameters in total as shown in Equation 2.1.

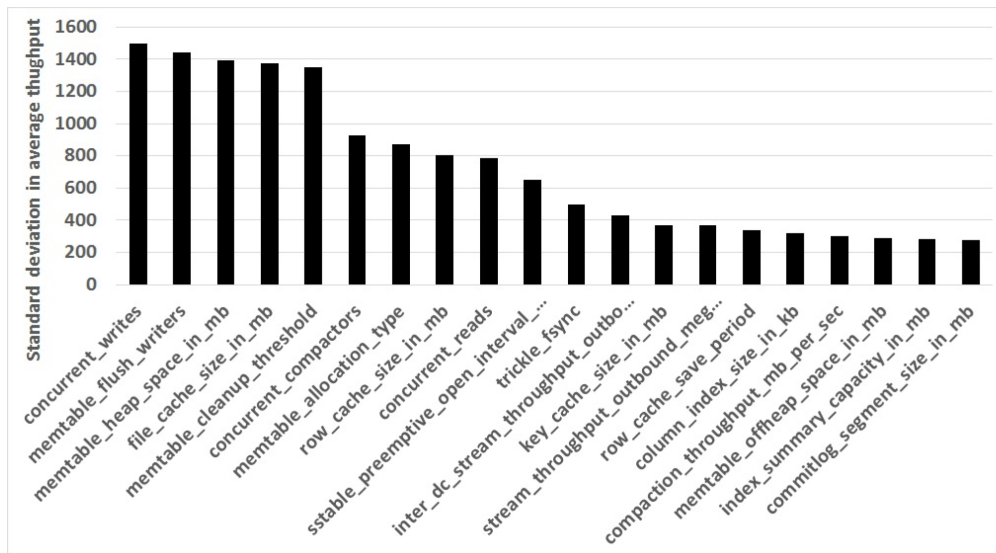


Figure 2.5. ANOVA analysis for Cassandra to identify the key parameters, *i.e.*, the ones that most significantly control its performance.

Table 2.1. Cassandra maximum, minimum, and default throughputs as the key-configuration parameters are varied

	Maximum	Default	Minimum
Average Throughput (read=90%)	78,556	53,461	38,785
Improvement % over Minimum	102.5%	27.4%	-
Average Throughput (read=50%)	89,981	63,662	53,372
Improvement % over Minimum	68.5%	16.16%	-
Average Throughput (read=10%)	102,259	88,771	78,221
Improvement % over Minimum	30.7%	11.8%	-

2.5.6 Effectiveness of Selected Configuration Parameters

In this experiment, we assess the effectiveness of the key configuration parameters selected by our statistical analysis. We start by collecting the average throughput for the 220 configuration sets. From the collected data, we examine how impactful the change in performance is compared to the default configuration setting. Table 2.1 shows the maximum, minimum, and default performance for three different workloads. Which shows that the selected parameters have a significant effect on performance.

One obvious way to tune such parameters is by sweeping greedily through each of them individually while fixing the values of the rest of the parameters. However, we infer that this obvious technique is suboptimal, because it ignores the interdependencies among the selected parameters. Figure 2.6 shows the effect of changing two parameters from this set: Compaction Method (CM) and Concurrent Writes (CW). Changing one parameter's value results in changing the optimal values for the other parameter. For example, doubling the value of CW from 16 to 32 has a positive effect on performance when CM is set to Size-Tiered Compaction, *i.e.*, throughput increases by 30%, but the same change has very little effect when CM is set to Leveled Compaction. However, doubling the value of CW from 32 to 64 has a negative effect on performance when CM is set to Leveled Compaction, *i.e.*, throughput decreases by 12.7%, but this time it has very little effect when CM is set to Size-Tiered Compaction. This suggests that using a greedy optimization technique, *i.e.*, tuning each configuration parameter individually, cannot find the optimal solution, and motivates us to look for a more sophisticated search strategy for finding the optimal configuration.

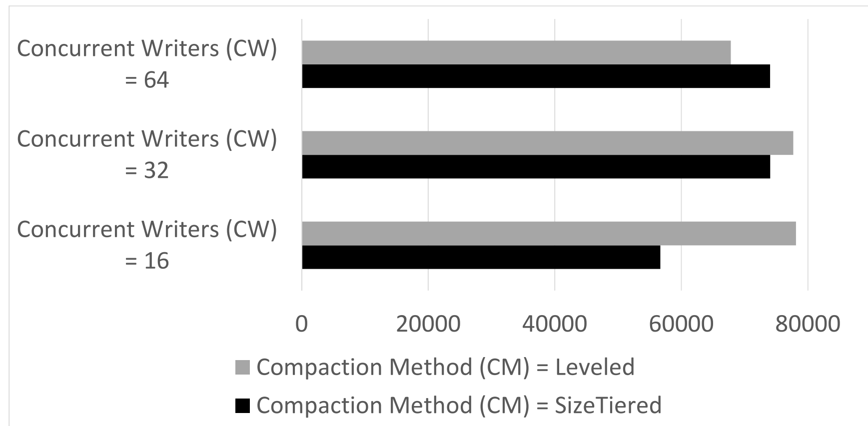


Figure 2.6. Interdependency among selected parameters necessitating use of non-greedy search strategy.

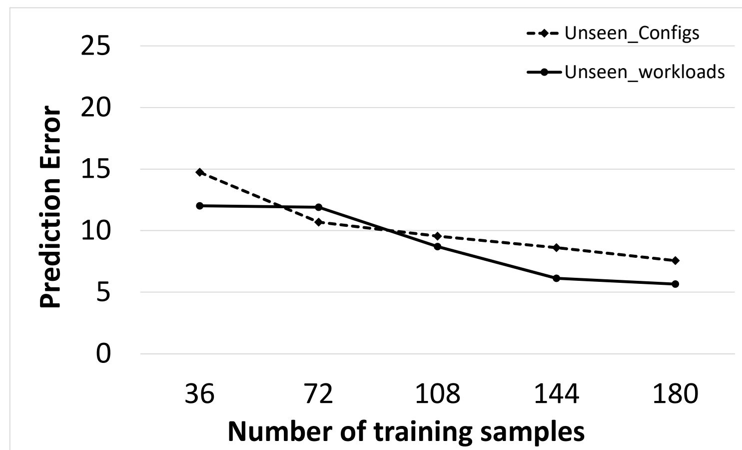


Figure 2.7. Prediction error for Cassandra using the surrogate performance model with Neural Network, as a function of the number of training samples

2.5.7 Performance Prediction

In this section, we create a surrogate model that can predict the performance of the NoSQL datastore for hitherto unseen workloads and unseen combinations of configuration parameters.

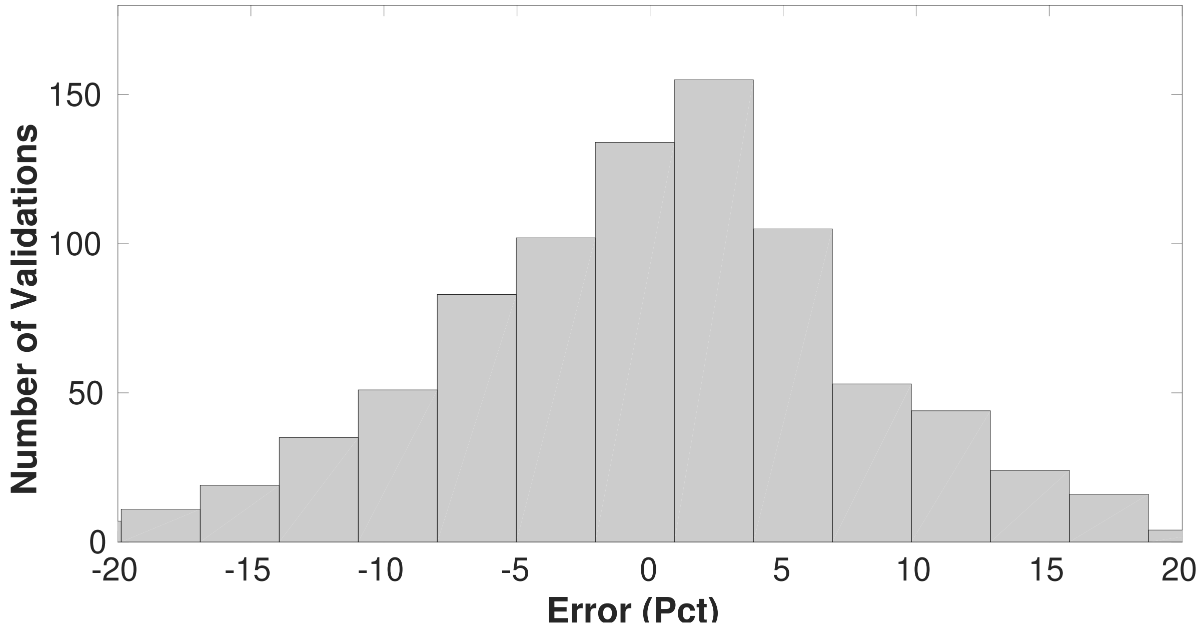


Figure 2.8. Cassandra: Distribution of performance prediction errors for unseen configurations. The average absolute error is 7.5% with most projections lying in the | 5 |% range.

Training the Prediction Model

Figure 2.7 shows the change in error for predicting performance under unseen workloads and configurations with respect to the number of training samples. We notice that training the model with more data samples enhances the performance but the performance improvement begins to level off at 180 collected training samples. The collected samples were sufficient for getting a prediction error of 7.5% for unseen configurations and 5.6% for unseen workloads using an ensemble of 20 neural nets, compared to 10.1% and 5.95% using a single net (Table 2.2). Going beyond 20 neural nets again gives diminishing improvements.

Validating Model Performance

We validate the prediction model separately for two dimensions: workloads and server configurations. To validate server configurations, we randomly withhold 25% of the configurations and predict on these. The network is then trained with the remaining 75% of the data, until convergence, and performance statistics are taken for the network (shown in

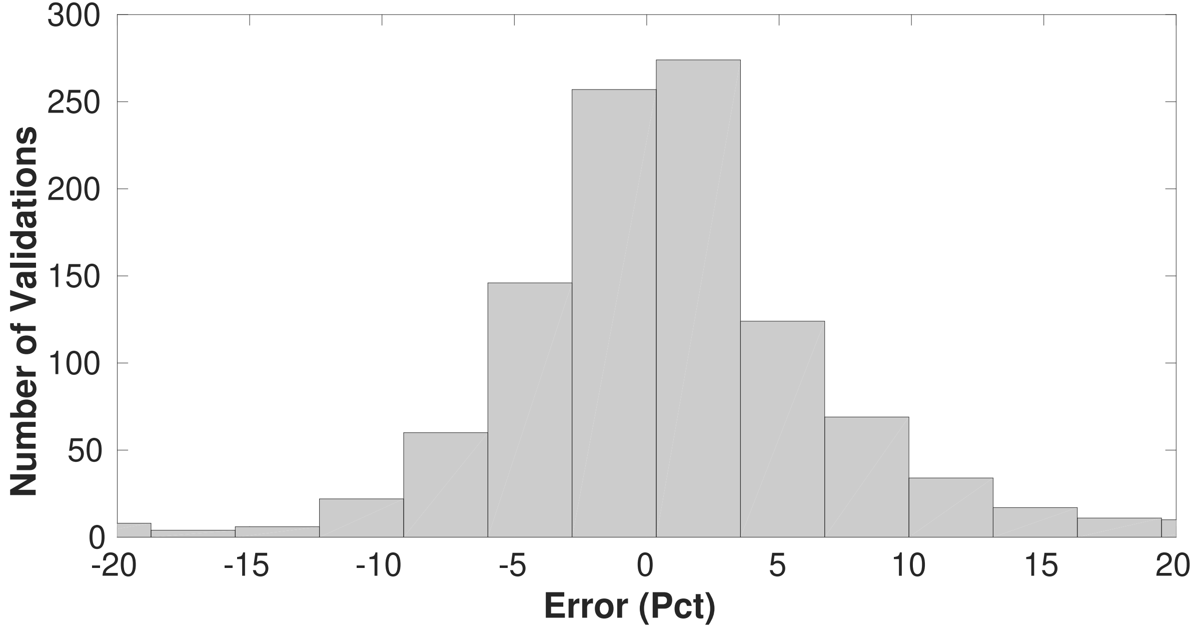


Figure 2.9. Cassandra: Distribution of performance prediction errors for unseen workloads. The average absolute error is 5.6% with most projections lying in the | 5 |% range

Table 2.2. Prediction Model Performance for Cassandra

	20 Nets		1 Net	
	Config.	Workload	Config.	Workload
Prediction Error	7.5%	5.6%	10.1%	5.95%
R^2 Value	0.74	0.75	0.51	0.73
Avg. RMSE	6859 op/s	6157 op/s	9338 op/s	6378 op/s

Table 2.2). We perform 10 randomized trials where we vary which 25% items we withhold and present the average results.

The same procedure was used to validate workload prediction performance, and Figures 2.8 and 2.9 show the histogram for the validation cases. The model maintains slightly better prediction accuracy across workloads, suggesting that the single feature that represents the workload's Read-proportion can capture the system dynamics well. The histogram also shows little bias, since the mean is close to zero, and this indicates that our DNN model is expressive enough for the task.

2.5.8 Performance Improvement through Optimal Configuration

For this experiment, Rafiki searches for better configuration settings for improving the performance of Cassandra. Since we make no assumptions about the relationships between configuration parameters (*e.g.*, their linearity), we utilize a Genetic Algorithm to optimize the set of configuration parameters. We train the network using all available 200 samples, unlike in the previous validation experiments (where we held out some samples). Thus, here while predicting, Rafiki has already seen that particular workload, but in all likelihood has not seen the optimal configuration parameter for that workload considering the large search space. We compare the performance of optimal configurations selected by Rafiki against the default performance, and and exhaustive search for three workloads (90% reads, 50% reads, and 10% reads). This exhaustive search is performed by testing 80 configuration sets for each workload. Figure 2.4 shows the performance gain when applying Rafiki's selected configuration compared to the default configuration for various workloads. On average, Rafiki shows 30% improvement over the default configuration across the range of workloads. We notice that higher gains are achieved for read-heavy workloads, with 41% improvement on average (range = 39-45%) for read-heavy workloads, *i.e.*, having read-proportion $\geq$ 70%. Lower gains are achieved with respect to write-heavy workloads, *i.e.*, with read-proportion $\leq$ 30%, average of 14% with a range of 6-24%. This reveals that Cassandra's default configuration is more suited for write-heavy workloads, while our metagenomics workload is read-heavy most of the time.

It should be noted that the Rafiki selected configuration comes within 15% with respect to the exhaustive grid search. However, the surrogate model is able to generate a new performance sample in 45 μs, thus allowing the GA to investigate approximately 3000 samples in the search space every 0.17 seconds, whereas around 14 days are needed to collect the equivalent number of samples with grid searching. The combined GA+surrogate takes 1.8 seconds to find the optimal configuration utilizing 3,350 surrogate evaluations on average. A single sample with grid searching takes around 2 minutes for loading data and another 5 minutes for collecting stable performance metrics experimentally. Thus, training the surrogate models and searching using GA is four orders of magnitude faster than exhaus-

Table 2.3. Cassandra: Performance improvement of optimal configuration selected by RAFIKI vs. Default configuration performance for single-server and two-server setups.

workload	RR=10%	RR=50%	RR=100%
Single Server Improve	15.2%	41.34%	48.35%
Two Servers Improve	3.2 %	67.37%	51.4 %

Table 2.4. ScyllaDB: Performance of RAFIKI selected configurations vs. Grid search

	WL1(R=70%)		WL2(R=100%)	
Opt. Technique	RAFIKI	Grid	RAFIKI	Grid
Avg Throughput	69,411	75,351	66,503	63,595
Gain over Default	12.29%	21.8%	9%	4.57%

tive grid search, suggesting that RAFIKI could be used to tune the compaction-related and concurrency-related parameters, and possibly others, at runtime as workload characteristics change.

2.5.9 Performance Improvement for Multiple Cassandra Instances

In this experiment, we measure the improvement in performance for multiple Cassandra instances using RAFIKI's selected configurations over the default configurations performance. For the two-servers experiment, we add one more shooter to utilize the increased performance of the created cluster. We also increase the replication factor by one, so that each instance stores an equivalent number of keys as the single-server case. From Table 2.3, we see similar improvements, on average, using the configuration set given by RAFIKI over the default configuration (34% for single-server case, and 40% for two-servers case). Again, the improvement due to RAFIKI increases with increasing RR, but it reaches an inflection point at RR=70% and then decreases a little.

2.5.10 ScyllaDB performance tuning

In this section, we investigate the effect of applying our performance tuning for ScyllaDB[6]. First, we wish to select the key performance tuning parameters. ScyllaDB provides a user-

[6]↑https://github.com/scylladb/scylla/blob/master/conf/scylla.yaml

transparent auto-tuning system internal to its operation, and this system makes parameter selection especially difficult because user settings for many configuration parameters are ignored by ScyllaDB, giving preference to its internal auto-tuning [27]. Consequently, even in an otherwise stationary system, without any change to the workload or to the configuration parameters, the throughput of ScyllaDB varies significantly. Figure 2.10 demonstrates this tuning-induced variance.

We find through experiments that the tuner acts like a hidden parameter—changing a parameter causes variance due to interdependence with the auto-tuning system that we cannot control. Due to these internal interactions, the ANOVA analysis yields significance to configuration settings that have high interaction with the auto-tuning system rather than sustained net-positive impacts on performance. For this reason, despite the ANOVA analysis, poor prediction performance will be obtained. For ScyllaDB, we rectify this by taking the ANOVA analysis for Cassandra, stripping out any parameters that are ignored by ScyllaDB, and adding in new parameters (sorted by variance) until 5 parameters are in the set, matching Cassandra in count. As shown in Table 2.4, using the selected parameters, Rafiki was able to achieve a performance gain of 12% for workload with 70% reads, and a gain of 9% for workload with 100% reads.

2.6 Conclusion

In this paper, we have highlighted the problem of tuning NoSQL datastores for dynamic workloads, as is typical for metagenomics workloads, seen in platforms such as MG-RAST. e application performance is particularly sensitive to changes to the workloads and datastore configuration parameters. We design and develop a middleware called Rafiki for automatically configuring the datastore parameters, using traces from the multi-tenant metagenomics system, MG-RAST. We demonstrate the power of Rafiki to achieve tuned performance in throughput for a leading NoSQL datastore, Cassandra, and a latest generation re-implementation of it, ScyllaDB, with the additional distinguishing feature that ScyllaDB has an auto-tuning feature. We apply ANOVA-based analysis to identify the key parameters that are the most impactiful to performance. We find, unsurprisingly, that the

relation between the identified configuration parameters and performance is non-monotonic and non-linear, while the parameter space is infinite with both continuous and integer control variables. We therefore create a DNN framework to predict the performance for unseen configurations and workloads. It achieves a performance prediction with an error in the range of 5-7% for Cassandra and 7-8% for ScyllaDB. Using a root-cause investigation indicates that ScyllaDB's native performance tends to fluctuate, sometime to a large degree (60% for 40 seconds), making prediction more challenging. We then create a Genetic Algorithm-based search process through the configuration parameter space, which improves the throughput for Cassandra by 41.4% for read-heavy workloads, and 30% on average. Further, to get an estimate of the upper bound of improvement, we compare to an exhaustive search process and see that , using 4 orders of magnitude lower searching time than exhaustive grid search, reaches within 15% and 9.5% of the theoretically best achievable performances for Cassandra and ScyllaDB, respectively. In future work, we are developing algorithms for the actual online reconfiguration process keeping the downtime to a minimum. We are also developing a prediction model for the workloads, which will allow us to develop an algorithm to cluster reads and writes for higher throughput.

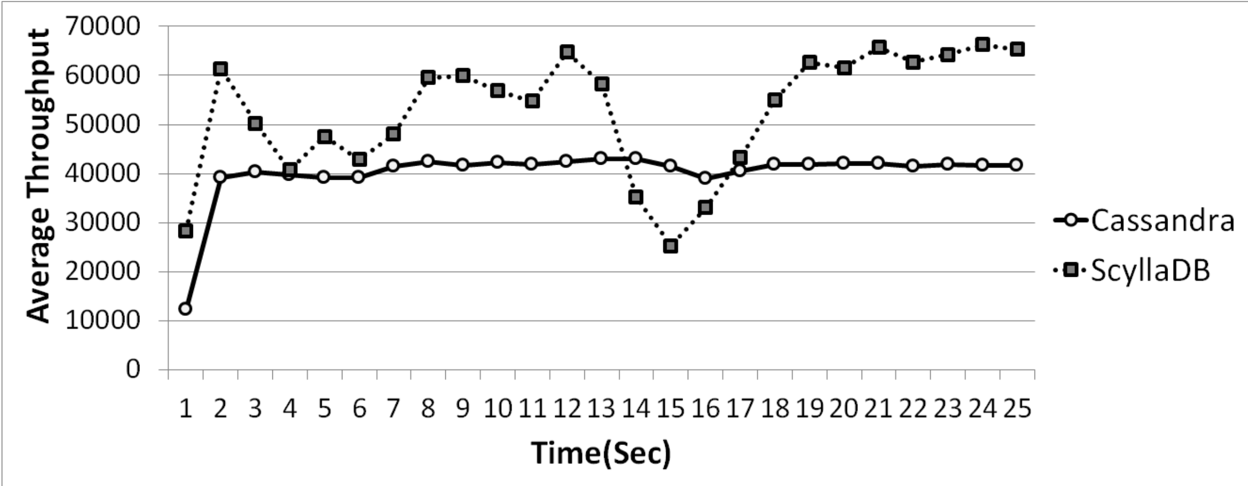


Figure 2.10. Average throughput for Cassandra and ScyllaDB under a 70% reads workload (collected every 10 seconds). Since Cassandra has a more stable performance compared to ScyllaDB, throughput prediction for Cassandra is more accurate.

3. SOPHIA: ONLINE RECONFIGURATION OF CLUSTERED NOSQL DATABASES FOR TIME-VARYING WORKLOADS

3.1 Abstract

Reconfiguring NoSQL databases under changing workload patterns is crucial for maximizing database throughput. This is challenging because of the large configuration parameter search space with complex interdependencies among the parameters. While state-of-the-art systems can automatically identify close-to-optimal configurations for static workloads, they suffer for dynamic workloads as they overlook three fundamental challenges: (1) Estimating performance degradation during the reconfiguration process (such as due to database restart). (2) Predicting how transient the new workload pattern will be. (3) Respecting the application's availability requirements during reconfiguration. Our solution, Sophia, addresses all these shortcomings using an optimization technique that combines workload prediction with a cost-benefit analyzer. Sophia computes the relative cost and benefit of each reconfiguration step, and determines an optimal reconfiguration for a future time window. This plan specifies when to change configurations and to what, to achieve the best performance without degrading data availability. We demonstrate its effectiveness for three different workloads: a multi-tenant, global-scale metagenomics repository (*MG-RAST*), a bus-tracking application (*Tiramisu*), and an HPC data-analytics system, all with varying levels of workload complexity and demonstrating dynamic workload changes. We compare Sophia's performance in throughput and tail-latency over various baselines for two popular NoSQL databases, Cassandra and Redis.

3.2 Introduction

Automatically tuning database management systems (DBMS) is challenging due to their plethora of performance-related parameters and the complex interdependencies among subsets of these parameters [10], [11], [28]. For example, Cassandra has 56 performance tuning parameters and Redis has 46 such parameters. Several prior works like Rafiki [28], OtterTune [10], BestConfig [29], and others [11], [30], [31], have solved the problem of optimizing

a DBMS when workload characteristics relevant to the data operations are relatively static. We call these "*static configuration tuners*". However, these solutions cannot decide on a set of configurations over a window of time in which the workloads are changing, i.e., what configuration to change to and when. Further, existing solutions cannot perform the reconfiguration of a cluster of database instances without degrading data availability.

Workload changes lead to new optimal configurations. However, it is not always desirable to switch to new configurations because the new workload pattern may be short-lived. Each reconfiguration action in clustered databases incurs costs because the server instance often needs to be restarted for the new configuration to take effect, causing a transient hit to performance during the reconfiguration period. In the case of dynamic workloads, the new workload may not last long enough for the reconfiguration cost to be recouped over a time window of interest to the system owner. Therefore, a proactive technique is required to estimate when executing a reconfiguration is going to be globally beneficial.

Fundamentally, this is where the drawback of all prior approaches to automatic performance tuning of DBMS lies—in the face of dynamic changes to the workload, they are either silent on when to reconfigure or perform a naïve reconfiguration whenever the workload changes. We show that a naïve reconfiguration, which is oblivious to the reconfiguration cost, actually *degrades* the performance for dynamic workloads relative to the default configurations and also relative to the best static configuration achieved using a static tuner with historical data from the system (Figure 3.3). For example, during periods of high dynamism in the read-write switches in a metagenomics workload in the largest metagenomics portal called MG-RAST [32], naïve reconfiguration degrades throughput by a substantial 61.8% over default.

Our Solution: We develop an online reconfiguration system—SOPHIA—for a NoSQL cluster comprising of multiple server instances, which is applicable to dynamic workloads with various rates of workload shifts. SOPHIA uses historical traces of the workload to train a workload predictor, which is used at runtime to predict future workload patterns. Workload prediction is a challenging problem and has been studied in many prior works [33]–[35]. However, the workload predictor itself is not a contribution of SOPHIA, and it can operate with any workload predictor with sufficiently accurate and long-horizon predictions.

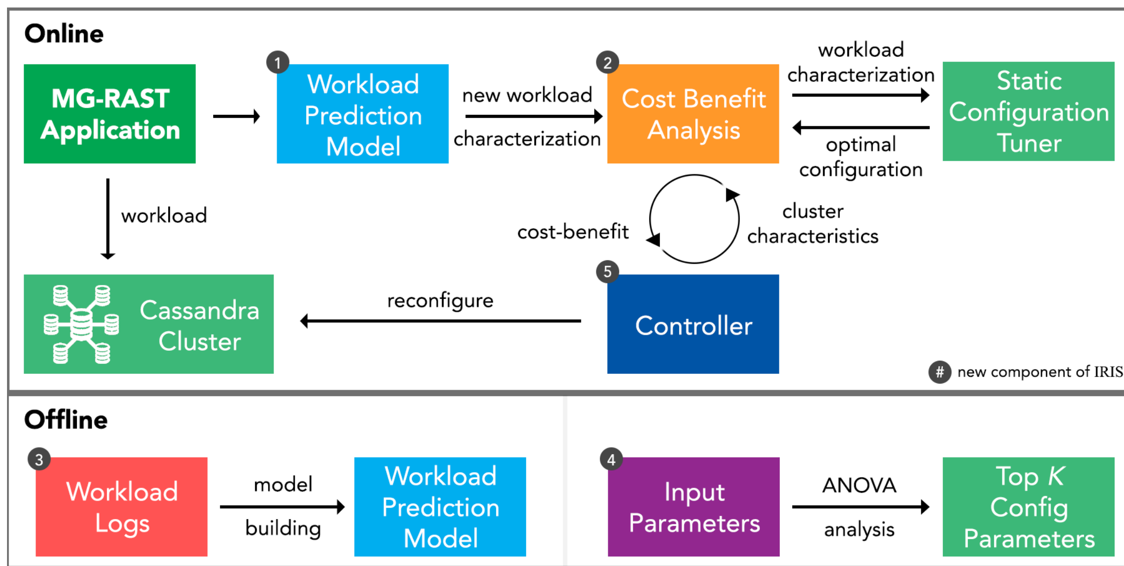


Figure 3.1. Workflow of Sophia with offline model building and the online operation, plus the new components of our system. It also shows the interactions with the NoSQL cluster and a static configuration tuner, which comes from prior work.

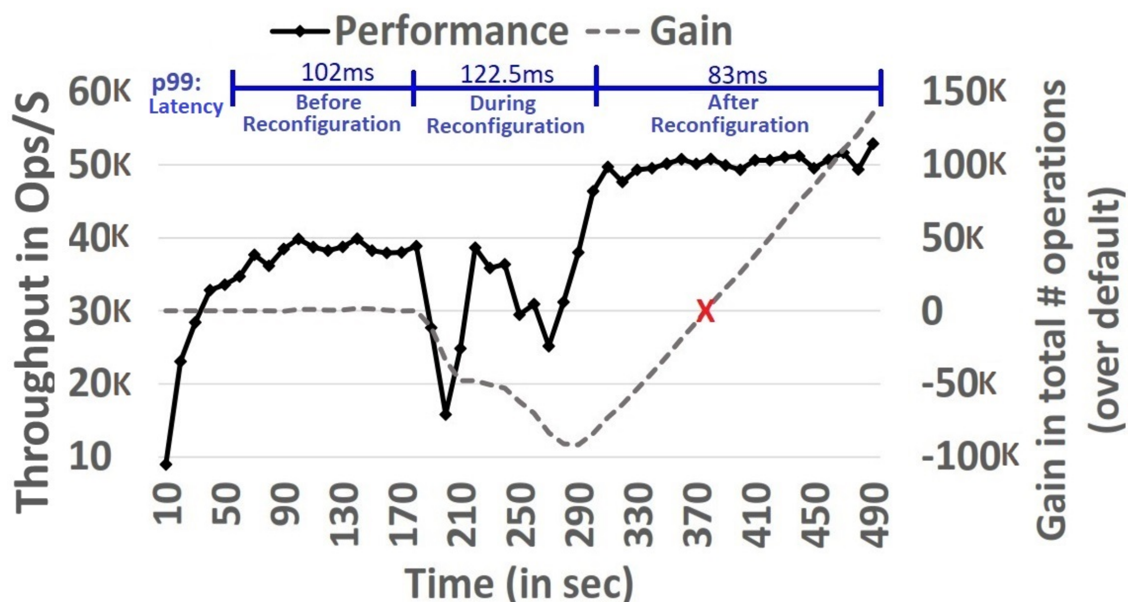


Figure 3.2. The effect of reconfiguration on the performance of the system. Sophia uses the workload duration information to estimate the cost and benefit of each reconfiguration step and generates plans that are globally beneficial.

Sophia searches the vast space of all possible reconfiguration plans, and hence determines the best plan through a novel Cost-Benefit-Analysis (CBA) scheme. For each shift in the predicted workload trace, Sophia interacts with any existing static configuration tuner (we use Rafiki in our work because it is already engineered for NoSQL databases and is publicly available [36]), to quickly provide the optimal *point configurations* for the new workload and the estimated benefit from this new configuration. Sophia performs the CBA analysis, taking into account the predicted duration of the new workload and the estimated benefit from each reconfiguration step. Finally, for each reconfiguration step in the selected plan, Sophia initiates a distributed protocol to reconfigure the cluster without degrading data availability and maintaining the required data consistency requirement.

During its reconfiguration, Sophia can deal with different replication factors (RF) and consistency level (CL) requirements specified by the user. It ensures that the data remains continuously available through the reconfiguration process, with the required CL. This is done by controlling the number of server instances that are concurrently reconfigured. However, this is only possible when RF>CL. In cases where RF=CL, reconfiguring any node in the cluster will degrade data availability as every request will require a response from every replica before it is returned to the user. Therefore, we also implement an aggressive variant of our system (Sophia-aggressive), which relaxes the data availability requirement in exchange for faster reconfiguration and hence better performance.

Evaluation Cases

We evaluate Sophia on two NoSQL databases, Cassandra [37] and Redis [38]. The first use case is based on real workload traces from the metagenomics analysis pipeline, MG-RAST [39], [40]. It is a global-scale metagenomics portal, the largest of its kind, which allows users to simultaneously upload their metagenomic data and use its computational pipeline.

The workload does not have any discernible daily or weekly pattern, as the requests come from all across the globe and we find that the workload can change drastically over a few minutes. This presents a challenging use case as only 5 minutes or less of accurate lookahead is possible. The second use case is a bus-tracking application where read, scan, insert, and update operations are submitted to a backend database. The data has strong daily and

weekly patterns to it. The third use case is a queue of data analytics jobs such as would be submitted to an HPC computing cluster. Here the workload can be predicted over long time horizons (order of an hour) by observing the jobs in the queue and leveraging the fact that a significant fraction of the job patterns are recurring. Thus, our three cases span the range of patterns and corresponding predictability of the workloads.

We compare our approach to existing solutions and show that SOPHIA increases throughput (and decreases tail-latency) under all dynamic workload patterns and for all types of queries, with no downtime. For example, SOPHIA achieves 24.5% higher throughput over default configurations and 21.5% higher throughput over a statically determined idealized optimal configuration in the bus-tracking workload. SOPHIA achieves 38% and 30% higher aggregate throughput over these two baselines in the HPC cluster workload. With SOPHIA's auto-tuning capability, Redis is able to operate through the entire range of workload sizes and read/write intensities, while the *vanilla Redis fails with large workloads.* The main contributions of SOPHIA are:

1. We show that state-of-the-art static tuners when applied to dynamic workloads degrade throughput below the state-of-practice of using the default parameter values and also degrade data availability. For a clustered database like in our target environment, a naïve reconfiguration upon any workload change actually degrades performance below the default static configuration.

2. SOPHIA performs cost-benefit analysis to achieve *long-horizon optimized performance for clustered NoSQL instances* in the face of dynamic workload changes, including unpredictable and fast changes to the workload.

3. SOPHIA executes a decentralized protocol to gracefully switch over the cluster to the new configuration while respecting the data consistency guarantees and keeping data continuously available to users.

3.3 Design of Sophia

Sophia seeks to answer the following two broad questions: When should the cluster be reconfigured? How should we apply the reconfiguration steps?

The answer to the first question leads to what we call a *reconfiguration plan*. The answer to the second question is given by our distributed protocol that reconfigures the various server instances in rounds. Next, we describe Sophia's components.

3.3.1 Workload Modeling and Forecasting:

In a generic sense, we can define the workload at a particular point in time as a vector of various time-varying features:

$$W(t) = \{p_1(t), p_2(t), p_3(t), ..., p_n(t)\} \tag{3.1}$$

where the workload at time t is $W(t)$ and $p_i(t)$ is the time-varying i-th feature. These features may be directly measured from the database, such as the load (i.e., requests/s) and the occupancy level of the database, or they may come from the computing environment, such as the number of users or jobs in a batch queue. These features are application dependent and are identified by analyzing the application's historical traces. Details for time-varying features of each application are described in Section 5.5.

For workload forecasting, we discretized time into sliced T_d durations (= $30s$ in our model) to bound the memory and compute cost. We then predicted future workloads as:

$$W(t_{k+1}) = f_{\text{pred}}(W(t_k), W(t_{k-1}), ..., W(t_0)) \tag{3.2}$$

where k is the current time index into T_d-wide steps. For ease of exposition for the rest of the paper, we drop the term T_d, assuming implicitly that this is one time unit.

The function f_{pred} is any function that can make such a prediction, and in Sophia, we utilize a simple Markov-Chain model for MG-RAST and Bus-Tracking, while we use

a deterministic, fully accurate output from a batch scheduler for the HPC data analytics workload, i.e., a perfect f_{pred}.

However, more sophisticated estimators, such as neural networks [33], [41], [42], even with some degree of interpretability [43], have been used in other contexts and SOPHIA is modular enough to use any such predictor.

3.3.2 Adapting a Static Configuration Tuner for SOPHIA:

SOPHIA uses a static configuration tuner (RAFIKI), designed to work with Cassandra, to output the best configuration for the workload at any given point in time. RAFIKI uses Analysis-of-variance (ANOVA) [44] in order to estimate the importance of each parameter. It selects the top-k parameters in its configuration optimization method, which is in turn determined by a significant drop-off in the importance score. The ability to adapt optimized "kernels" to build robust algorithms comes from our vision to accelerate the pipeline of creating efficient algorithms, conceptualized in Sarvavid [45].

The 7 highest performance-sensitive parameters for all three of our workloads are:

(1) Compaction method, (2) # Memtable flush writers, (3) Memory-table clean-up threshold, (4) Trickle fsync, (5) Row cache size, (6) Number of concurrent writers, and (7) Memory heap space. These parameters vary with respect to the reconfiguration cost that they entail. The change to the compaction method incurs the highest cost as it causes every Cassandra instance to read all its SSTables and re-write them to the disk in the new format. However, due to inter-dependability between these parameters, the compaction frequency is still being controlled by reconfiguring the second and third parameters with the cost of a server restart. Similarly, parameters 4, 6, 7 need a server restart for their new values to take effect and these cause the next highest level of cost. Finally, some parameters (parameter 5 in our set) can be reconfigured without needing a server restart and therefore have the least level of cost.

In general, the database system has a set of performance-impactful configuration parameters $C = \{c_1, c_2, \cdots, c_n\}$ and the optimal configuration C_{opt} depends on the particular workload $W(t)$ executing at that point in time.

In order to optimize performance across time, SOPHIA needs the static tuner to provide an estimate of throughput for both the optimal and the current configuration for any workload:

$$H_{\text{sys}} = f_{\text{ops}}(W(t), C_{\text{sys}}) \tag{3.3}$$

where H_{sys} is the throughput of the cluster of servers with a configuration C_{sys} and $f_{\text{ops}}(W(t), C_{\text{sys}})$ provides the system-level throughput estimate. C_{sys} has $N_s \times$ `\bm{C}` dimensions for N_s servers and C different configurations.

Cassandra by careful design achieves efficient load balancing across multiple instances whereby each contributes approximately equally to the overall system throughput [37], [46]. Thus, we define a single server average performance as $H_{\text{i}} = \frac{H_{\text{sys}}}{N_s}$.

From these models of throughput, optimal configurations can be selected for a given workload:

$$C_{\text{opt}}(W(t)) = \arg\max_{C_{\text{sys}}} H_{\text{sys}} = \arg\max_{C_{\text{sys}}} f_{\text{ops}}(W(t), C_{\text{sys}}) \tag{3.4}$$

In general, C_{opt} can be unique for each server, but in SOPHIA, it is the same across all servers and thus has a dimension of `\bm{C}` making the problem computationally easier. This is due to the fact that SOPHIA makes a design simplification—it performs the reconfiguration of the cluster as an atomic operation. Thus, it does not abort a reconfiguration action mid-stream and all servers must be reconfigured in round i prior to beginning any reconfiguration of round $\text{i} + 1$. We also speed up the prediction system f_{ops} by constructing a cached version with the optimal configuration C_{opt} for a subset of W and using nearest-neighbor lookups whenever a near enough neighbor is available.

3.3.3 Dynamic Configuration Optimization:

SOPHIA's core goal is to maximize the total throughput for a database system when faced with dynamic workloads. This introduces time-domain components into the optimal configuration strategy $C_{\text{opt}}^{T} = C_{\text{opt}}(W(t))$, for all points in (discretized) time till a lookahead T_L.

Here, we describe the mechanism that SOPHIA uses for CBA modeling to construct the best reconfiguration plan (defined formally in Eq. 4.1) for evolving workloads.

In general, finding solutions for C_{opt}^{T} can become impractical since the possible parameter space for C is large and the search space increases linearly with T_L.

To estimate the size of the configuration space, consider that in our experiments we assumed a lookahead $T_L = 30$ minutes and used 7 different parameters, some of which are continuous, e.g., `Memory-table clean-up threshold`. If we take an underestimate of each parameter being binary, then the size of the search space becomes $2^{7\times 30} = 1.6 \times 10^{63}$ points, an impossibly large number for exhaustive search. We define a compact representation of the reconfiguration points (Δ's) to easily represent the configuration changes.

The maximum number of switches within T_L, say M, is bounded since each switch takes a finite amount of time. The search space for the dynamic configuration optimization is then $Combination(TL, M), M) \times$ `\bm{C}`.

This comes from the fact that we have to choose at most M points to switch out of all the T_L time points and at each point there are `\bm{C}` possible configurations.

We define the reconfiguration plan as:

$$C_{\text{sys}}^{\Delta} = [T = \{t_1, t_2, ..., t_M\}, C = \{C_1, C_2, ..., C_M\}] \tag{3.5}$$

where t_k is a point in time and C_k is the Thus, the reconfiguration plan gives *when* to perform a reconfiguration and at each such point, *what* configuration to choose.

The objective for SOPHIA is to select the best reconfiguration plan $(C_{\text{sys}}^{\Delta})^{\text{opt}}$ for the period of optimization, lookahead time T_L:

$$(C^{\Delta}_{\text{sys}})^{\text{opt}} = \arg\max_{C^{\Delta}_{\text{sys}}} B(C^{\Delta}_{\text{sys}}, W) - L(C^{\Delta}_{\text{sys}}, W) \tag{3.6}$$

where C^{Δ}_{sys} is the reconfiguration plan, B is the benefit function, and L is the cost (or loss) function, and W is the time-varying workload description. When SOPHIA will extend to allow scale out, we will have to consider the data movement volume as another cost to minimize. The L function captures the opportunity cost of having each of N_s servers offline for T_r seconds for the new workload versus if the servers remained online with the old configuration. As the node downtime due to reconfiguration never exceeds Cassandra's threshold for declaring a node is dead (3 hours by default), data-placement tokens are not re-assigned due to reconfiguration. Therefore, we do not include cost of data movement in functions L. SOPHIA can work with any reconfiguration cost, including different costs for different parameters—these can be fed into the loss function L.

The objective is to maximize the time-integrated gain (benefit – cost) of the reconfiguration from Eq. (3.6). The three unknowns in the optimal plan are M, T, and C, from Eq. (4.1). If only R servers can be reconfigured at a time (explained in Sec. 4.5 how R is calculated), at least $T_r \times \frac{N_s}{R}$ time must elapse between two reconfigurations. This puts a limit on M, the maximum number of reconfigurations that can occur in the lookahead period T_L.

A greedy solution for Eq. (3.6) that picks the first configuration change with a net-increase in benefit may produce suboptimal C^{Δ}_{sys} over the horizon T_L because it does not consider the coupling between multiple successive workloads. For example, considering a pairwise sequence of workloads, the best configuration may not be optimal for either $W(t_1)$ or $W(t_2)$ but *is* optimal for the paired sequence of the two workloads. This could happen if the same configuration gives reasonable performance for $W(t_1)$ or $W(t_2)$ and has the advantage that it does not have to switch during this sequence of workloads. This argument can be naturally extended to longer sequences of workloads.

A T_L value that is too long will cause SOPHIA to include decision points with high prediction errors, and a value that is too short will cause SOPHIA to make almost greedy decisions. The appropriate lookahead period is selected by benchmarking the non-monotonic but convex throughput while varying the lookahead duration and selecting the point with maximum end-to-end throughput.

We give our choices for our three applications when describing the first experiment with each application (Section 5.6).

3.3.4 Finding Optimal Reconfiguration Plan with Genetic Algorithms:

We use a heuristic search technique, Genetic Algorithms or GA, to find the optimal reconfiguration plan. Although meta-heuristics like GA do not guarantee finding global optima, they have two desirable properties for SOPHIA. Our space is non-convex because many of the parameters impact the same resources such as CPU, RAM, and disk, and settings of one parameter impact the others. Therefore, greedy or gradient descent-based searches are prone to converge to a local optima. Also the GA's tunable completion is needed in our case for speedy decisions, as the optimizer executes in the critical path.
The representation of the GA solution incorporates two parts. First, the chromosome orientation, which is simply the reconfiguration plan (Eq. 4.1).

The second part is the fitness function definition used to assess the quality of different reconfiguration plans. For this, we use the cost-benefit analysis as shown in Eq. 3.6 where fitness is the total number of operations (normalized for bus-tracking traces to account for different operations' scales) for the T_L window for the tested reconfiguration plan and given workload. We build a simulator to apply the individual solutions and to collect the corresponding fitness values, which are used to select the best solutions and to generate new solutions in the next generation. We utilize a Python library, `pyeasyga`, with 0.8 crossover fraction and population size of 200. We run 10 concurrent searches and pick the best configuration from those. The runtime of this algorithm is dependent on the length of the lookahead period and the number of decision points. For MG-RAST, the GA has 30 decision points in the lookahead period and results in execution time of 30-40 sec. For the HPC workload, the number of decision points is 180 as it has a longer lookahead period, resulting in a runtime of 60-70 sec. For the bus-tracking workload, the GA has 48 decision points and a runtime of 20-25 sec. The GA is typically re-run toward the end of the lookahead period to generate the reconfiguration plan for the next lookahead time window. Also, if the actual workload is

different from the predicted workload, the GA is re-invoked. This last case is rate limited to prevent too frequent invocations of the GA during (transient) periods of non-deterministic behavior of the workload.

3.3.5 Distributed Protocol for Online Reconfiguration:

Cassandra and other distributed databases maintain high availability through configurable redundancy parameters, consistency level (CL) and replication factor (RF). CL controls how many confirmations are necessary for an operation to be considered successful.

RF controls how many replicas of a record exist throughout the cluster. Thus, a natural constraint for each record is RF $\geq$ CL. SOPHIA queries token assignment information (where a token represents a range of hash values of the primary keys which the node is responsible for) from the cluster, using tools that ship with all popular NoSQL distributions (like `nodetool ring` for Cassandra), and hence constructs what we call a *minimum availability subset* (N_{minCL} for short). We define this subset as the minimum subset of nodes that covers at least CL replicas of *all* keys. To meet CL requirements, SOPHIA insures that N_{minCL} nodes are operational at any point of time. Therefore,

SOPHIA makes the design decision to configure up to $R = N_s - N_{minCL}$ servers at a time, where N_{minCL} depends on RF, CL, and token assignment. If we assume a single token per node (Cassandra's default with vnodes disabled), then a subset of $\lceil \frac{N_s}{RF} \rceil$ nodes covers all keys at least once. Consequently, N_{minCL} becomes CL $\times \lceil \frac{N_s}{RF} \rceil$ to cover all keys at least CL times. Thus, the number of reconfiguration steps $= \frac{N_s}{R} = \frac{RF}{RF-CL}$ becomes independent of the cluster size N_s.

In the case where RF = CL, N_{minCL} becomes equivalent to N_s and hence SOPHIA cannot reconfigure the system, without harming data availability, hence we use the SOPHIA-aggressive variant in that case. However, we expect most systems with high consistency requirements to follow a read/write quorum with CL $= \lceil \frac{\text{RF}}{2} \rceil$ [47].

Note that SOPHIA reduces the number of available data replicas during the transient reconfiguration periods, and hence reduces the system's resilience to additional failures.

However, one optional parameter in SOPHIA is how many failures during reconfiguration the user will want to tolerate (our experiments were run with zero). This is a high-level parameter that is intuitive to set by the database admin. Also notice that data that existed on the offline servers prior to reconfiguration is not lost due to the drain step, but data written during the transient phase has lower redundancy until the reconfigured servers get back online. In order to reconfigure a Cassandra cluster, SOPHIA performs the following steps, R server instances at a time:

1. **Drain**: Before shutting down a Cassandra instance, we flush the entire Memtable to disk by using Cassandra's tool `nodetool drain` and this ensures that there are no pending commit logs to replay upon a restart.
2. **Shutdown**: The Cassandra process is killed on the node.
3. **Configuration file**: Replace the configuration file with new values for all parameters that need changing.
4. **Restart**: Restart the Cassandra process on the same node.
5. **Sync**: SOPHIA waits for Cassandra's instance to completely rejoin the cluster by letting a coordinator know of where to locate the node and then synchronizing the missed updates during the node's downtime.

In Cassandra, writes for down nodes are cached by available nodes for some period and re-sent to the nodes when they rejoin the cluster. The time that it takes to complete all these steps for one server is denoted by T_r, and T_R for the whole cluster, where $T_R = T_r \times \frac{RF}{RF-CL}$. During all steps 1-5, additional load is placed on the non-reconfiguring servers as they must handle the additional write and read traffic. Step 5 is the most expensive and typically takes 60-70% of the total reconfiguration time, depending on the amount of cached writes.

We minimize step 4 practically by installing binaries from the RAM and relying on draining rather than commit-log replaying in step 1, reducing pressure on the disk.

3.3.6 DBA level of Intervention:

Although Sophia's design aims at automating the whole reconfiguration management process, a few steps are still needed to be performed by the DBA. The following list represent the main items the DBA needs to provide Sophia with:

1. Historical traces of the target application's workload: These traces are used by Sophia to identify the time-varying features of the workload, and hence use these features to train the workload prediction model.

2. A list of commands which Sophia uses to: (1) query the status of a given node (alive, down, sync, ... etc.). (2) query the token assignment and replication information. (3) query the topology of the target cluster.

3. Administrator credentials to kill, start or restart the DBMS on any node in the cluster.

3.4 Datasets

MG-RAST Workload: We use real workload traces from MG-RAST, the leading metagenomics portal operated by the US Department of Energy. As the amount of data stored by MG-RAST has increased beyond the limits of traditional SQL stores (23 PB as of August 2018), it relies on a distributed NoSQL Cassandra database cluster. Users of MG-RAST are allowed to upload "jobs" to its pipeline, with metadata to annotate job descriptions. All jobs are submitted to a computational queue of the US Department of Energy private Magellan cloud. We analyzed days of query trace from the MG-RAST system from till . From this data, we make several observations: (i) Workloads' read ratio (RR) switches rapidly with over 26,000 switches in the analyzed period. (ii) A negative correlation of -0.72 is observed between the Workloads' read ratio and number of requests/s (i.e., load). That is due to the fact that most of the write operations are batched to improve network utilization. (iii) Majority (i.e., more than 80%) of the switches are abrupt, from RR=0 to RR=1 or vice versa. (iv) KRD (key reuse distance) is very large. (v) No daily or weekly workload pattern is discernible, as expected for a globally used cyberinfrastructure.

Bus Tracking Application Workload: Secondly, we use real workload traces from a bus-tracking mobile application called **Tiramisu** [33]. The system provides live tracking of the public transit bus system. It updates bus locations periodically and allows users to search for nearby bus stops. There are four types of queries—read, update, insert, and scan (retrieving all the records in the database that satisfy a given predicate, which is much heavier than the other operations). A sample of the traces is publicly available [48]. We trained our model using 40 days of query traces, while 18 days were used as testing data. We draw several observations from this data: (i) The traces show a daily pattern of workload switches. For example, the workload switches to scan-heavy in the night and switches to update-heavy in the early morning. (ii) The Workload is a mixture of Update, Scan, Insert, and Read operations in the ratio of 42.2%, 54.8%, 2.82%, and 0.18% respectively. (iii) KRD is very small. From these observations, we notice that the workload is very distinct from MG-RAST and thus provides a suitable point for comparison.

Simulated Analytics Workload: For long-horizon reconfiguration plans, we simulate synthetic workloads representative of batch data analytics jobs, submitted to a shared HPC queue. We integrate SOPHIA with a job scheduler (like PBS [49]), that examines jobs while they wait in a queue prior to execution. Thus, the scheduler can profile the jobs waiting in the queue, and hence forecast the aggregate workload over a lookahead horizon, which is equal to the length of the queue. We model the jobs on data analytics jobs submitted to a Microsoft Cosmos cluster [50] and as in that paper, we achieve high accuracy in predicting when a job will start executing. Thus, SOPHIA is able to drive long-horizon reconfiguration plans. Each job is divided into phases: a write-heavy phase resembling an upload phase of new data, a read-heavy phase resembling executing analytical queries to the cluster, and a third, write-heavy phase akin to committing the analysis results. However, some jobs can be recurring (as shown in [50], [51]) and running against already uploaded data. These jobs will execute the analysis phase directly, skipping the first phase. The size of each phase is a random variable with U(200,100K) operations, and whenever a job finishes, a new job is selected from the queue and executed. We vary the level of concurrency and have an equal mix of the two types of jobs and monitor the aggregate workload. With increase in

concurrency, the aggregate pattern becomes smoother and the latency of individual jobs increases.

3.5 Experimental Results

Here we evaluate the performance of SOPHIA under different experimental conditions for the 3 applications. We use a simple query model typical for NoSQL databases and is in contrast to complex analytics queries supported by more complex database engines. Hence, our throughput is defined as the number of queries per second. In all experiments, we collect both throughput and tail latency (p99) performance metrics. However, since the two parameters have an almost perfect inverse relationship in all experiments, we omit tail latency (except in Figures 3.4 and 3.6).

We evaluate SOPHIA on Amazon EC2 using instances of size M4.xlarge with 4 vCPU's, 16 GB of RAM, provisioned IOPS (SSD) EBS for storage and network bandwidth of 0.74 Gbits/s for all Cassandra servers and workload drivers. Each node is loaded with 6 GB of data initially (SOPHIA's performance is evaluated with greater data volumes in Experiment 4).

We use multiple concurrent clients to saturate the database servers and aggregate the throughput and tail latency observed by every client.

Baseline Comparisons

We compare the performance of SOPHIA to baseline configurations (1-5). We consider 3 variants of SOPHIA (6-8).

(1) Default:

The default configuration that ships with Cassandra. This configuration favors write-heavy workloads by design [52].

(2) Static Optimized: This baseline resembles the static tuner () when queried to provide the one *constant* configuration that optimizes for the entire future workload. This is an impractically ideal solution since it is assumed here that the future workload is known

perfectly. However, non-ideally no configuration changes are allowed dynamically.

(3) Naïve Reconfiguration: Here, when the workload changes, 's provided reconfiguration is always applied, instantiated by concurrently shutting down all server instances, changing their configuration parameters, and restarting all of them. Practically, this makes data unavailable and may not be tolerable in many deployments such as user-facing applications. The static configuration tuners are silent about when the optimal configurations determined by them must be applied and this baseline is a logical instantiation of all of the prior work.

(4) ScyllaDB: The performance of NoSQL database ScyllaDB [53] in its vanilla form. ScyllaDB is touted to be a much faster (10X or higher) drop-in replacement to Cassandra [54]. This stands in for other self-tuning databases [55].

(5) Theoretical Best: This baseline resembles the theoretically best achievable performance over the predicted workload period. This is simulated by assuming Cassandra is running with the optimal configuration at any point of time and not penalizing it for the cost of reconfiguration. This serves as an upper bound for the performance.

(6) SOPHIA with Oracle: This is SOPHIA with a fully accurate workload predictor.

(7) SOPHIA-aggressive: A variant from SOPHIA that prefers faster reconfiguration over data availability and is used only when RF=CL. SOPHIA-aggressive violates the availability requirement by reconfiguring all servers at the same time. Unlike Naïve, it uses the CBA model to decide when to reconfigure, and therefore it does not execute reconfiguration every time the workload changes.

(8) SOPHIA: This is our complete system.

Major Insights

We draw some key insights from the experimental results. First, globally shared infrastructures with black-box jobs only allow for short-horizon workload predictions. This causes SOPHIA to take single-step reconfiguration plans and limits its benefit over a static optimized

approach (Figure 3.3). In contrast, when job characteristics can be predicted well (bus tracking and data analytics applications), SOPHIA achieves significant benefit over both default and static optimized cases (Figures 3.4 and 3.5). This benefit stays even when there is significant uncertainty in predicting the exact job characteristics as shown in Figure 3.9.

Second, Cassandra can be used in preference to the recent popular drop-in ScyllaDB, an auto-tuning database, with higher throughput across the entire range of workload types, as long as we overlay a dynamic tuner, such as SOPHIA, atop Cassandra (Figures 3.3 and 3.5). Third, as the replication factor increases while the number of server are fixed, the reconfiguration time of SOPHIA decreases, thus improving its benefit (Figure 3.7). Contrarily, as CL increases, the benefit of SOPHIA shrinks (Figure 3.7). Finally, SOPHIA is applied to a different NoSQL database, Redis, and solves a long-standing configuration problem with it, one which has caused Redis to narrow its scope to being an in-memory database only (Figure 3.10).

Experiment 1: MG-RAST Workload

We present our experimental evaluation with test days of MG-RAST data. To zoom into the effect of SOPHIA with different levels of dynamism in the workload, we segment the workload into 4 scenarios and present those results in addition to the aggregated ones.

Workload Prediction Model: We created 16,512 training samples composed of $T_d = 5min$ steps across the days MG-RAST workloads.

We compare the performance of a first-order and a second-order Markov Chain model.

We represent the states as the proportion of read operations during the T_d interval. We use a quantization level of 10% in the read ratio between different states. We categorize the test days into 4: "Slow", "Medium", and "Fast", by the frequency of switching from the read- to the write-intensive workloads and this maps to the average read ratios (RR) shown in Table 3.1. "Write" represents days with long write-heavy periods. Table 3.1 shows the prediction RMSE for the four representative workload scenarios. Because of the lack of application-level knowledge, in addition to the well-known uncertainty in job execution times in genomics pipelines [56], the Markov Chain model only provides accurate predic-

tions for short time intervals. Moreover, increasing the order of the model has very little impact on the prediction performance and also increases the number of states (11 states in the First-order model vs. 120 states in the Second-order model). We also tried to train a more complex model (RNN) but its prediction quality was similar. We notice that the best accuracy is for the "Slow" scenario, whereas it drops below 50% for "Medium", and it is always below 50% for the "Fast" and "Write" scenarios. Because the "Slow" scenario is the most common (observed 74% of time in the training data), we use a value of $T_L = 5min$.

Table 3.1. RMSE for predicting MG-RAST and Bus-Tracking workloads.

MG-RAST					
MC-Order	First		Second		RR
Frequency	5m	10m	5m	10m	-
Slow	34.4%	56%	34%	55%	70%
Medium	59%	90%	59%	89%	50%
Fast	66%	93%	63%	89%	45%
Write	52.8%	76.1%	51.5%	75.5%	35%
Aggregate	43.7%	68.7%	43.4%	68.2%	-

Bus-Tracking		
Lookahead	First	Second
15m	6.9%	7.12%
1h	7.4%	7.4%
2h	7.9%	7.4%
5h	10%	7.5%
10h	13.7%	8%
#States	117	647

Performance Comparison:

Now we show the performance of Sophia with respect to the four workload categories. We first present the result with the smallest possible number of server instances, 4, run with operational MG-RAST's parameters RF=3 and CL=1 [57].

We show the result in terms of total operations for each test workload as well as a weighted average "combined" representation that models behavior for the entire MG-RAST workload. Figure 3.3 shows the performance improvements for our test cases.

From Figure 3.3, we see that Sophia always outperforms naïve in total ops/s (average of 31.4%) and individually in read (31.1%) and write (33.5%) ops/s.

Sophia also outperforms the default for the slow and the mid frequency cases, while it slightly under performs in the fast frequency case. The average improvement due to Sophia across the 4 categories is 20.4%. The underperformance for the fast case is due to increased prediction error. Naïve baseline has a significant loss compared to default: 21.6%.

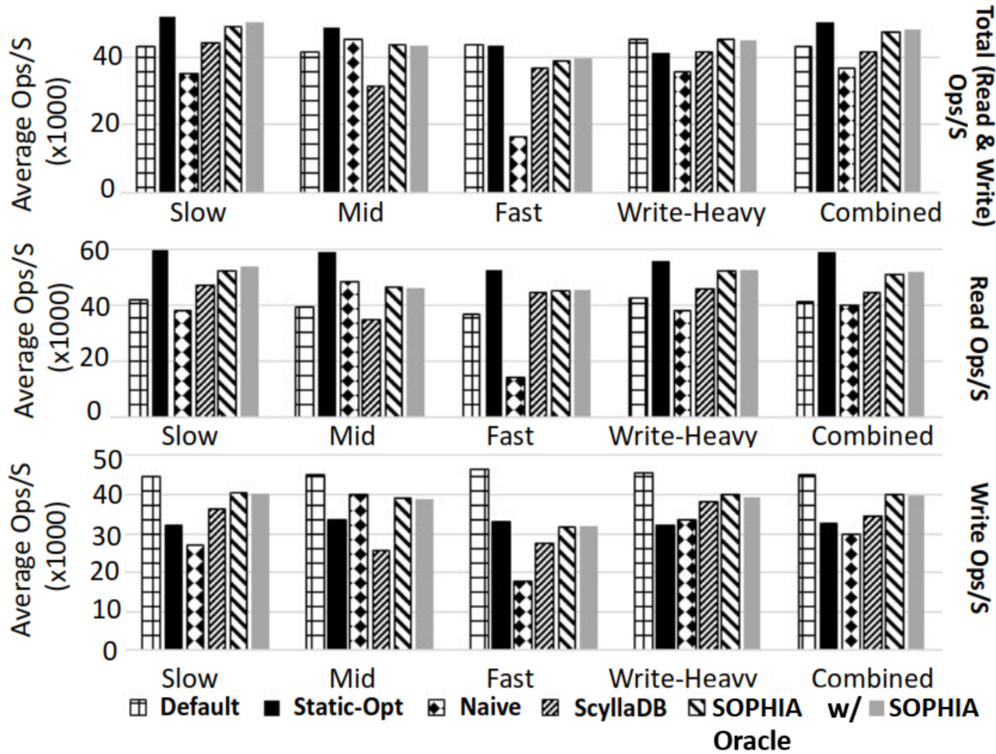


Figure 3.3. Improvement for four different 30-minute test windows from MG-RAST real traces over the baseline solutions.

The static optimized configuration (which for this workload favors read-heavy pattern) has a slightly higher throughput over Sophia by 6.3%. This is because the majority of the selected samples are read periods (RR=1), which hides the gain that Sophia achieves for write periods. However, we see that with respect to write operations, Sophia achieves 17.6% higher throughput than the static optimized configuration. Increased write throughput is critical for MG-RAST to support the bursts of intense writes. This avoids unacceptable queuing of writes, which can create bottlenecks for subsequent jobs that rely on the written shared dataset. Moreover, we observe that Sophia performs within 2-3% to Sophia w/ Oracle in all scenarios, which shows the minor impact of the workload predictor accuracy. For instance, Sophia w/ Oracle shows a 2% reduction in performance compared to Sophia in the slow trace. This is because Oracle has perfectly accurate predictions for $T_L = 5min$ only. With this very short lookahead, Sophia makes greedy reconfiguration decisions, and hence does not achieve globally optimal performance over other baselines.

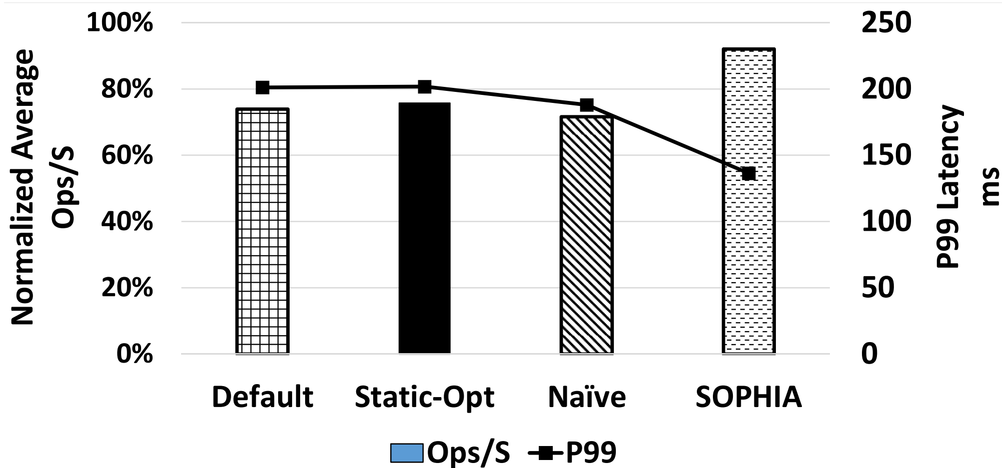


Figure 3.4. Gain of applying Sophia to the bus-tracking application. We use 8 Cassandra servers with RF=3, CL=1. A 100% on the Y-axis represents the theoretical best performance. Sophia achieves improvements of 24.5% over default, 21.5% over Static-Opt, and 28.5% over naïve.

ScyllaDB has an auto-tuning feature that is supposed to continuously react to changes in workload characteristics and the current state (such as, the amount of dirty memory state).

ScyllaDB is claimed by its developers to outperform Cassandra in all workload mixes by an impressive 10X [54]. However, this claim is not borne out here and only in the read-heavy case (the "Slow" scenario) does ScyllaDB outperform. Even in this case, Sophia is able to reconfigure Cassandra at runtime and turn out a performance benefit over ScyllaDB. We conclude that based on this workload and setup, a system owner can afford to use Cassandra with Sophia for the *entire range* of workload mixes and not have to transition to ScyllaDB.

Experiment 2: Tiramisu Workload

We evaluate the performance of Sophia using the bus-tracking application traces. Figure 3.4 shows the gain of using Sophia over the various baselines. In this experiment, we report the normalized average Ops/s instead of the absolute average Ops/s metric. This means we normalize each of the 4 operation's throughputs by dividing by the maximum Ops/s seen under a wide variety of configurations and then average the 4 normalized throughputs. The reason for this is that for this workload, different operations have vastly different throughput

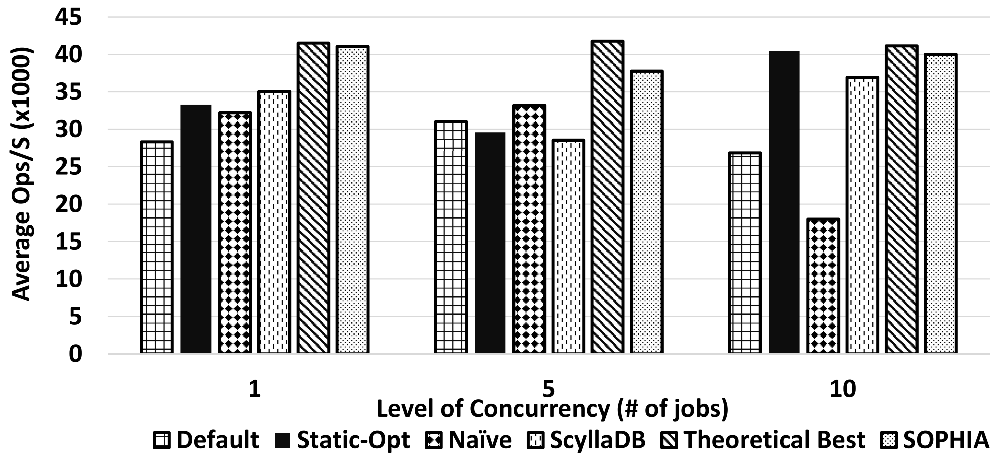


Figure 3.5. Improvement for HPC data analytics workload with different levels of concurrency. We notice that SOPHIA achieves higher average throughput over all baselines

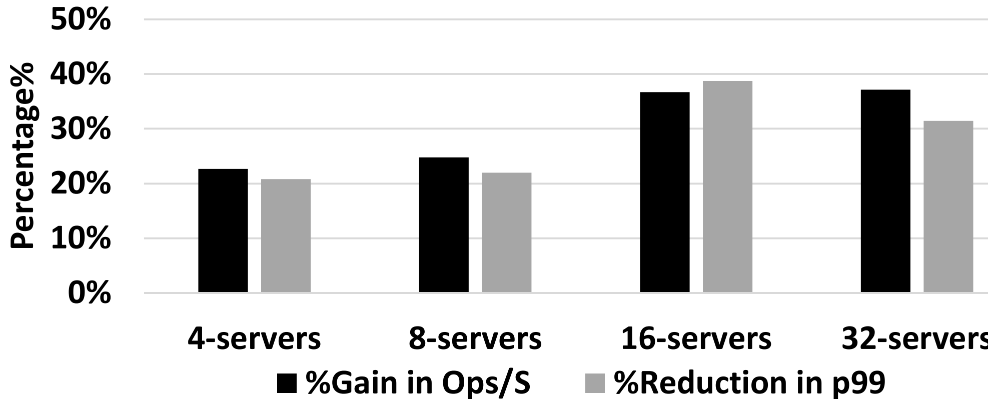


Figure 3.6. Improvement with scale using HPC workload with 5 jobs with RF=3 and CL=1. SOPHIA provides consistent gains across scale because the cost of reconfiguration does not change with scale (for the same RF and CL). The higher gains for 16 and 32 servers is due to the use of M5 instances, which can be exploited by SOPHIA better than Static-Opt.

scales. For example, when the workload switches to a Scan-heavy phase, the performance of the cluster varies from 34 Ops/s to 219 Ops/s depending on the configuration of the cluster. For an Update-heavy phase, the performance varies from 1,739 Ops/s to 5,131 Ops/s. This is because Scan is a much heavier operation for the DBMS compared to Update.

SOPHIA outperforms default configuration by 24.5%, Static-Opt by 21.5%, and Naïve by 28.5%. The gains are higher because SOPHIA can afford longer lookahead times with accurate workload prediction. We notice that Naïve is achieving a comparable performance to both Default and Static-Opt configurations, unlike MG-RAST. This is because the frequency of workload changes is lower here. However, Naïve still renders the data unavailable during the reconfiguration period.

Workload Prediction Model: Unlike MG-RAST, the bus-tracking application traces show a daily pattern which allows our prediction model to provide longer lookahead periods with high accuracy (Table 3.1). We use a Markov Chain prediction model to capture the workload switching behavior. We start by defining the state of the workload as the proportion of each operation type in an aggregation interval (15 minutes in our experiments). For example, Update=40%, Read=20%, Scan=40%, Insert=0% represents a state of the workload. We use a quantization level of 10% in any of the 4 dimensions to define the state.

We use the second-order Markov Chain with a lookahead period of 5 hours as this is when our prediction error is $\leq$ 8%. As expected theoretically, the second order model is more accurate at all lookahead times, since there is enough training data available for training the models. Seeing the seeming regular diurnal and weekly pattern in the workload, we create two simple predictor straw-men that uses only the current time-stamp or the current time-stamp and day of the week as input features to perform prediction. The predicted workload is the average of the mixtures at the previous 10 points. These predictors have unacceptably high RMSE of 31.4% and 24.0%. Therefore, although the workload is showing a pattern, we cannot generate the optimal plan once and use it for all subsequent days. Therefore, online workload monitoring and prediction is needed to achieve the best performance

Experiment 3: HPC Data Analytics

We evaluate the performance of Sophia using HPC data analytics workload patterns described in Section 3.4. Here our lookahead is the size of the job queue, which is conservatively taken as 1 hour.

Figure 3.5 shows the result for the three levels of concurrency (1, 5, and 10 jobs).

We see that Sophia outperforms the default for all the three cases, with average improvement of 30%. In comparison with Static-Opt (which is a different configuration in each of the three cases), we note that Sophia outperforms for the 1 job and 5 jobs cases by 18.9% and 25.7%, while it is identical for the 10 jobs case. This is because in the 10 jobs case, the majority of the workload lies between RR=0.55 and RR=0.85, and in this case, Sophia switches only once: from the default configuration to the same configuration as Static-Opt.

We notice that Sophia achieves within 9.5% of the theoretical best performance for all three cases.

We notice that Sophia achieves significantly better performance over Naïve by 27%, 13%, and 122% for the three cases. Naïve, in fact, degrades the performance by 32.9% (10 concurrent jobs).

In comparison with ScyllaDB, Sophia achieves a performance benefit of 17.4% on average, which leads to a similar conclusion as in MG-RAST about the continued use of Cassandra.

Experiment 4: Scale-Out & Greater Volume

Figure 3.6 shows the behavior of Sophia with increasing scale using the data analytics workload. We show the comparison between Sophia and Static-Opt (all other baselines performed worse than Static-Opt). We use a weak scaling pattern, i.e., keeping the amount of data per server fixed while still operating close to saturation. We increase the number of shooters as well to keep the request rate per server fixed. By our design (Sec. 4.5), the number of reconfiguration steps stays constant with scale.

We notice that the network bandwidth needed by Cassandra's gossip protocol increases with the scale of the cluster, causing the network to become the bottleneck in the case of 16 and 32 servers when M4.xlarge instances are used. Therefore, we change the instance

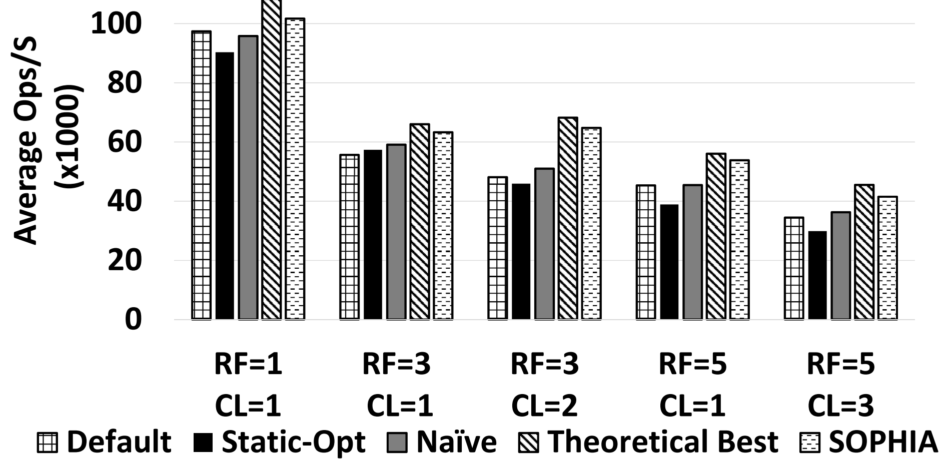


Figure 3.7. Effect of varying RF and CL on system throughput. We use a cluster of 8 nodes and compare the performance of SOPHIA to Default, Static-Opt, and naïve. SOPHIA outperforms the static baselines and approaches the theoretical best as RF–CL increases.

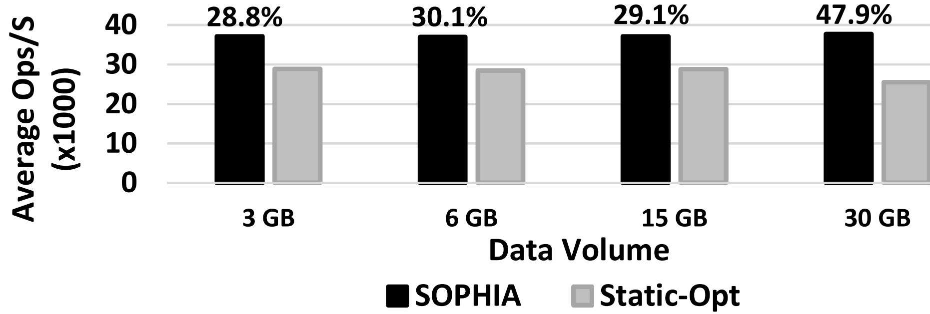


Figure 3.8. Effect on increasing data volume per node. We use a cluster of 4 servers and compare the performance to the static optimized. The results show that SOPHIA's gain is consistent with increasing data volumes per node.

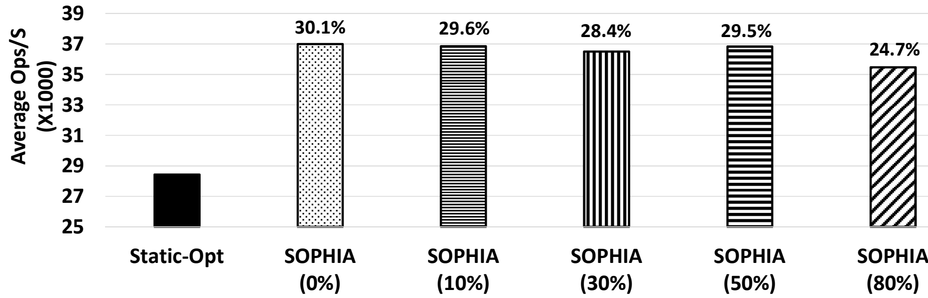


Figure 3.9. Effect of noise in workload prediction on the performance of SOPHIA on the data analytics workload with level of concurrency = 5. The percentage represents the amount of noise added to the predicted workload pattern.

type to M5.xlarge in these cases (network bandwidth of 10 Gbit/s compared to 0.74 Gbit/s). The results show that SOPHIA's optimal reconfiguration policy has a higher performance over Static-Opt across all scales. Moreover, we see a higher gain in the cases of 16 and 32 servers since M5 instances have higher CPU power than M4 ones. This extra CPU capacity allows for faster leveled compaction, which is used by SOPHIA's plan (while Static-Opt uses size-tiered compaction), and hence leads to greater performance difference for reads.

We also evaluate SOPHIA with the same workload when the data volume per node increases. We vary the amount of data loaded initially into each node (in a cluster of 4 nodes) and measure the gain over Static-Opt in Figure 3.8. For the 30GB case, the data volume grows beyond the RAM size of the used instances (M4.xlrage with 16 GB RAM). We notice that the gain from applying SOPHIA's reconfiguration plan is consistent with increasing the data volume from 3 GB to 30 GB. We also notice that the gain increases for the case of 30 GB. This is also due to the different compaction methods used by Static-Opt (size-tiered) and SOPHIA (Leveled compaction), the later can provide better read performance with increasing data volumes. However, this benefit of Leveled compaction was not captured by predictions, which was trained on a single node with 6 GB of data. This can be addressed by either replacing by a data volume-aware static tuner, or re-training when a significant change in data volume per node occurs.

Experiment 5: Varying RF and CL

We evaluate the impact of applying SOPHIA to clusters with different RF and CL values. We use the HPC workload with 5 concurrent jobs. We fix the number of nodes to 8 and vary RF and CL as shown in Figure 3.7 (CL quorum implies CL = $\lceil RF/2 \rceil$). We notice that SOPHIA continues to achieve better performance than all 3 static baselines for all RF, CL values. For RF=1, CL=1, we use SOPHIA-aggressive because when RF=CL, we cannot reconfigure the cluster without degrading availability. The key observation is that SOPHIA's performance gets closer to the *Theoretical best* as RF-CL becomes higher (compare the RF=3,CL=1 to the RF=5,CL=1 case). This is because the number of steps SOPHIA needs to perform the

reconfiguration is inversely proportional to RF-CL as discussed in Sec. 4.5). This allows SOPHIA to react faster to changes in the applied workload and thus achieve better performance. Moreover, we notice that the performance of the cluster degrades with increasing RF or CL. Increasing RF increases the data volume stored by each node, which increases the number of SSTables and hence reduces the read performance. Also increase in CL requires more nodes to respond to each request before acknowledgment to the client, which also reduces the performance.

Experiment 6: Noisy Workload Predictions

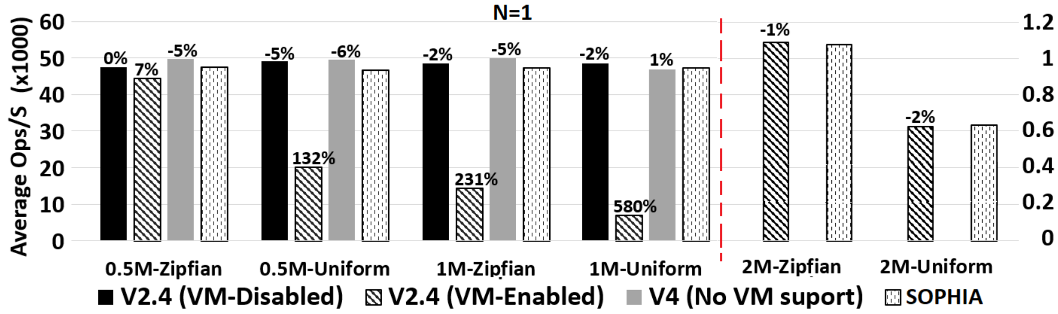
We show how sensitive SOPHIA is to the level of noise in the predicted workload pattern. We use the HPC workload with 5 concurrent jobs. In HPC queues, there are two typical sources of such noise—an impatient user removing a job from the queue and the arrival of hitherto unseen jobs. We add noise to the predicted workload pattern $\sim$ U(-R,R), where R gives the level of noise. The resulting value is bounded between 0 and 1.

From Figure 3.9, we see that adding noise to SOPHIA slightly reduces its performance. However, such noise will not cause significant changes to SOPHIA's optimal reconfiguration plan. This is because SOPHIA treats each entry in the reconfiguration plan as a binary decision, i.e., reconfigure if Benefit $\geq$ Cost. So even if the values of both Benefit and Cost terms change, the same plan takes effect as long as the inequality still holds. This allows SOPHIA to achieve significant improvements for long-term predictions even with high noise levels.

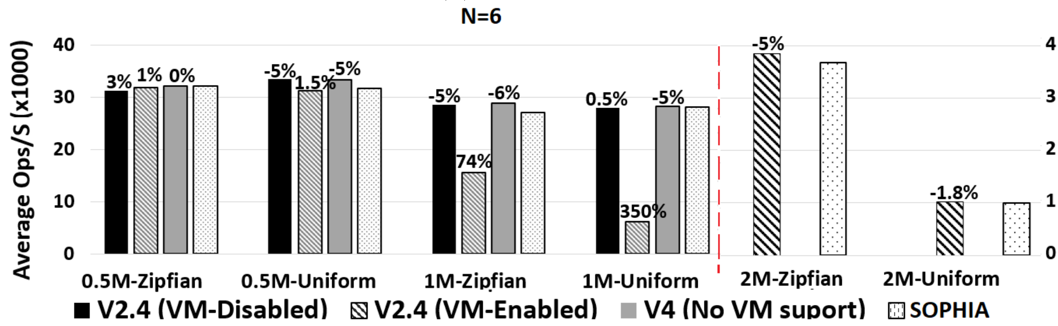
Experiment 7: Redis Case Study

We now show a case study with the popular NoSQL database Redis, which has a long-standing pain point in setting a performance-critical parameter against changing workloads.

Large-scale processing frameworks such as Spark can deliver much higher performance when combined with Redis due to its shared distributed memory infrastructure [58], [59]. Redis is an in-memory data store (stores all keys and values in memory) while writing to persistent storage is only supported for durability. However, in its earlier versions (till V2.4), Redis used to offer a feature called *Virtual Memory* [60]. This feature allowed Redis to work on datasets larger than the available memory by swapping rarely used values to disk, while



(a) Single node



(b) Cluster of 6 nodes

Figure 3.10. Impact of tuning Redis' VM configuration parameters with SOPHIA with the data analytics workload. The percentage improvement of SOPHIA is shown on each bar and the right Y-axis is for the 2M jobs. A missing bar represents a failed job. We notice that the current Redis fails for large workloads (2M), while SOPHIA achieves the best of both worlds

keeping all keys and hot values in memory. Since V2.4, this feature was removed as it caused serious performance degradation in many Redis deployments due to non-optimal setting as reflected in many posts in discussion forums [61]–[63]. We use SOPHIA to tune this feature and compare the performance to three baselines: (1) Redis V2.4 with VM-disabled (Default configuration), (2) Redis V2.4 with VM-enabled, (3) Redis V4 with default configuration (no VM support, most production-proven).

To tune Redis' *VM*, we investigate the impact of two configuration parameters: (1) *vm-enable*: a Boolean parameter that enables or disables the feature. (2) *vm-max-memory*: the memory limit after which Redis starts swapping least-recently-used values to disk. These features cannot be reconfigured without a server restart.

We tune the performance of Redis for simulated data analytics workloads that vary with

respect to job sizes and access patterns. We use the popular YCSB (Yahoo! Cloud Serving Benchmark) tool [64] to simulate HPC workloads as in [65], [66]. We collect 128 data points for jobs that vary with respect to their sizes (0.5M, 1M, 2M), their access patterns (i.e., read-heavy vs write-heavy) and also their request distribution (i.e., Uniform vs Zipfian). We use 75% of the data points (selected uniformly) to train a linear regression model and 25% for testing. The model provides accurate prediction of throughput for any job and configuration (avg. R^2=0.92). Therefore, we use this simpler model in place of Rafiki.

Redis can operate in `Stand-alone` mode as well as a `Cluster` mode [67]. In `Cluster` mode, data is automatically sharded across multiple Redis nodes. We show the gain of using our system with Redis for both modes of operation. No replication is used for `Stand-alone` mode. Whereas for `Cluster` mode, we use a replication factor of 1 (i.e., a single slave per master). We use AWS servers of type C3.Large with 2 vCPUs and 3.75GB RAM each. Selecting such a small RAM server demonstrates the advantage of using *VM* with jobs that cannot fit in memory—1.8M records fit in the memory. We evaluate the performance SOPHIA on a single server (Figure 3.10a) as well as a cluster of 6 servers with 3 masters and 3 slaves (Figure 3.10b) and report the average. From Figure 3.10 we see that for all record sizes and request distributions, SOPHIA performs the best or close to the best. If records fit in memory, then the no VM configuration is better. For Uniform distribution, VM performs worst, because records often have to be fetched from disk. If records do not fit in memory, the no VM options (including the latest Redis) will simply fail (hence the lack of a bar for 2.0M records). Thus, SOPHIA, by automatically selecting the right parameters for changing workloads, can achieve the best of both worlds: fast in-memory database, and leverage disk in case of spillover.

4. OPTIMUSCLOUD: HETEROGENEOUS CONFIGURATION OPTIMIZATION FOR DISTRIBUTED DATABASES IN THE CLOUD

4.1 Abstract

Reconfiguring NoSQL databases under changing workload patterns is crucial for maximizing database throughput. This is challenging because of the large configuration parameter search space with complex interdependencies among the parameters. While state-of-the-art systems can automatically identify close-to-optimal configurations for static workloads, they suffer for dynamic workloads as they overlook three fundamental challenges: (1) Estimating performance degradation during the reconfiguration process (such as due to database restart). (2) Predicting how transient the new workload pattern will be. (3) Respecting the application's availability requirements during reconfiguration. Our solution, , addresses all these shortcomings using an optimization technique that combines workload prediction with a cost-benefit analyzer. computes the relative cost and benefit of each reconfiguration step, and determines an optimal reconfiguration for a future time window. This plan specifies when to change configurations and to what, to achieve the best performance without degrading data availability. We demonstrate its effectiveness for three different workloads: a multi-tenant, global-scale metagenomics repository (*MG-RAST*), a bus-tracking application (*Tiramisu*), and an HPC data-analytics system, all with varying levels of workload complexity and demonstrating dynamic workload changes. We compare 's performance in throughput and tail-latency over various baselines for two popular NoSQL databases, Cassandra and Redis.

4.2 Introduction

Cloud deployments reduce initial infrastructure investment costs and provide many operational benefits. An important class of cloud deployments is NoSQL databases, which allow applications to scale beyond the limits of traditional databases [64]. Popular NoSQL databases such as Cassandra, Redis, and MongoDB, are widely used in web services, big data

services, and social media platforms. Tuning cloud-based NoSQL databases for performance[1] under cost constraints is challenging due to several reasons.

First, the search space is very large due to VM configurations and database application configurations. For example, cloud services provide many VMs that vary in their CPU-family, number of cores, RAM size, storage, network bandwidths, etc., which affect the VM's $ cost. At the time of writing, AWS provides 133 instance types while Azure provides 146 and their prices vary by a factor of 5,000×.

On the NoSQL side, there are many performance-impacting configuration parameters. For example, Cassandra has 25 such parameters and sub-optimal parameter setting for one parameter (*e.g.* the *Compaction method*) can degrade throughput by 3.4× from the optimal. On the cloud side too, selecting the right VM type and size is essential to achieve the best Perf/$. *Second*, there is the need for *joint* optimization while taking into account the dependencies between the NoSQL-level and VM-level configurations. For example, our evaluation shows that the optimal cache size of Cassandra for a VM type `M4.large` (with 8GB of RAM) is 8× the optimal cache size for `C4.large` (with 3.75GB RAM). Additionally, larger-sized VMs do not always provide better Perf/$ [68] as they may overprovision resources and unnecessarily increase the $ cost.

Third, there are many use cases of cloud applications where the workload characteristics change over time, sometimes unpredictably, necessitating reconfigurations [69], [70]. A configuration that is optimal for one phase of the workload can become very poor for another phase of the workload. For example, in Cassandra, with a large working set size, reads demand instances with high memory, while writes demand high compute power and fast storage.

Changing the configuration at runtime for NoSQL databases, which are stateful applications (*i.e.* with persistent storage), has a performance impact due to the downtime caused to the servers being reconfigured.

---

[1]↑We use the standard metrics of throughput and (tail) latency for measuring database performance. Specifically, we target maximizing throughput normalized by price in $, *i.e.* performance-per-unit-$ or Perf/$ for short.

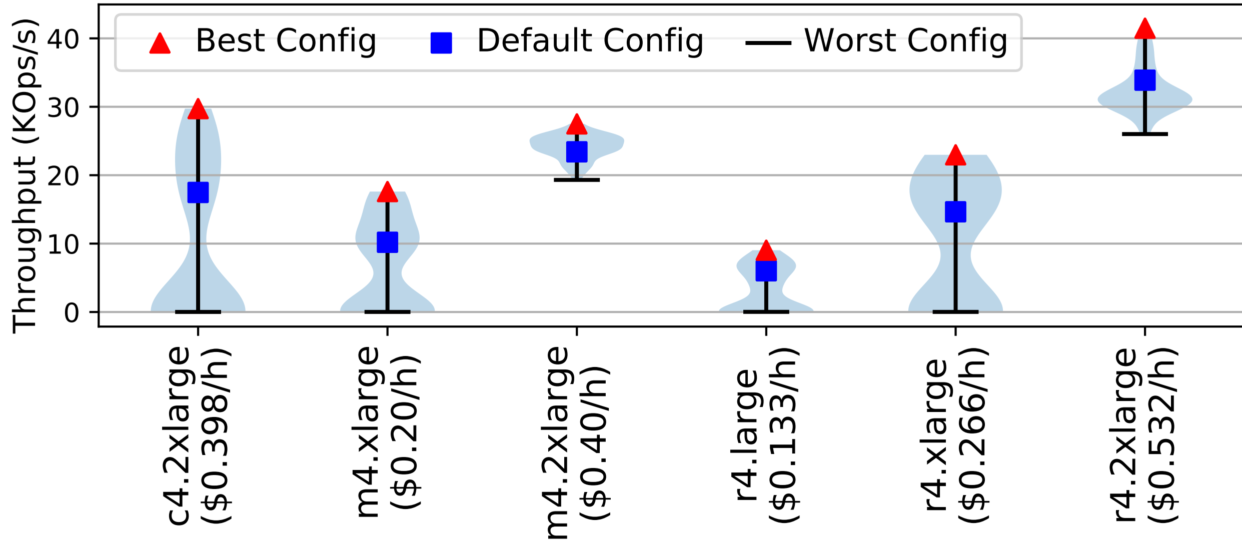


Figure 4.1. Violin plot showing performance (throughput) of Best, Default, and Worst database configurations across different EC2 VM types.

Therefore, for fast changing workloads, frequent reconfiguration of the overall cluster could severely degrade performance [71].

Consequently, deciding *which* subset of servers to reconfigure is vital to minimize reconfiguration performance hit and to achieve globally optimal Perf/$ while respecting the user's availability requirements. However, changing the configurations of only a subset of servers naturally leads to *heterogeneous* clusters, which no prior work is equipped to deal with.

Existing Solutions: State-of-the-art cloud configuration tuners such as CherryPick [72] and Selecta [73] focus mainly on stateless, recurring workloads, such as big-data analytics jobs, while Paris [68] relies on a carefully chosen set of benchmarks that can be run offline to fingerprint which application is suitable for which VM type. Due to their target of static workloads and stateless jobs, a single cloud configuration is selected based on a representative workload and then fixed throughout the operation period. However, small workload changes can cause these "static tuners" to produce drastically degraded configurations.

For example, a 25% increase in workload size with CherryPick makes the proposed configuration 2.6× slower than optimal (Section 5.4 in [72]). Also, in our experiments (Sec. 4.5.3), we find that CherryPick's proposed configuration for the write-heavy phase achieves only 12% of the optimal when the workload switches to a read-heavy phase.

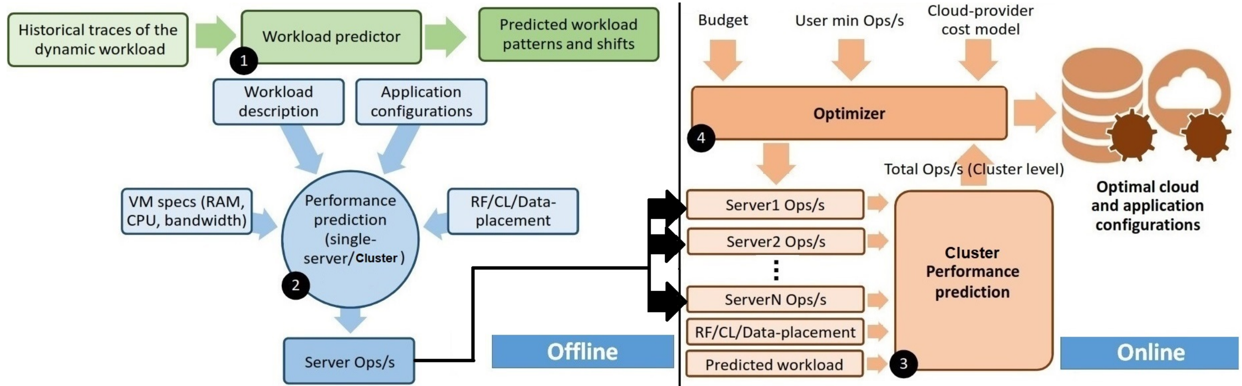


Figure 4.2. Overview of OptimusCloud's workflow. First, a workload predictor is trained with historical traces from the database to be tuned. Second, a single server performance predictor is trained to map workload description, VM specs, and NoSQL application configuration to throughput. Third, a cluster-level performance predictor is used to estimate the throughput of the heterogeneous cluster of servers. In the online phase, our optimizer uses this predictor to evaluate the fitness of different VM/application configurations and provides the best performance within a given budget.

Hence, these *prior systems are not suitable for dynamic workloads.*

SOPHIA [71] addresses database configuration tuning for clustered NoSQL databases and can handle dynamic workloads. However, like the static tuner RAFIKI [28], SOPHIA's design focuses only on NoSQL configuration tuning and does not consider cloud VM configurations nor dependencies between VM and NoSQL configurations. Naïvely combining the NoSQL and VM configuration spaces causes a major increase in the search space size and limits SOPHIA's ability to provide efficient configurations (Sec. 4.4.5). Further, due to its atomic reconfiguration strategy (*i.e.* either reconfigure all servers or none), it suffers from all the drawbacks of the homogeneity constraint. Table 4.1 compares key features of OptimusCloud to various prior works in this field.

Our Solution: We introduce our system OptimusCloud, which jointly tunes the database and cloud (VM) configurations for dynamic workloads. **There are three key animating insights behind the design of OptimusCloud.** The *first* is that jointly tuning the database and cloud (VM) configurations for dynamic workloads is essential. To show how important this is, we benchmark one Cassandra server with a 30-min trace from one of our

Table 4.1. OptimusCloud's key features vs. existing systems.

Feature	Single Server Prediction	Multiple Server Prediction (Homogenous)	Multiple Server Prediction (Heterogeneous)	Application Level Configurations	Cloud Configurations	Dynamic Workloads + Cost Benefit Analysis
Ernest (NSDI'16) CherryPick (NSDI'17)	Supported	Supported	Not Supported	Not Supported	Supported	Not Supported
Selecta (ATC'18)	Supported	Supported	Partially Supported	Not Supported	Supported	Not Supported
Rafiki (Middleware'17) / Ottertune (Sigmod'17)	Supported	Not Supported	Not Supported	Supported	Not Supported	Not Supported
Sophia (Usenix ATC'19)	Supported	Supported	Not Supported	Supported	Not Supported	Partially Supported
OptimusCloud	Supported	Supported	Supported	Supported	Supported	Supported

Supported / Partially Supported / Not Supported

three workloads (MG-RAST) on 9 different EC2 VM types[2]. For each type, we use 300 different database configurations selected through grid search. We show the performance in terms of Ops/s for the best, default, and worst configurations in Fig. 4.1. We see a big variance in performance w.r.t. the database configurations—up to 74% better performance over default configurations (45% on average). Further, the best configurations vary with the VM type and size (for the 6 VM types shown here, there are 5 distinct best DB configurations).

This emphasizes the need for tuning both types of configurations *jointly* to achieve the best Perf/$. The *second key insight* is that in order to optimize the Perf/$ for a dynamic workload, it is necessary to perform non-atomic reconfigurations, *i.e.* for only part of the cluster. Reconfiguration in a distributed datastore is a sequential operation (in which one or a few servers at a time are shutdown and then restarted) to preserve data availability [71], [74]. This operation causes transient performance degradation or lower fault tolerance. Reconfiguration is frequent enough for many workloads that this performance degradation should be avoided, *e.g.* MG-RAST has a median of 430 significant switches per day in workload characteristics. Accordingly, *heterogeneous configurations* have the advantage of minimizing the performance hit during reconfiguration.

[2]↑MG-RAST is the largest metagenomics portal and data repository and gets queries from across the globe which cause unpredictable read-write patterns to the backend Cassandra.

Further, in the face of dynamic workloads, there may only be time to reconfigure part of the overall cluster.

Also, from a cost-benefit standpoint, maximizing performance does not need *all* instances to be reconfigured (such as to a more resource-rich instance type), rather a carefully selected subset. We give a simple example to make this notion concrete in Section 4.3.

The *third key insight* is that for a particular NoSQL database (with its specifics of data placement and load balancing), it is possible to create a model to map the configuration parameters to the performance of each server. From that, it is possible to determine the overall heterogeneous cluster's performance. OptimusCloud leverages performance modeling to search for the optimal cluster configuration.

The workflow of OptimusCloud comprises offline training and online prediction and reconfiguration phases as shown in Fig. 4.2. At runtime, OptimusCloud takes user requirements of budget, availability, and consistency. It then combines the performance model with a workload predictor and a cost-benefit analyzer to decide: when the workload changes sufficiently, what should be the new (possibly heterogeneous) configuration. It decides what minimal set of servers should be reconfigured.

Evaluation: We apply OptimusCloud to two popular NoSQL databases—Cassandra and Redis—and evaluate the system on traces from two real-world systems, and one simulated trace from an HPC analytics job queue. All three use cases represent dynamic workloads with different query blends. We evaluate the Perf/$ achieved by OptimusCloud and compare this to three leading prior works, CherryPick [72], Selecta [73], and SOPHIA [71]. Additionally, we compare ourselves to the best static configuration determined with oracle-like prediction of future workloads and the theoretical best.

OptimusCloud achieves between 80-90% of the theoretical best performance for the 3 workloads and achieves improvements between 9%-86.5%, 18%-173%, 17%-174%, and 12%-514% in Perf/$ over Homogeneous-Static, Cherry-Pick, Selecta, and SOPHIA respectively without degrading P99 latency (Sec. 4.5).

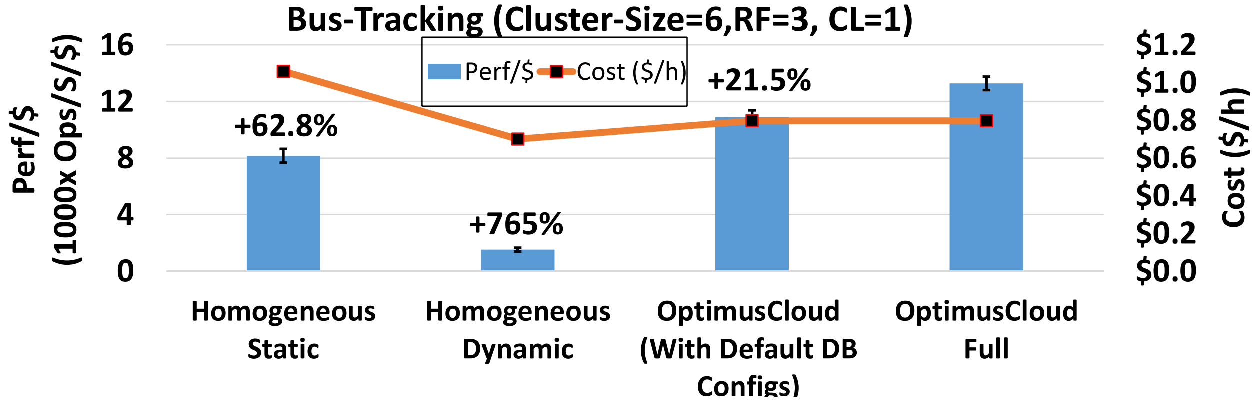


Figure 4.3. Importance of creating heterogeneous clusters (OptimusCloud Full) over homogeneous clusters (both Static and Dynamic) for Bus-Tracking application. Tuning both application and cloud configurations (OptimusCloud Full) has benefit over tuning only the VM configuration (3rd bar from left). The percentage value on the top of each bar denotes how much OptimusCloud improves over that particular scheme.

Fig. 4.3 shows the improvement in Perf/$ due to OptimusCloud's heterogeneous configurations.

We make the following novel contributions in this paper.

1. We design a performance modeling-based technique for efficient *joint optimization* of database *and* cloud configurations to maximize the Perf/$ of a clustered database.
2. We design a technique to identify the minimal set of servers in a clustered database to reconfigure (concurrently) to obtain a throughput benefit. This naturally leads to heterogeneous configurations. To reduce the much larger search space that this causes, we design for a simplification that groups multiple servers that should be configured to the same parameters.
3. We show that OptimusCloud generalizes to two distinct NoSQL databases and different workloads, cluster sizes, data volumes, and user-specified requirements for replication and data consistency.

The rest of the paper is organized as follows. Section 4.3 gives the necessary background for the problem and a quantitative rationale.

Section 7.3 describes the details of OptimusCloud's design. We evaluate OptimusCloud in Section 4.5.

4.3 Background and Rationale

To evaluate the generalizability of , we select two popular NoSQL databases with very different architectures.

4.3.1 Cassandra

designed for high scalability, availability, and fault-tolerance. To achieve these, Cassandra uses a peer-to-peer (P2P) replication strategy, allowing multiple replicas to handle the same request. Other popular datastores such as DynamoDB [75] and Riak [76] implement the same P2P strategy and we select Cassandra as a representative system from that category.

Cassandra's replication strategy determines where replicas are placed. The number of replicas is defined as "Replication Factor" (RF). By default, Cassandra assigns an equal number of tokens to each node in the cluster where a token represents a sequence of hash values for the primary keys that Cassandra stores. Based on this token assignment, a Cassandra cluster can be represented as a ring topology [77].

Fig. 4.5 shows an example of 4 Cassandra servers and RF of 2.

4.3.2 Redis

Redis is an in-memory database and serves all requests from the RAM, while it writes data to permanent storage for fault tolerance. This design principle makes Redis an excellent choice to be used as a cache on top of slower file systems or datastores [78]. Redis can operate as either a stand-alone node or in a cluster of nodes [79] where data is automatically sharded across multiple Redis nodes. Our evaluation applies to the clustered mode of Redis. When a Redis server reaches the maximum size of its allowed memory (specified by the `maxmemory` configuration parameter), it uses one of several policies to decide how to handle new write requests. The default policy will respond with error. Other policies

will replace existing records with the newly inserted record (the `maxmemory-policy` configuration parameter specifies which records will be evicted).

The value of `maxmemory` needs to be smaller than the RAM size of the VM instance and the headroom that is needed is workload dependent (lots of writes will need lots of temporary buffers and therefore larger headroom). Thus, it is challenging to tune `maxmemory-policy` and `maxmemory` parameters with changing workloads and these two form the target of our configuration decision.

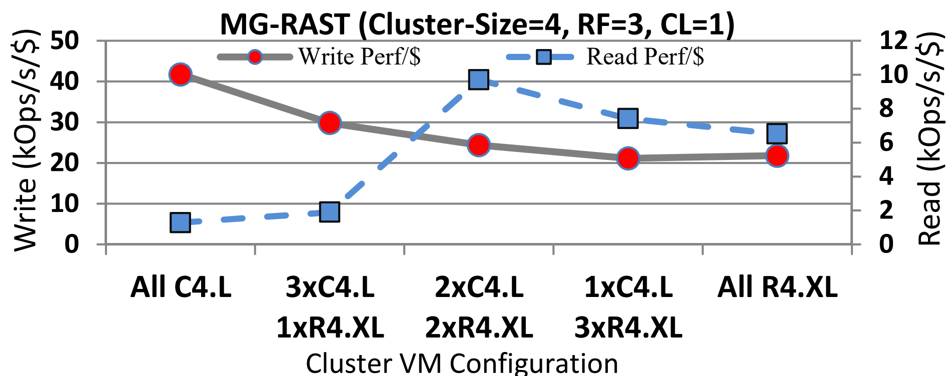


Figure 4.4. Change in Perf/$ for the write (solid) and read throughput (dotted) as we reconfigure the nodes from C4.large to R4.xlarge.

4.3.3 Example Rationale for Heterogeneous Configurations

Here we give a motivating example for selecting subset of servers to reconfigure. Consider a Cassandra cluster of 4 nodes with a consistency-level (CL[3]) = 1 and replication-factor (RF[4]) = 3, *i.e.* any pair of nodes has a complete copy of all the data. Also, assume that we only have two cloud configurations: C4.large, which is compute-optimized, and R4.xlarge, which is memory-optimized.

C4.large is cheaper than R4.xlarge by 58% [80], whereas R4.xlarge has larger RAM (30.5GB vs 3.75GB) and serves read-heavy workloads with higher throughput. Now we test the performance of all possible combinations of VM configurations (All C4.L, 1 C4.L

[3]↑CL: the minimum number of Cassandra nodes that must acknowledge a read or write operation before the operation can be considered successful
[4]↑RF: the total number of replicas for a key across a Cassandra cluster

+ 3R4.XL, ... etc.) for both read-heavy and write-heavy phases of the MG-RAST workload and show the saturation level throughput for each configuration in Fig. 4.4. The "All C4.large" configuration achieves the best write Perf/$ (41.7 KOps/s/$), however, it has the worst read Perf/$ (only 1.28 KOps/s/$) because reads of even common records spill out of memory. Now if two servers are reconfigured to R4.xlarge, the write Perf/$ decreases (24.4 KOps/s/$), while the read performance increases significantly (9.7 KOps/s/$), showing an improvement of 7.5× for read throughput over the all C4.large configuration. The reason for this huge improvement is Cassandra's design by which it redirects new requests to the fastest replica [81], directing all read requests to the two R4.xlarge servers. *Now we notice that switching more C4.large servers to R4.xlarge does not show any improvement in either reads or writes Perf/$, as the two R4x.large servers are capable of serving the applied workload with no queued requests.* This means that switching more servers will only reduce the Perf/$. Thus, the best Perf/$ is achieved by configuring to all C4.large in write-heavy phases, while configuring only 2 servers to R4x.large in read-heavy phases. **Therefore, heterogeneous configurations can achieve better Perf/$ compared to homogeneous ones under mixed workloads.**

4.4 Design

4.4.1 Workload Representation and Prediction

OptimusCloud uses a query-based model [82] to represent time-varying workloads. This model characterizes the applied workload in terms of the proportion of the different query types and the total volume of queries, denoted by W.

We use a workload predictor to learn time-varying patterns from the workload's historical traces, and predict the workload characteristics for a particular lookahead period. We notate the time varying workload at a given point in time t as $W(t)$.

The task of the workload predictor is to provide OptimusCloud with $W(t+1)$ given $W(t)$,$W(t-1)$,... , $W(t-h)$, where h is the length of history. OptimusCloud then iteratively predicts the workload till a lookahead time l, *i.e.* $W(t+\mathrm{i}), \forall \mathrm{i} \in (1, l)$. We

execute OptimusCloud with a simple Markov-Chain prediction model for both MG-RAST and Bus-tracking workloads while we have a deterministic fully accurate predictor for HPC.

We do not claim any novelty in workload prediction and OptimusCloud is modular enough to easily integrate more complex estimators, such as neural networks [33], [83].

4.4.2 Performance Prediction

Combining NoSQL and cloud configurations produces a massive search space, which is impractical to optimize through exhaustive search. However, it is well known that not all the application parameters impact performance equally [10], [28], [71] and therefore OptimusCloud reduces the search time by automatically selecting the most impactful parameters. Further, there exist dependencies among parameters, such as the dependency between the VM type (EC2) and Cassandra's *file-cache-size* (FCS) (Fig. 4.7). OptimusCloud uses *D*-optimal design [84] to optimize the offline data collection process for training our performance model.

D-optimal design answers this question: "*Given a budget of N data points to sample for a workload, which N points are sufficient to reveal the dependencies between configuration parameters?*".

We experimentally determine that the significant dependencies in our target applications are at most pairwise and therefore we restrict the search to linear and quadratic parameters.

We create a set of filters for feasible combinations of parameter values by mapping each parameter to the corresponding resource (*e.g.* *file-cache-size* parameter is mapped to RAM). Afterward, we check that the sum of all parameters mapped to the same resource is within that resource limit of the VM (*e.g.* the total size of all Cassandra buffer memories should not exceed the VM instance memory).

We feed to *D*-optimal design the budget in terms of the number of data points that we can collect for offline training.

After collecting the data points determined by the *D*-optimal design, we train a random forest to act as a regressor and predict the performance of a single NoSQL server for any given set of configuration parameters, both database and VM.

The average output of the different decision trees is taken as the final output. We choose random forest over other prediction models because of its easily interpretable results [85] and it has only two hyper-parameters to tune (`max_depth` and `forest_size`) compared to black-box models such as DNNs.

OptimusCloud trains a second random forest model to predict the overall cluster performance, using the predicted performance for each server, RF, CL and data-placement information. For both random forests, we use 20 trees and a maximum depth of each as 5 as that gives the best result within reasonable times.

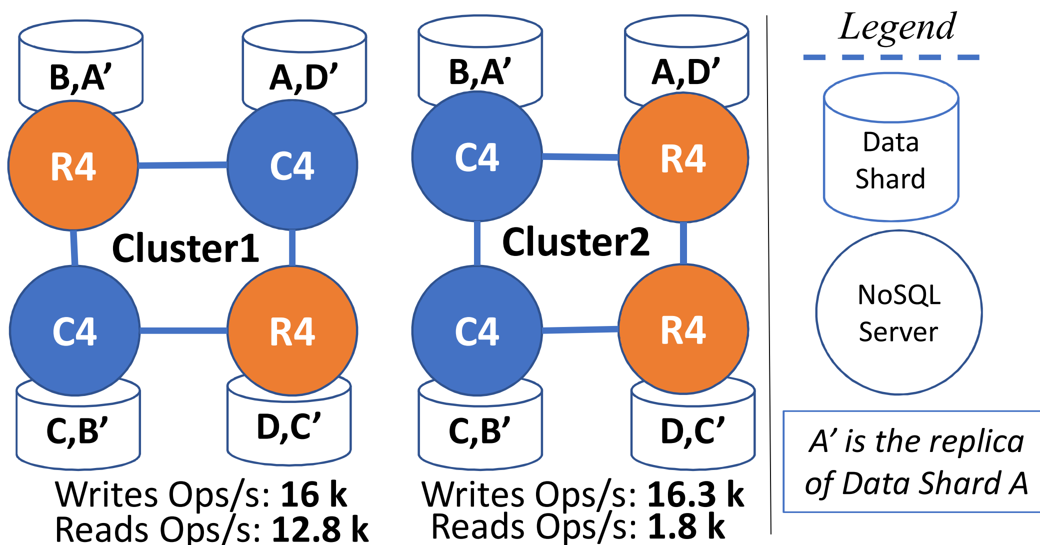


Figure 4.5. (RF=2, CL=1) Cluster performance depends not just on the configuration of each server, but also on the relative positions of the instances on the token ring. Cluster1 achieves 7× reads Ops/s over Cluster2 with the same VM types and sizes.

4.4.3 Selection of Servers to Reconfigure

Selecting the right servers to reconfigure in a cluster is essential to achieve the best Perf/$. We introduce the notion of *Complete-Sets* to determine the right subset of servers to reconfigure. **We define a *Complete-Set* as the minimum subset of nodes for which the union of their data records covers all the records in the database at least once.**

To see why the notion of *Complete-Set* is important, consider the two clusters shown in

Fig. 4.5. Both clusters 1 and 2 use 2 C4.large and 2 R4.large and hence have the same $ cost. However, *Cluster1* achieves 7× the read Ops/s compared to *Cluster2*. The reason for the better performance of *Cluster1* is that it has one *Complete-Set* worth of servers configured to the memory-optimized R4.large architecture and therefore serves all read requests efficiently. On the other hand, *Cluster2*'s read performance suffers since all read requests to shard `B` (or its replica `B'`) have to be served by one of the C4.large servers, which has a smaller RAM and therefore serves most of the reads from desk. Accordingly, read requests to shards `B` or `B'` represent a bottleneck in *Cluster2* and cause a long queuing time for the reading threads, which brings down the performance the *entire* cluster for all the shards.

This means that all the servers within a Complete-Set must be upgraded to the faster configuration for the cluster performance to improve.

Otherwise, the performance of the Complete-Set will be bounded by the slowest server in the set.

OptimusCloud partitions the cluster into one or more *Complete-Sets* using the cluster's data placement information. To identify the Complete-Sets, we collect the data placement information for each server of the cluster.

OptimusCloud queries this information either from any server (such as in Cassandra, using `nodetool ring` command) or from one of the master servers (such as in Redis, using `redis-cli cluster info` command). In Redis, identifying the Complete-Sets is easier since data tokens are divided between the master nodes only, while slaves have exact copies of their master's data. Therefore, a *Complete-Set* is formed by simply selecting a single slave node for every master node.

Maintaining Data Availability: To maintain data availability during reconfiguration of a Cassandra cluster, at least CL replicas of each data record must be up at any point in time. This puts an upper limit on the number of *Complete-Sets* that can be reconfigured concurrently as $Count(\textit{Complete-Sets}) - CL$.

We show that the number of Complete-Sets in a cluster is *not* dependent on the number of nodes in the cluster, but is a constant factor. This is because when the cluster size increases, the range of keys assigned to every node decreases and therefore the number of nodes that

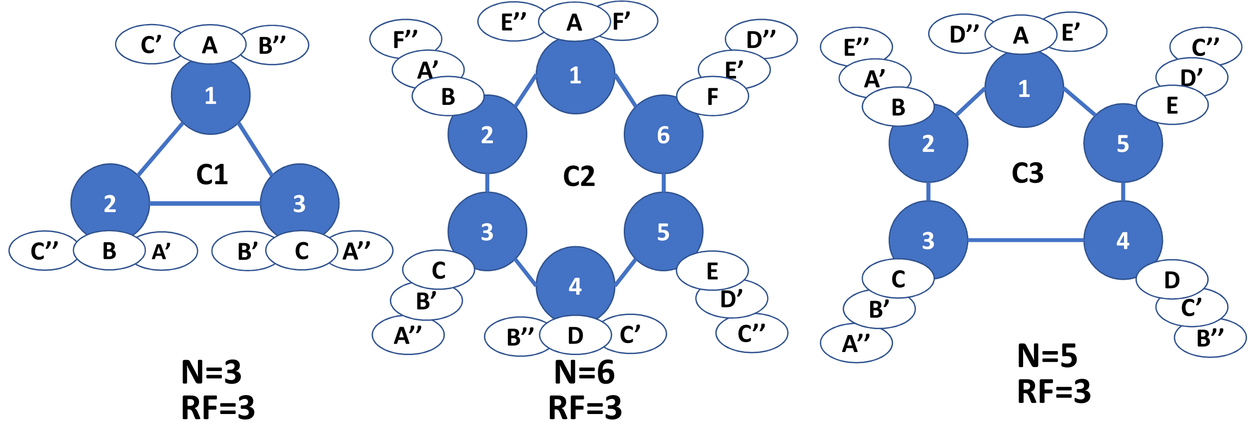


Figure 4.6. Replication examples with 3 different cluster sizes with RF=3. Cluster C1 has 3 nodes and each node has a complete copy of the data, therefore each node is a *Complete-Set*. Cluster C2 has 6 nodes and has the following *Complete-Set*s: [1,4] , [2,5] & [3,6]. Cluster C3 has 5 nodes (not divisible by RF=3), therefore it has two *Complete-Set*s: [1,3 (or 4)], [2,4 (or 5)].

form a Complete-Set increases. This means that since OptimusCloud reconfigures the instances in groups of one or more Complete-Sets concurrently, the total time to reconfigure a cluster is a constant factor independent of the cluster size. Figure 4.6 shows examples of *Complete-Set* for different cluster sizes

Property: OptimusCloud partitions the cluster into S Complete-Sets, and S is independent of the cluster size N.

Proof. For a cluster of N servers with replication factor RF, there exists a total of RF copies of each record in the cluster, with no two copies of the same record stored in the same server. Assuming each node in the cluster is assigned an equal portion of the data (which NoSQL load-balancers try to achieve [86]), the size of a *Complete-Set* is $\text{Size}_{CompSet} = \lceil \frac{N}{RF} \rceil$.

Consequently, the number of *Complete-Sets* in the cluster $S = \lfloor \frac{N}{\text{Size}_{compSet}} \rfloor$.

If RF divides N, then the number of *Complete-Sets* is $S = \frac{N}{\text{Size}_{CompSet}} = RF$. Else, say $N\%RF = r$, then $S = \frac{RF}{1-r/N+RF/N}$, which is $\approx RF$ since in practice RF is not large, 3 being a practical upper bound. Thus, the number of *Complete-Sets* is independent of the cluster size and hence the reconfiguration time is also a constant.

□

Search Space Size Reduction: Heterogeneous configurations make the search space size much larger than with homogeneous configurations. Consider a cluster of N nodes and I VM options to pick from. If we are to pick a homogeneous cloud configuration for the cluster, we have I options. However, if we are to pick a heterogeneous cloud configuration, our search space becomes I^N. If we assume balanced data placement among the servers in the cluster (as clustered NoSQL databases are designed for), the search space becomes $C(N+I-1, I-1)$ (distribute N identical balls among I boxes). However, this search space size is still too large to perform an exhaustive search to pick the optimal configurations. A cluster of size N=20 nodes and I=15 VM options gives 1.3×10^9 different configurations to select from. One may use domain-specific insights about the domain to reduce the search space for specific applications [87] or for customized distributed strategies [88]. However we aim for generalizability here.

We use one insight about Complete-Sets to reduce the search space. The nodes within each Complete-Set should be homogeneous in their configuration. Otherwise, the performance of the Complete-Set will be equal to that of the slowest node in the set. *This means that the smallest atomic unit of reconfiguration is one Complete-Set.*

This insight reduces the search space, while still allowing different *Complete-Sets* to have different configurations. Thus, the search space reduces to $C\left(S+I-1, I-1\right)$=680 configurations when $S = RF = 3$.

Also note that the configuration search space is constant rather than growing with the size of the cluster.

4.4.4 Selecting the Reconfiguration Plan

Objective Function Optimization

The objective of OptimusCloud is to find a reconfiguration plan that maximizes Perf/$ of the cluster under a given budget and with a minimum acceptable throughput. A *reconfiguration plan C* is represented as a time series of a vector of configurations (both NoSQL and VM):

$$\mathbf{C} = [\{C_1, C_2, \cdots, C_M\}, \{t_1, t_2, ..., t_M\}] \tag{4.1}$$

Where M is the number of steps in the plan and timestamp t_{i} represents how long the configuration C_{i} is applied. The lookahead is $t_L = \sum_{\mathrm{i}=1}^{M} t_{\mathrm{i}}$. The optimization problem is defined as:

$$\begin{gathered}\mathbf{C}^{*} = \arg\max_{\mathbf{C}} \frac{f\Big(\mathbf{W}, \mathbf{C}\Big)}{Cost\Big(\mathbf{C}\Big)} \\ \text{subject to } f\Big(\mathbf{W}, \mathbf{C}\Big) \geq \text{minOps } \& \ Cost\Big(\mathbf{C}\Big) \leq \text{Budget}\end{gathered} \tag{4.2}$$

Here, $f(\mathbf{W}, \mathbf{C})$ is the function that maps the workload vector $\mathbf{W}$ and the configuration vector $\mathbf{C}$ to the throughput (the cluster prediction model) and $\mathbf{C}^{*}$ is the best reconfiguration plan selected by OptimusCloud. The two constraints in the problem prevent us from selecting configurations that exceed the budget or those that deliver unacceptably low performance.

The optimization problem described in Equation 4.2 falls under the category of gradient-free optimization problems [89], in which no gradient information is available nor can any assumption be made regarding the form of the optimized function.

For this category of optimization problems, several meta-heuristic search methods have been proposed, such as, Genetic Algorithms (GA) , Tabu Search [90], and Simulated Annealing. We use GA due to two relevant advantages. First, constraints can be easily included in its objective function (*i.e.* the fitness function). Second, it provides a good balance between exploration and exploitation through crossover and mutation [91]. We use Python `Solid` library for GA [92] and `Scikit-learn` for random forests [93].

Cost-Benefit Analysis

Changing either NoSQL or cloud configurations at runtime has a performance cost due to downtime caused to nodes being reconfigured.

We find that most of the performance-impacting NoSQL parameters (83% for Cassandra) necessitate a server restart and naturally, changing the VM type needs a restart as well.

When a workload change is predicted in the online phase, OptimusCloud uses its performance predictor to propose new configurations for the new workload. Afterward, OptimusCloud estimates the reduction in performance given the expected downtime duration and compares that to the expected benefit of the new configurations. OptimusCloud se-

lects configurations that maximize the difference between the benefit and the cost (both in terms of Throughput/$) .

This cost-benefit analysis prevents OptimusCloud from taking greedy decisions, whenever the workload changes. Rather, it uses a long-horizon prediction of the workload over a time window to decide which reconfiguration actions to instantiate and when.

The benefit of the ith step in the plan is given by:

$$B_{(\mathrm{i}+1,\mathrm{i})} = \sum_{t \in t_{\mathrm{i}+1}} f(W_t, C_{\mathrm{i}+1}) - f(W_t, C_{\mathrm{i}}) \tag{4.3}$$

where $f\Big(W_t, C_{\mathrm{i}+1}\Big)$ is the predicted throughput using the new configuration $C_{\mathrm{i}+1}$. The configuration cost is given by:

$$L_{(\mathrm{i}+1,\mathrm{i})} = \sum_{p \in (C_{\mathrm{i}} - C_{\mathrm{i}+1})} t_{down} \times \delta_p \times f(W_t, C_{\mathrm{i}}) \tag{4.4}$$

where p is any *Complete-Set* that is being reconfigured to move from configuration C_{i} to $C_{\mathrm{i}+1}$, t_{down} is the expected downtime during this reconfiguration step, and δ_p is the portion of the cluster throughput that p contributes as estimated by our cluster predictor. We then normalize the benefit (Equation. 4.3) and the cost (Equation. 4.4) by the difference in price between configurations C_{i} and $C_{\mathrm{i}+1}$. The value of t_{down} is measured empirically and its average value is 30 sec for NoSQL configurations and 90 sec for VM configurations.

4.4.5 Distinctions from Closest Prior Work

We describe the substantive conceptual differences of OptimusCloud from two recent, related works: Selecta and SOPHIA. OptimusCloud provides joint configuration tuning of both NoSQL and cloud VMs, while it considers heterogeneous clusters to achieve the best Perf/$. In Selecta, only heterogeneous cloud storage configurations are permissible (*i.e.* HDD, SSD, or NVMe). Accordingly, the configuration space in Selecta is much smaller and simpler to optimize using matrix factorization techniques. A simple extension of Selecta to our large search space produces very poor performance due to the sparsity of the generated matrix and the dependency between NoSQL and cloud configurations as we empirically show in Sec. 4.5.6.

In SOPHIA, only NoSQL parameters are configured and no computing platform parameters such as VM configurations are optimized. Even within NoSQL configurations, it only considers homogeneous configurations. Accordingly, SOPHIA makes a much simpler decision to either configure the complete cluster to the new configuration, or keep the old configuration—correspondingly its cost-benefit analysis is also coarse-grained, at the level of the entire cluster. For fast-changing workloads, it therefore often has to stick to the current configuration since there is not enough time to reconfigure the entire cluster (which needs to be done in a partly sequential manner to preserve data availability). *Similar to Selecta, a simple extension of SOPHIA to VM options cannot achieve the best Perf/$ for dynamic workloads, as it can only create homogeneous configurations across all phases of the workload.* We empirically show this in Sections 4.5.3 and 4.5.7.

4.5 Experimental Setup and Results

In this section, we evaluate OptimusCloud under different experimental conditions for the 3 applications.

We deploy OptimusCloud and the datastore clusters (Cassandra or Redis) in Amazon EC2 in the US West (Northern California) Region. We also deploy a separate set of nodes in the same region to serve as workload generators (*i.e.* shooters). We vary the number of shooting threads in runtime to simulate the changes in the request rate in the workload trace. The results are averages of 20 runs, with each run using a different subset of the training data. Our evaluation answers four broad questions. (1) How does OptimusCloud compare in terms of Perf/$ and P99 latency with three state-of-the-art systems (which can only create homogeneous configurations) and two oracle-based baselines? (2) What is the accuracy of each module of OptimusCloud, such as, the workload and the performance predictors? (3) How do application requirements such as RF and CL impact OptimusCloud? (4) How does OptimusCloud generalize to different applications (we use three), databases (Cassandra and Redis), and levels of prediction errors?

Major Insights: We draw several key insights from our evaluation.

First, the flexibility afforded by being able to reconfigure different parts of the cluster to different configurations is useful—all three prior protocols being compared (and in fact,

all works to date) can only create homogeneous configurations. *Further, the proactive approach of initiating reconfiguration upon predicted workload change helps to handle dynamic workloads and keeps latency low (CherryPick and Selecta are both reactive).* **Second**, OptimusCloud's design reduces the heterogeneous configurations search space significantly. Accordingly, it is able to search efficiently and finds higher performing VM and NoSQL configurations than cluster configurations selected by CherryPick, Selecta, or SOPHIA. This improvement persists across applications (highest for the more predictable HPC analytics workload and lowest for the MG-RAST workload) (Fig. 4.8, 4.11, 4.13), different (RF,CL) values (larger values of RF and smaller values of CL) (Fig. 4.11), and data volumes (benefit stays unchanged) (Fig. 4.8). **Third**, OptimusCloud achieves comparable or better P99 latency than the baselines, thus showing that it does not sacrifice raw performance in search of the performance per unit cost.

4.5.1 Applications

Here we give the details for our three use case applications, which together span a wide range in terms of predictability and nature of the requests in the workload. **MG-RAST** is a global-scale metagenomics portal [69], the largest of its kind, which allows many users to simultaneously upload their metagenomic data to the repository, apply a pipeline of computationally intensive processes and optionally commit the results back to the repository for shared use. Its workload does not have any discernible daily or weekly pattern, as the requests come from all across the globe and we find that the workload can change drastically over a few minutes.

A total of 80 days of real query trace were analyzed, 60 days for training and 20 days for testing. In production, MG-RAST is executed with the values RF=3, CL=1 and it shows abrupt switches in the Read-Ratio (typically from RR=0 to RR=1) and vice versa. The frequency of these switches are 430/day on median. This presents a challenging use case as only 5 minutes of accurate lookahead is possible.

The second workload is the **Bus-Tracking** application [33] where read, scan, insert, and update operations are submitted to a backend database. The data has strong daily and

Table 4.2. (MC stands for *Markov Chain*). Workload prediction RMSE for MG-RAST and Bus-tracking workloads with different lookahead periods.

MG-RAST			BUS-Tracking		
MC-Order	Lookahead	RMSE	MC-Order	Lookahead	RMSE
First	5m	43.7%	First	15m	6.9%
First	10m	68.7%	**First**	**1h**	**7.4%**
Second	**5m**	**43.4%**	Second	5m	7.12%
Second	10m	68.2%	Second	1h	7.4%

weekly patterns to it. This workload has less frequent switches of 60/day on median. For this workload, 60 days of real query trace were analyzed for the application (40 for training and 20 for testing). The relative proportions of the different kinds of queries are 42.2% updates, 54.8% scans, 2.82% inserts, and 0.18% reads. As shown in Table 4.2, the prediction accuracy for Bus-Tracking workload is much better compared to the MG-RAST workload, as expected due to the more regular patterns, and here we use a longer lookahead period of 1 hour.

The third use case is a queue of **data analytics jobs** such as would be submitted to an HPC computing cluster. Here the workload can be predicted over long time horizons (order of an hour) by observing the jobs in the queue and leveraging the fact that a significant fraction of the job patterns are recurring. Thus, our evaluation cases span the range of patterns and corresponding predictability of the workloads. We simulate a shared queue of batch data analytics jobs.

We modeled the jobs on data analytics requests submitted to a real Microsoft Cosmos cluster [50]. Each job is divided into stages and the workload characteristics of the job change with every stage. The job size is a random variable $\sim$ U(200,100K) operations. The workload switches are 780/day on median, with a level of concurrency of 10 jobs. We achieve accurate prediction over a lookahead duration of 1 hour and we use that for our setting with this use case.

4.5.2 Baselines

We compare OptimusCloud to the following baselines:

1. *Homogeneous-Static*: We use our cluster predictor to select the single best configuration to use for the entire duration of the predicted workload. The entire workload is assumed to be known in advance, making this an impractically optimistic baseline. Nevertheless, it is a measure of how well a *statically determined* homogeneous configuration can perform when powered by a hypothetically perfect workload predictor.
2. *CherryPick*: We use CherryPick's Bayesian Optimization (BO) to find a heterogeneous cloud configuration which maximize our objective metric. When the workload changes, BO collects 20 points and selects the best cloud configuration. This process takes about 3 minutes with parallelization on a 16-core machine. The reconfiguration is done by restarting servers all at once, thus making data unavailable transiently.
3. *Selecta*: We use Selecta's SVD prediction model to select the optimized homogeneous configuration with workload changes. We populate the SVD matrix with the same training data as used for training of OptimusCloud. Further, we give a benefit to Selecta that all the workload characteristics are assumed to be pre-filled in the matrix (or close to it) so that it does not have to execute and profile the workload that arrives at runtime, but can be looked up in the matrix. Both CherryPick and Selecta run reactively with workload changes and neither can change the application configuration. They operate in a greedy manner initiating reconfiguration whenever the workload changes, unlike OptimusCloud that optimizes for the workload over a lookahead time window.

4. *SOPHIA*: We use SOPHIA for homogeneous NoSQL configurations while the VM configurations are fixed to the recommended VM types in Cassandra's and Redis' documentations *i.e.* Compute-Optimize C4.large for Cassandra [94] and Memory-Optimized R4.large for Redis [95].

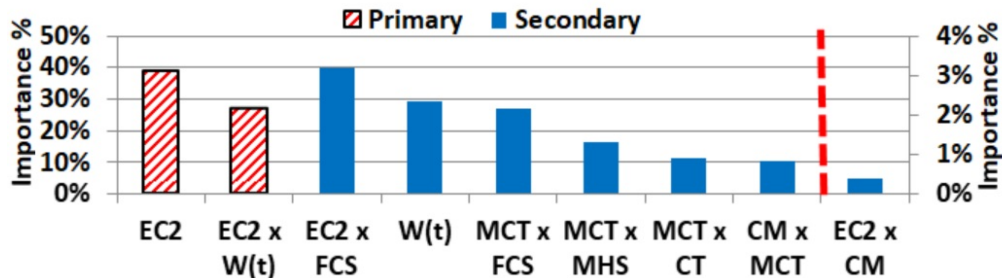


Figure 4.7. Importance of various parameters, including pairwise combinations. Parameters with black solid bars are w.r.t. the right Y-axis. EC2 configuration, the workload, and top 5 Cassandra parameters describe 81% of data variance, after which there is a significant drop in importance, denoted by the red dotted line. Top parameters are: file_cache_size (FCS), memtable_cleanup_threshold (MCT), memtable_heap_space (MHS), compaction_throughput (CT), and compaction_method (CM)

5. *Theoretical-Best*: This is a baseline that knows what is the best-performing configuration for every workload. It then switches the cluster to this configuration without any downtime cost. Though impractical, this baseline provides a quantitative upper bound for any protocol and is used for normalization of our results.

4.5.3 End-to-end System Evaluation

We evaluate how effective OptimusCloud and each of the baselines are in selecting the best reconfiguration plan. In Fig. 4.8 we show the evaluation for the MG-RAST application, the most challenging one for us due to its unpredictable workload characteristics.

We use a 1 hour trace from MG-RAST and apply it to a cluster of 6 or 30 servers We show the performance of the different plans with data volume per server of 16GB and 100GB. The performance of each solution is normalized by that of the Theoretical-Best. OptimusCloud's plan achieves the highest Ops/s/$ and the lowest latency over all baselines. OptimusCloud achieves 86% and 74% improvement over Homogeneous-Static for the 16GB and 100GB cases respectively. This shows there is no single static configuration that can achieve the optimal performance for all phases of the workload. Compared to CherryPick and Selecta, OptimusCloud achieves 87% and 45% improvement on average. This is because both systems are striving to create a homogeneous configuration to meet

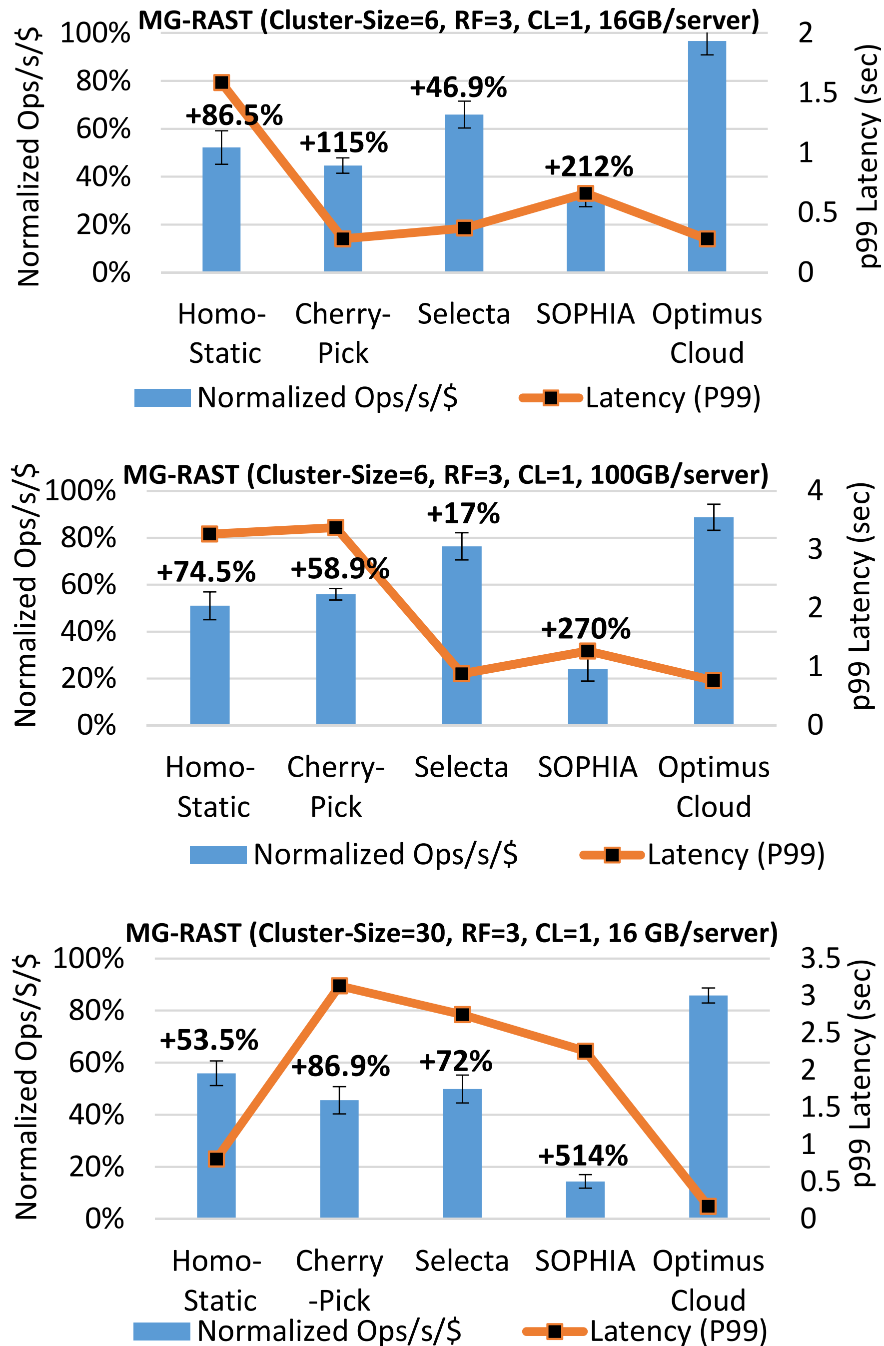


Figure 4.8. Evaluation of MG-RAST traces in Cassandra using OptimusCloud vs state-of-the-art tuning systems. The primary Y-axis represents the ratio of the normalized Ops/s/$ achieved by each system to the theoretical-best performance. OptimusCloud achieves the highest Perf/$ and lowest P99 latency.

the performance requirement and end up increasing the cost. Further, CherryPick incurs the delay of performing the Bayesian optimization, which takes 3 minutes on average and causes the cluster to operate with sub-optimal configurations during this long delay. The aggressive reconfiguration of CherryPick and Selecta (shutting down all servers and restarting all together) causes a significant performance hit (throughput of zero) for the cluster. Compared to SOPHIA, OptimusCloud achieves 212% and 270% improvement in Perf/$. This highlights the benefit of jointly tuning VM and database configurations.

We draw several other conclusions. First, with increasing data volume, the throughput of all protocols goes down because what would once fit in memory (R4.large has 16 GB) now has to go to disk. But the effect on Selecta and CherryPick is smaller because they use expensive and well-resourced memory VMs.

Consequently, the performance benefit of OptimusCloud decreases. Second, Selecta is achieving better performance than CherryPick, which is consistent with the results reported in [73]. This is because of the longer response time of CherryPick's Bayesian Optimization versus Selecta's matrix lookup.

In terms of latency, OptimusCloud scales well (comparing the N=6 to N=30), while Selecta and CherryPick both suffer (latency increases of 7.4× and 11×). This is because the control message traffic is very large in these two baselines as they aggressively shut down and restart all the servers together to achieve a reconfiguration, causing the scalability bottleneck. We also notice that SOPHIA scalability is better than the other baselines as it performs sequential reconfiguration.

OptimusCloud achieves as low tail-latency as Selecta and CherryPick for small scales, and lower at larger scale due to the reason above. Homogeneous-Static has a high latency for all cases due to its inability to adapt to dynamic workloads. The comparison with Selecta and CherryPick shows how important it is to apply *online* reconfiguration to minimize tail-latencies for fast changing workloads.

4.5.4 Sensitive Parameter Identification

We test the feasibility of pruning the joint configuration search space (*i.e.* VM and NoSQL), while maintaining the dependencies among the configuration parameters. We collect a total of 3K data points equally from 15 different VM types. We use D-optimal design to decide which data points to collect for building the performance prediction model. Fig. 4.7 shows the importance of the most impactful parameters, either singly or pairwise, as determined from the regression model. The instance architecture (EC2) and the workload (W(t)) are the most impactful, followed by top-5 database configuration parameters.

Note that the costs of changing different configuration parameters are different as it takes about 90 sec to change EC2 type, whereas changing Cassandra's configuration only requires around 30 sec with no impact on the cluster's $ cost. Expectedly, the instance architecture has high inter-dependency with the NoSQL parameters because the architecture controls the physical resources available to the DBMS.

4.5.5 Single Server Performance Prediction

We evaluate three possible single server prediction models for inclusion in OPTIMUS-CLOUD. In each case, we use a Random Forest using 75%:25% for training and prediction.
1. *N-Solitary-Models*: This builds a separate prediction model per architecture. It predicts the performance of a given architecture/configuration combination using previously collected data points from the same architecture.
2. *Combined-Categorical*: This builds a combined model using all points from all architectures, while it represents the architecture as a categorical parameter (with integral values). Thus, knowledge transfer is limited across architectures.
3. *Combined-Numerical*: This also builds a combined model for all architectures. However, it describes the architecture in terms of its resources *e.g.* C4.large is represented as vCPU 8, RAM 3.75 GB, Network-Bandwidth 0.62 Gbits/s. Thus this allows extrapolating model knowledge across architectures.

We test the accuracy of each predictor using the same number of data points (100 points per architecture) and show the result in terms of R^2 (Table 4.3). We see that using a

Table 4.3. Comparison of different Single-server prediction techniques. OptimusCloud achieves better performance in terms of R^2 and $RMSE$ over all baselines.

Workload	MG-RAST		BUS		HPC	
Metric	R^2	$RMSE$	R^2	$RMSE$	R^2	$RMSE$
N-Solitary -Models	0.2	3401.4	0.127	109.9	0.04	2778
Selecta	-0.14	4149.3	0.66	110.6	0.932	2451
Optimus -Categorical	0.41	1334.2	0.986	21.87	0.983	1172.9
Optimus -Numerical	**0.89**	**1260.9**	**0.988**	**19.77**	**0.986**	**1076.2**

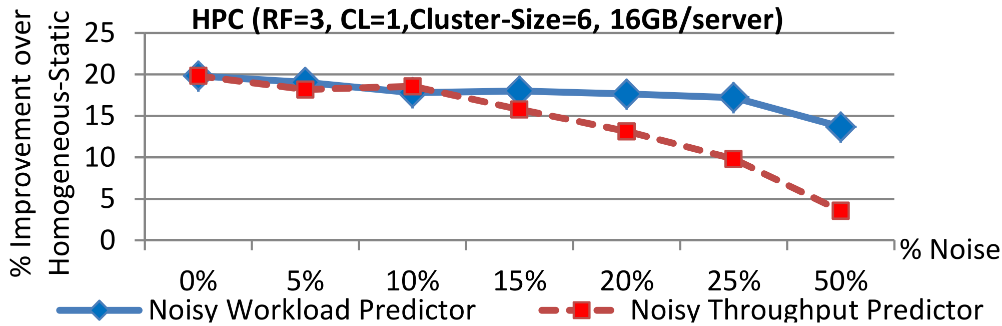


Figure 4.9. Impact of noisy predictions on OptimusCloud's improvement over best Homogeneous-Static configurations

separate model per architecture gives very poor performance due to the lack of knowledge transfer between architectures. Moreover, the numerical representation shows a significant improvement in prediction performance over the categorical representation due to better knowledge transfer across architectures. Thus, we use the *Combined-Numerical* model in OptimusCloud.

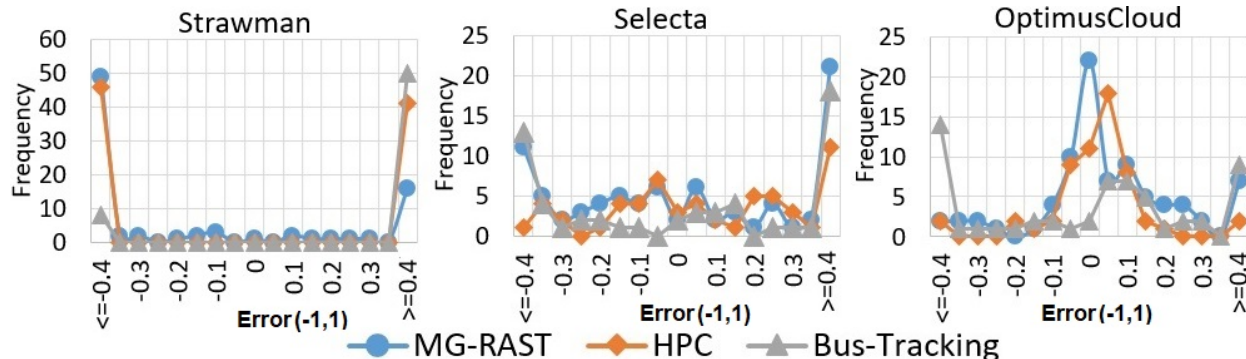


Figure 4.10. Performance prediction error histogram for heterogeneous clusters. We notice that OptimusCloud's error percentage is within -15% to +15% for 70% of the test points, with R^2 value of 0.91 and RMSE of 7.7 KOps/s. On the other hand, the best strawman shows poor performance (RMSE 36 KOps/s) while Selecta performs better (RMSE 21 KOps/s).

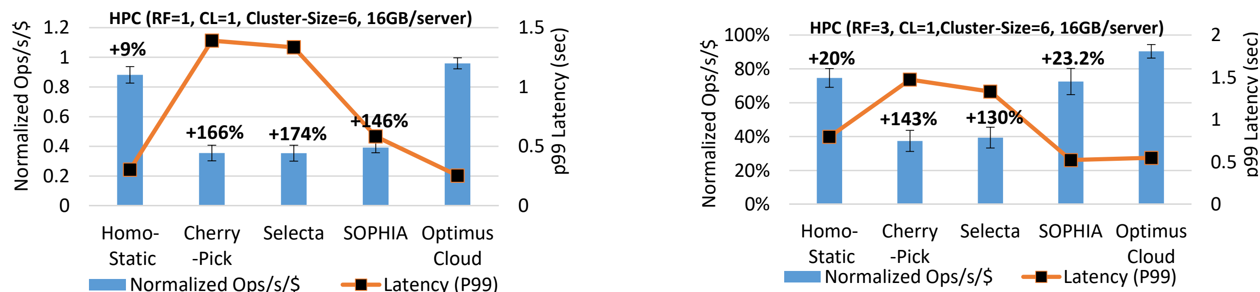


Figure 4.11. HPC workload evaluation with 10 concurrent jobs, and varying (RF,CL) requirements

4.5.6 Cluster Performance Prediction

We evaluate the accuracy of our cluster performance prediction model. We use a cluster of 6 nodes with RF=3, CL=1 and investigate the impact of changing the EC2 architecture of each *Complete-Set*. Thus, the cluster is partitioned into 3 *Complete-Set*s. We use 3 families of EC2's 4$^{\text{th}}$ generation (C4, R4, M4) and three different sizes of each family (large, xlarge, 2xlarge). We collect 330 data points covering all combinations of assigning instance types to these 3 *Complete-Set*s. In Fig. 4.10, we compare our model with a strawman predictor that uses the sum of Ops/s for each individual server as the overall cluster performance (we also tested Average, Min, and Max and got worse performance). This strawman achieves poor

prediction performance with a low R^2 value of 0.08 and an unacceptably high RMSE of 36 KOps/s.

We also compare our model with the latent factor collaborative filtering technique, SVD, used in Selecta [73], which we reimplement using the sci-kit `surprise` library [96]. Selecta achieved better R^2 value of 0.69 and a lower RMSE of 21 KOps/s compared to the strawman predictor. However, our model achieves better performance due to the fact that our Random-Forest model can use non-linear combinations of the elementary features (up to quadratic), while SVD is confined to using linear combinations only. The shapes of the error curves are also different—the Selecta and the strawman curves are bathtub-shaped indicating significant overestimation or underestimation, while the OptimusCloud curve is bell-shaped with a mean close to zero. The bathtub curves are due to the fact that these protocols are ignorant of the token assignment of the cluster and consider erroneously that each node's throughput has the same contribution to the cluster throughput.

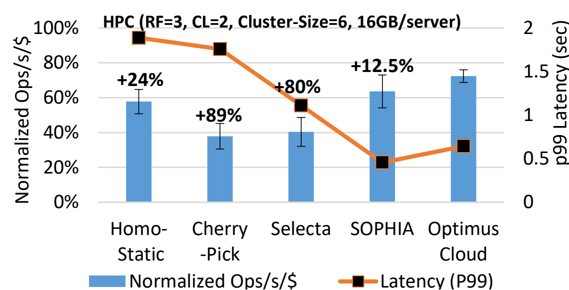


Figure 4.12. HPC workload evaluation with 10 concurrent jobs, and varying (RF,CL) requirements

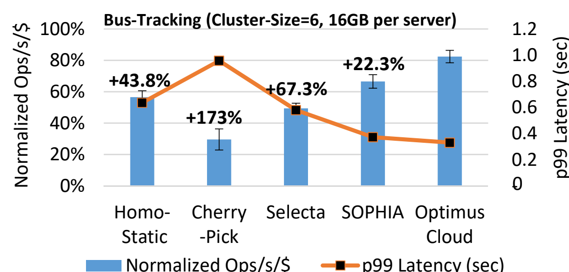


Figure 4.13. Bus-Tracking workload pattern with RF=3, CL=1

4.5.7 Evaluation with Diverse Workloads

Now we evaluate the performance for different workloads, cluster scales, and (RF, CL) requirements.

HPC Workload: Figure 4.11 shows the improvement of OptimusCloud over the baselines for the HPC data analytics traces. We first change RF from 1 to 3, holding CL at 1. Then we change CL from 1 to quorum, which is 2, holding RF at 3. OptimusCloud's plan achieves

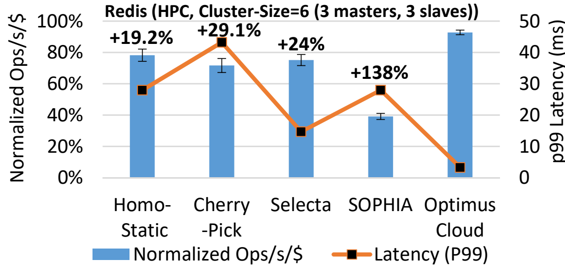


Figure 4.14. Evaluation on Redis for HPC workload with a cluster of 6 servers

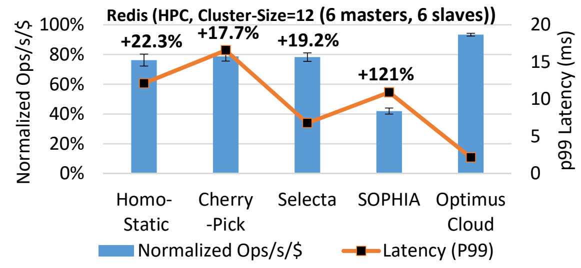


Figure 4.15. Evaluation on Redis for HPC workload with a cluster of 12 servers

the highest Perf/$ and the lowest latency over all baselines for all setups. At RF=3, OptimusCloud achieves 20% and 24% better Perf/$ over Homogeneous-Static configuration for CL=1 and CL=Q respectively. This again shows the importance of dynamic reconfiguration to handle workloads with changing characteristics. In comparison to CherryPick, OptimusCloud achieves 143% and 89% better performance for CL=1 and CL=Q respectively and 130% and 80% over Selecta. Compared to SOPHIA, OptimusCloud achieves 23% and 12.5% better Perf/$. Notice that when RF=1 and CL=1, OptimusCloud can no longer perform non-atomic configurations since the *Complete-Set* in this case is the entire cluster. Thus, its improvement over Homogeneous-Static decreases to 9%.

As RF goes up, the amount of data on each node goes up bringing down the absolute performance. Homogeneous-Static is affected more than OptimusCloud and therefore the benefit of OptimusCloud increases. CherryPick and Selecta are relatively unaffected by increasing RF as they had low throughputs to begin with and they reconfigure all the nodes at once, irrespective of RF and CL. SOPHIA benefits from increasing RF since it can reconfigure more servers concurrently without degrading data availability. As CL goes up, again CherryPick and Selecta are relatively unaffected. The performance of OptimusCloud goes down because the maximum number of *Complete-Sets* that OptimusCloud can reconfigure at a time is inversely proportional to CL. Thus, its benefit over CherryPick and Selecta reduces. However, we note that for most of our target environments, CL is unlikely to be higher than 1 as deployments often favor availability and latency over consistency. In terms of latency, OptimusCloud achieves the lowest P99 latency across all setups.

We also notice that SOPHIA has lower latency than other baselines as it performs a gradual reconfiguration of the different server instances to maintain data availability.

Bus-Tracking Workload: This workload has strong weekly and daily patterns, which allows the workload predictor to provide accurate predictions for long lookahead periods. For a 1 day-trace, the result is shown in Fig. 4.13. OptimusCloud achieves better performance/$ over Homogeneous-Static, CherryPick, Selecta, and SOPHIA by 46%, 178%, 67%, and 28% respectively. As before, OptimusCloud achieves the lowest tail latency across all techniques. The tail latency metric is very important for this workload as it represents a user-facing application. We observe that OptimusCloud achieves higher gains in performance over CherryPick and Selecta compared to the MG-RAST workload, which shows the benefit of longer accurate predictions for OptimusCloud.

4.5.8 Evaluation with Redis

Redis has a very different architecture than Cassandra and is therefore a suitable target to evaluate the generalizability of OptimusCloud. Here we use Redis in clustered mode as a distributed cache (a common use case for Redis)—if the key is found in Redis' memory, it is served by Redis, else, it is served by a slower disk-based database. We apply the HPC analytics workload to a cluster of 6 or 12 Redis servers, keeping the replication degree as 2. We select HPC workload as it has the shortest key-reuse-distance between the three workloads and for which using Redis as a cache is most beneficial [97], [98]. We tune Redis' `maxmemory-policy` and `maxmemory` parameters and observe that changing the cloud configurations for more or less RAM size has an impact on the best value of both parameters. We also found that Redis' Perf/$ is sensitive to workload parameters such as job size, access distribution, and read-to-write. We start by collecting 108 data points for different jobs with varying job sizes, access distributions, and read-write ratios. Job size is a random variable with U(0.5M,1.5M) operations. Access distribution is randomly selected from *Uniform*, *Latest*, and *Zipfian*. Read-write ratio is a random variable with distribution U(0,1). We experiment with traces of 75 jobs which span a total of 5 hours.

Our performance predictor achieves an R^2 of 0.922 averaged over 20 runs. From Fig. 4.14, we see that OptimusCloud achieves a better Perf/$ of 19% (Homogeneous-Static), 29% (CherryPick), 24% (Selecta), and 138% (SOPHIA). Also, OptimusCloud reduces the tail latency by 8.4X (Homogeneous-Static), 13X (CherryPick), 4.4X (Selecta), and 8.1X (SOPHIA). We draw the following insights. First, as SOPHIA uses an expensive cluster of R4.large (as recommended in Redis' documentation [95]), it achieves a very low Perf/$. Second, both CherryPick and Selecta switch from R4.large to the less expensive C4.large and M4.large when the workload changes (and less memory is required) and therefore achieve a higher Perf/$. Finally, by using a heterogeneous cluster of the three VM types as well as jointly tuning the application configuration, we achieve the best Perf/$ and the lowest latency among all techniques. To test scalability, we increase the number of servers to 12 (Fig. 4.15) and note that the normalized performance of each system stays approximately constant.

4.5.9 Tolerance to Prediction Errors

We investigate how tolerant OptimusCloud is to errors in both predictors—performance (throughput) and workload. We add synthetic noise to the output of each predictor separately

and then show how does the benefit of OptimusCloud change with the amount of synthetic noise for the HPC workload.

The percentage of noise is represented as a uniform random variable that is added to (or subtracted from) the output of the predictor.

For performance prediction, the noise is added to the overall throughput/$ predicted by our multi-server model. For workload prediction, the noise is added to the number of requests/sec in addition to the workload change duration. As shown in Fig.4.9, OptimusCloud's improvement over Homogeneous-Static decreases with increasing levels of noise, as the selected configurations deviate from the best configurations.

We note that OptimusCloud is more sensitive to errors in the throughput predictor compared to errors in the workload predictor, which is demonstrated in the steeper downward slope in the noisy throughput predictor curve.

The reason for this high sensitivity is that OptimusCloud uses the throughput predictor to select the best configuration and with increasing levels of noise, the selected configuration more frequently deviates from the optimal. As discussed earlier, a slight deviation from the optimal configuration may cause a significant reduction in Perf/$. On the other hand, slight errors in workload prediction causes OptimusCloud to reconfigure earlier or later than it optimally should. However, this has less impact on performance as long as it still switches to the best configuration.

5. SONIC : APPLICATION-AWARE DATA PASSING FOR CHAINED SERVERLESS APPLICATIONS

5.1 Abstract

Data analytics applications are increasingly leveraging serverless execution environments for their ease-of-use and pay-as-you-go billing. The structure of such applications is usually composed of multiple functions that are chained together to form a workflow. The current approach of exchanging intermediate (ephemeral) data between functions is through a remote storage (such as S3), which introduces significant performance overhead. We compare three data-passing methods, which we call *VM-Storage*, *Direct-Passing*, and state-of-practice *Remote-Storage*. Crucially, we show that no single data-passing method prevails under all scenarios and the optimal choice depends on dynamic factors such as the size of input data, the size of intermediate data, the application's degree of parallelism, and network bandwidth. We propose Sonic, a data-passing manager that optimizes application performance and cost, by transparently selecting the optimal data-passing method for each edge of a serverless workflow DAG and implementing communication-aware function placement. Sonic Sonic monitors application parameters and uses simple regression models to adapt its hybrid data passing accordingly. We integrate Sonic with OpenLambda and evaluate the system on Amazon EC2 with three analytics applications, popular in the serverless environment. Sonic provides lower latency (raw performance) and higher performance/$ across diverse conditions, compared to four baselines: SAND, vanilla OpenLambda, OpenLambda with Pocket, and AWS Lambda.

5.2 Introduction

Serverless computing platforms provide on-demand scalability and fine-grained resource allocation. In this computing model, cloud providers run the servers and manage all administrative tasks (*e.g.* scaling, capacity planning, etc.), while users focus on the application logic. Due to its elasticity and ease-of-use advantages, serverless computing is becoming increasingly popular for advanced workflows such as data processing pipelines [99], [100], machine

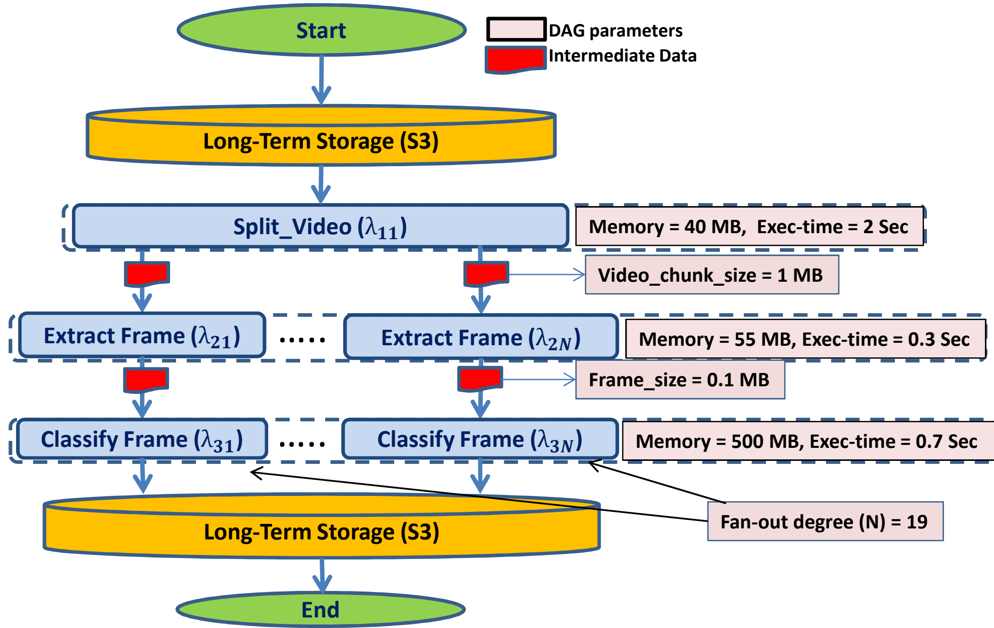


Figure 5.1. DAG overview (DAG definition provided by the user and our profiled DAG parameters) for Video Analytics application.

learning pipelines [101], [102], and video analytics [103]–[105]. Major cloud providers recently introduced serverless workflow services such as AWS Step Functions [106], Azure Durable Functions [107], and Google Cloud Composer [108], which provide easier design and orchestration for serverless workflow applications. Serverless workflows are composed of a sequence of execution stages, which can be represented as a directed acyclic graph (DAG) [99], [109]. DAG nodes correspond to serverless functions (or λs[1]) and edges represent the flow of data between dependent λs (*e.g.* our video analytics application DAG is shown in Fig. 5.1).

Exchanging intermediate data between serverless functions is a major challenge in serverless workflows [110]–[112]. By design, IP addresses and port numbers of individual λs are not exposed to users, making direct point-to-point communication difficult. Moreover, serverless platforms provide no guarantees for the overlap in time between the executions of the parent (sending) and child (receiving) functions. Hence, the state-of-practice technique for data passing between serverless functions is saving and loading data through remote storage (*e.g.* S3). Although passing intermediate data through remote storage has the benefit of

[1]↑For simplicity, we denote a serverless function as lambda or λ.

cleanly separating compute and storage resources, it adds significant performance overheads, especially for data-intensive applications [111]. For example, Pu [99] show that running the CloudSort benchmark with 100TB of data on AWS Lambda with S3 remote storage can be up to 500× slower than running on a cluster of VMs. Our own experiment with a machine learning pipeline that has a simple linear workflow shows that passing data through remote storage takes over 75% of the computation time (Fig. 5.2, fanout = 1). Previous approaches reduce this overhead by implementing exchange operators optimized for object storage [111], [113], replacing disk-based object storage (*e.g.* S3) with memory-based storage (*e.g.* ElastiCache Redis), or combining different storage media (*e.g.* DRAM, SSD, NVMe) to match application needs [99], [114], [115]. However, these approaches still require passing data over the network multiple times, adding latency.

Moreover, in-memory storage services are much more expensive than disk-based storage (*e.g.* ElastiCache Redis costs 700× more than S3 per GB [99]).

We show how cloud providers can optimize data exchange between chained functions in a serverless DAG workflow with communication-aware placement of lambdas. For instance, the cloud provider can leverage data locality by scheduling the sending and receiving functions on the same VM, while preserving local disk state between the two invocations. This data passing mechanism, which we refer to as *VM-Storage*, minimizes data exchange latency but imposes constraints on where the lambdas can be scheduled. Alternatively, the cloud provider can enable data exchange between functions on different VMs by directly copying intermediate data between the VMs that host the sending and receiving lambdas. This data passing mechanism, which we refer to as *Direct-Passing*, requires one copy of the intermediate data elements, serving as a middle ground between *VM-Storage* (which requires no data copying) and *Remote storage* (which requires two copies of the data). Each data passing mechanism provides a trade-off between latency, cost, scalability, and scheduling flexibility. Crucially, we find that no single mechanism prevails across all serverless applications, which have different data dependencies. For example, while *Direct-Passing* does not impose strict scheduling constraints, scalability can become an issue when a large number of receiving functions copy data simultaneously and saturate the VM's outgoing network bandwidth.

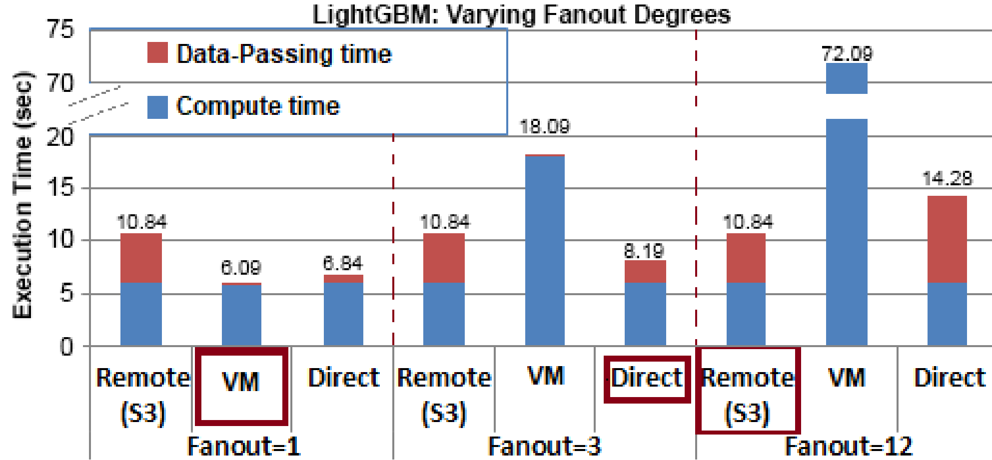


Figure 5.2. Execution time comparison with *Remote storage*, *VM storage*, and *Direct-Passing* for the LightGBM application with Fanout = 1, 3, 12. The best data passing method differs in every case.

Hence, we need a hybrid, fine-grained data passing approach that optimizes data passing for every edge of an application DAG.

Our solution: We propose SONIC, a management layer for inter-lambda data exchange, which adopts a hybrid approach of the three data passing methods (*VM-Storage*, *Direct-Passing* and *Remote storage*) to optimize application performance and cost. SONIC exposes a unified API to application developers which selects the optimal data passing method for every edge in an application DAG to minimize data passing latency and cost. We show that this selection depends on parameters such as the size of input, the application's degree of parallelism, and VM network bandwidth. SONIC adapts its decision dynamically as these parameters change. Since locally optimizing data passing decisions at given stages in a DAG can be globally sub-optimal, SONIC applies a Viterbi-based algorithm to optimize latency and cost across the entire DAG. Fig. 5.3 shows the workflow of SONIC. SONIC abstracts its hybrid data passing selection and provides users with a simple file-based API, so users always read and write data as files to storage that appears local.

SONIC is designed to be integrated with a cluster resource manager (*e.g.* Protean [116]) that assigns VM requests to the physical hardware and optimizes provider-centric metrics such as resource utilization and load balancing (Fig 5.4).

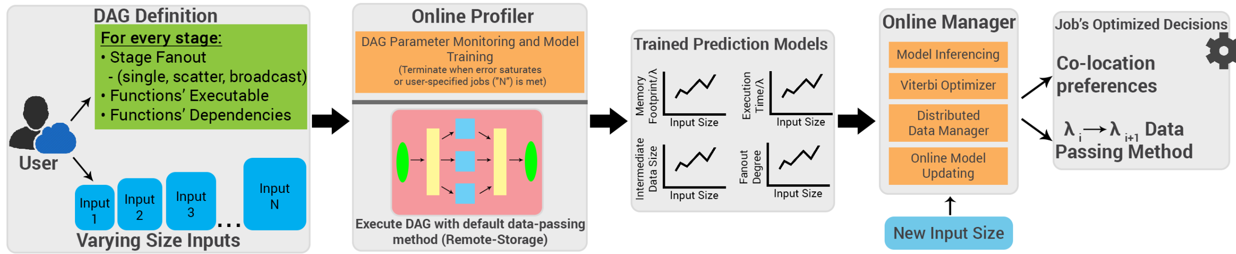


Figure 5.3. Workflow of SONIC: Users provide a DAG definition and input data files for profiling. SONIC's online profiler executes the DAG and generates regression models that map the input size to the DAG's parameters. For every new input, the online manager uses the regression models to identify the best placement for every λ and best data passing method for every pair of sending\receiving λs in the DAG

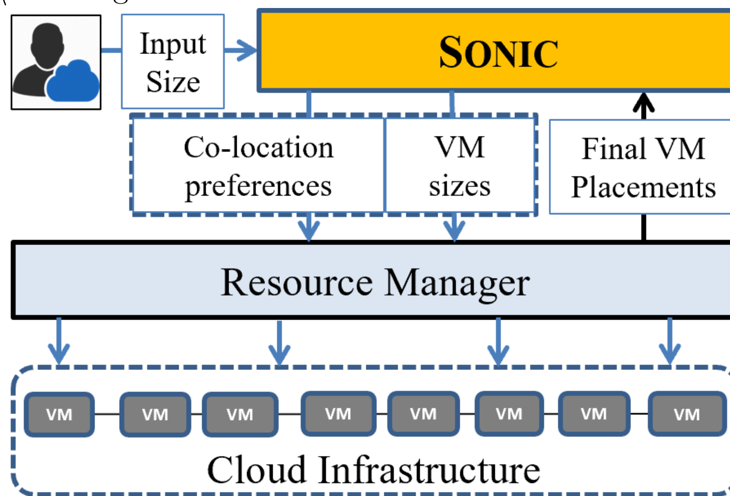


Figure 5.4. SONIC's interaction with an existing Resource Manager in the system.

We integrate SONIC with the open-source OpenLambda [117] framework and compare performance to several baselines: AWS-Lambda, with S3 and ElastiCache-Redis (which can be taken to represent state-of-the-practice); SAND [118]; and OpenLambda with S3 and Pocket [114]. Our evaluation shows that SONIC outperforms all baselines for a variety of analytics applications. SONIC achieves between 34% and 158% higher performance/$ (here performance is the inverse of latency) over OpenLambda+S3, between 59% and 2.3X over OpenLambda+Pocket, and between 1.9× and 5.6× over SAND, a serverless platform that leverages data locality to minimize execution time.

In summary, our contributions are as follows:

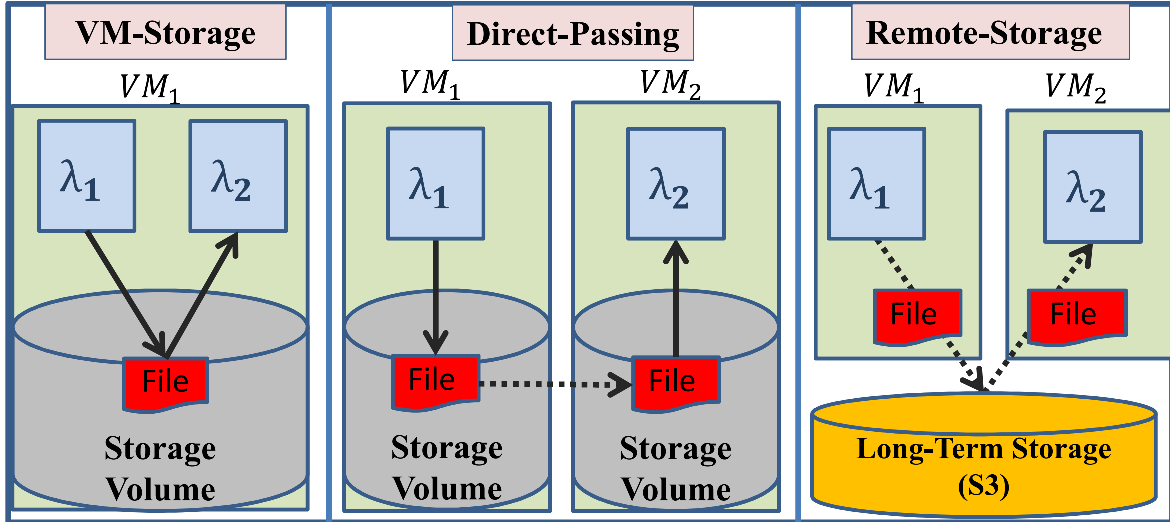


Figure 5.5. The three data passing options between two lambdas (λ_1 and λ_2). *VM-Storage* forces λ_2 to run on the same VM as λ_1 and avoids copying data. *Direct-Passing* stores the output file of λ_1 in the source VM's storage, and then copies the file directly to the receiving VM's storage. *Remote storage* uploads and downloads data through a remote storage (*e.g.* S3).

(1) We analyze the trade-offs of three different intermediate data passing methods (Fig. 5.5) in serverless workflows and show that no single method prevails under all conditions in both latency and cost. This motivates the need for our hybrid and dynamic approach.

(2) We propose SONIC, which automatically selects data passing methods between any two serverless functions in a workflow to minimize latency and cost. SONIC dynamically adapts to application changes.

(3) We evaluate SONIC's sensitivity to serverless-specific challenges such as the cold-start problem (set-up time for the application's environment when it is invoked for the first time), the lack of point-to-point communication, and the provider's lack of knowledge of lambda input data content. SONIC shows its benefit over all baselines with three common classes of serverless applications, with different input sizes and user-specified parameters.

5.3 Rationale and Overview

We discuss the trade-offs of SONIC's three data passing methods and motivate our hybrid and dynamic approach.

5.3.1 data passing Methods

from among the three data passing options shown in Fig. 5.5 for *each* edge in the application DAG.

VM-Storage: This method saves the local state of the sending λ in the VM's storage and schedules the receiving λ(s) to execute on the same VM. This method leverages data locality to minimize latency, but imposes scheduling constraints. If the sending VM's memory cannot fit all receiving λs, this method forces the scheduler to run the receiving λs serially or in batches, sacrificing parallelism in favor of data locality. This method is infeasible if the memory requirement of a single receiving λ exceeds the VM's capacity, or when the receiving λ collects data from multiple λs running on different VMs. Moreover, this data passing method may not be preferred by the resource manager in high load scenarios, where spreading the receiving functions over many servers is needed to avoid hot-spots and achieve better load balancing.

Direct-Passing: This data passing method saves the output of the sending λ on its VM storage and sends the λ's access information (IP address and File Path) to SONIC's metadata manager. When one of the receiving λs is scheduled to execute, the metadata manager uses the saved access information to copy the data file *directly* to the destination VM with the receiving λ. *Direct-Passing* allows higher degrees of parallelism and poses no restrictions on λ placements compared to *VM-Storage*, but requires data to be sent over the network between source and destination VMs. ***Remote-Storage:*** This data passing method involves uploading output files to a remote storage system and downloading them at destination λ(s). This is the state-of-practice in commercial serverless platforms and has been optimized in several recent papers [99], [114], [115]. This method provides high scalability with no restrictions on λ placement. It also has the advantage of almost uniform data passing time with increasing fanout degrees as shown in Fig. 5.6 due to the high bandwidth of the storage layer. The disadvantage of *Remote-Storage* is having two serial data copies in the critical path — one from the source lambda to the remote storage and one from the remote storage to the destination lambda.

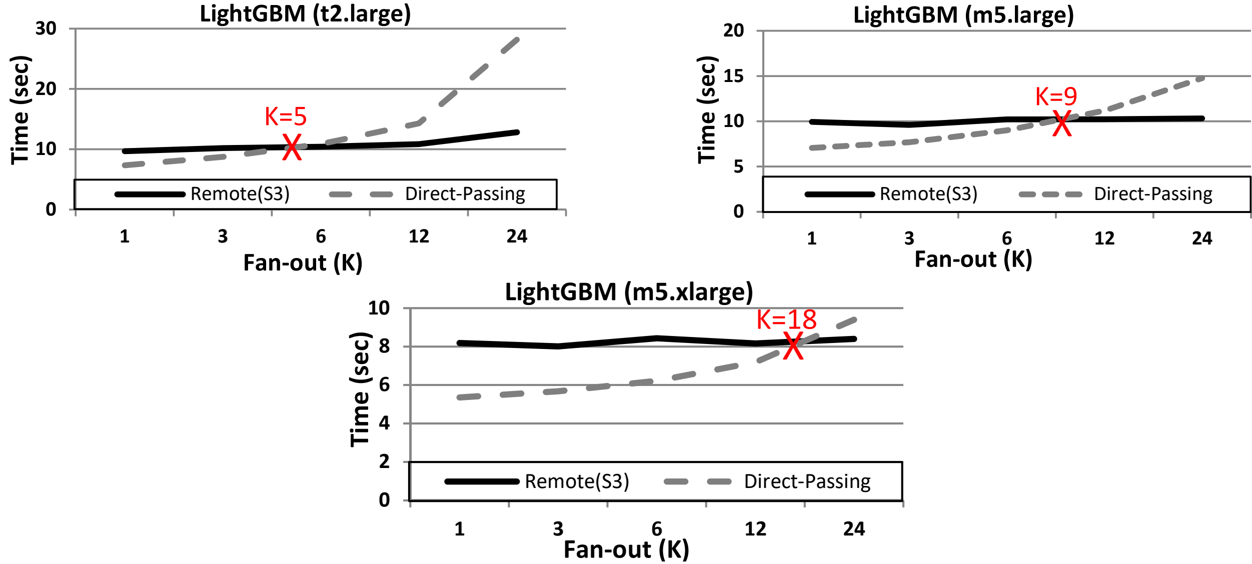


Figure 5.6. Comparison between *Remote-Storage* and *Direct-Passing* for LightGBM workload with varying fanout degrees. Typically, beyond a certain fanout, Remote has lower execution time than Direct-Passing. A more well-provisioned VM (m5.xlarge) will shift the crossover point to the right.

5.3.2 Dynamic Data Passing Method Selection

The optimal choice of data passing method for a job depends on the DAG's parameters, which can vary due to dynamic conditions such as the input size, changes in network bandwidth, or changes in user-specified parameters (*e.g.* degree of parallelism). For example, we run the LightGBM application (application details are given in §5.6.3) with varying degrees of fanout and find that *VM-Storage* achieves optimal end-to-end execution time when the fanout is low (Fig. 5.2). However, when the maximum degree of parallelism across all stages is three, *VM-Storage* requires executing functions serially to fit on the same VM, making *Direct-Passing* superior. With a sufficiently high fanout, the sending VM faces a network bottleneck, making *Remote-Storage* the optimal data passing mechanism. Hence, no single data passing method prevails under all conditions.

Sonic prefers the *VM-Storage* method in cases where the receiving λ(s) can be scheduled on the same VM as the sending λ while executing in parallel. However, in cases where *VM-Storage* is infeasible or sacrifices parallelism, Sonic selects between *Direct-Passing* or *Remote-Storage* methods. This selection depends on two factors: (1) VM's network bandwidth (2) the fanout (*i.e.* parallelism) type and degree. To understand how these two

parameters impact the selection between *Direct-Passing* and *Remote-Storage*, we show an experiment on the LightGBM application, which has a Broadcast Fanout stage (identical data from the sending λ is sent to all the receiving λs). We vary the fanout degree (K) while using different VM types with varying network bandwidths in Fig. 5.6. With low degrees of parallelism (*i.e.* low values of K), *Direct-Passing* achieves lower latency than *Remote-Storage*. This is because copying the data directly across the two VMs is faster than copying the data twice over the network, to and from the remote storage. However, with increasing values of K, *Direct-Passing* suffers from the limited network bandwidth of the VM of the sending λ, which becomes the bottleneck as it tries to copy the intermediate data to K VMs simultaneously.

In contrast, *Remote-Storage* can provide faster data passing in this case, since the sending λ saves its output file once to the remote storage and then every receiving λ downloads that file simultaneously. Thus, *Remote-Storage* provides better scalability over *Direct-Passing* in cases of large fanout degrees. The cross-over point shifts to the right as we go to more well-resourced VMs, from a network bandwidth standpoint (m5.xlarge > m5.large > t2.large).

From this example, we see that selecting the best data passing method depends on the VM's network bandwidth *and* DAG parameters.

We identify a number of trade-offs to be considered when performing this optimization. *First*, a trade-off between data locality and parallelism arises when the available VM's compute and memory capacities are not sufficient to execute all receiving λ(s) in parallel. This preference of data locality over parallelism can be beneficial for lightweight functions communicating large volumes of data, while it can be harmful for compute-heavy functions with small volumes of data. *Second*, executing functions serially has the benefit of avoiding cold-start executions, which can significantly increase the execution time for functions that need to fetch many dependencies before execution [119].

Finally, we differentiate between two types of fanout stages: Scatter and Broadcast, depending on whether the output data is split equally among all outgoing edges (Scatter) or the same data is sent on all outgoing edges (Broadcast). With Scatter fanout, the intermediate data volume being sent is constant with the fanout degree while with Broadcast fanout, the intermediate data volume being sent increases linearly with the fanout degree.

Therefore, the optimal data passing method also depends on the type of fanout and thus we cover both in our evaluation applications (video analytics and MapReduce sort for Scatter fanout and LightGBM for Broadcast fanout). In the next section, we describe the design of each component in SONIC.

5.4 Design

In § 5.4.1, we provide an overview of SONIC's usage model. § 5.4.2 describes SONIC's online profiling. § 5.4.3 discusses how SONIC estimates a job's execution time for different input data sizes. § 5.4.4 shows how SONIC selects globally optimized data passing decisions. Finally, § 5.4.5 highlights further design considerations.

5.4.1 Usage Model

Candidate for shortening — this section and the next combined into one called "Usage model" — one for the end user, one for the cloud provider.

We discuss SONIC's usage model from the perspective of application developers and the cloud provider's resource manager.

Application DAG: As shown in Fig. 5.3, application developers provide SONIC with an application represented as a DAG. We assume users execute application DAGs multiple times with potentially different input data. Each such execution instance is referred to as a *job*. SONIC does not require the FaaS provider to have access to the source code for the serverless functions or hints about the resource utilization of these functions. The DAG definition from the user includes: (1) an executable for every λ; (2) the dependencies between λs; (3) the fanout type in every stage (*i.e.* single, scatter, or broadcast). Users can also provide an upper limit on the execution time or $ budget for a single job or a batch of jobs. Notice, SONIC does not require users to specify the memory requirements for the functions in the DAG (this is a well-known pain point for users of serverless frameworks [120]–[122]). This is because SONIC predicts the memory footprint for every function, using our simple regression modeling (§ 5.4.2) and selects the right host VM size accordingly.

Data Passing Interface: SONIC abstracts the selection of data passing methods from application developers. λ functions write intermediate data to files using a standard file API (read and write), like writing to local storage. All λs within a job share a file namespace and if an application DAG has an edge $\lambda_s \rightarrow \lambda_r$, SONIC ensures that λ_r reads from the same file path that λ_s wrote to. It also ensures that all of a λ_r's input files are present in its local storage before it starts execution.

Resource Manager: SONIC's target is to minimize communication latency and cost, which are *user-centric* metrics. However, optimizing *provider-centric* metrics (*e.g.* load-balancing, resource utilization, and fairness) is also important. Fortunately, many resource management systems such as Graphene [123], Protean [116], AlloX [124] and DRF [125] are designed to optimize *provider-centric* metrics efficiently, while respecting the dependencies between the functions (*i.e.* parent-child execution order). Hence, SONIC is designed to integrate with a system from this category (which we refer to as *Resource Manager*), as shown in Figure 5.4. First, the user executes the DAG with a new input. Second, SONIC uses the input size and DAG definition information to predict the memory footprints, execution times, and intermediate-data sizes for the DAG, which are key DAG parameters in selecting the best VM size for every function and the best data passing method for every edge in the DAG. Finally, the Resource Manager uses SONIC's hints and decides which VMs to allocate on the available physical machines, and responds to SONIC with the final placement information, *i.e.* which functions go on which VMs. The Resource Manager may override SONIC's hint to use *VM-Storage* whenever its scheduling constraint is not acceptable to the manager. Therefore, for that selection, SONIC always proposes the second-best option.

5.4.2 Online Profiling and Model Training

SONIC profiles jobs *online* to determine the impact of changes in input size to the following parameters: (1) each λ's memory footprint, (2) each λ's compute time, (3) intermediate data volume between any two communicating λs, and (4)the fanout type and degree in every stage. Since the above parameters are *not* dependent on the data passing method,

initially SONIC uses a default data passing policy (*Remote Storage*) while it collects DAG parameters for the first N runs of the job to train its models. N is either the number of jobs needed to reach convergence (default) or explicitly set by the user. Next, SONIC trains a set of prediction models that estimate the DAG parameter values for new inputs. SONIC develops polynomial regression models and splits the data collected into training and validation sets, then performs 5-fold cross validation to find the best model to avoid overfitting. We use polynomial regression models as they can learn from limited data points, are lightweight, and are interpretable [126].

”Online” profiling means that SONIC serves workloads while the models are being trained, which is important for practical adoption in production environments. In discussions with commercial cloud providers, we have repeatedly sensed an anathema to solutions that require offline training

due to the concern that one will constantly be taking the system offline to (re-)train the models.

However, if the system owner has ready access to representative input traces, she can feed SONIC with these representative traces *offline* to initialize the prediction models and reduce the online training burden.

SONIC measures the compute time for every λ under two conditions: cold execution (*i.e.* a new VM/container needs to be created) and hot execution (*i.e.* a warm VM/container with pre-loaded models/dependencies already exists).

SONIC uses the difference between the two execution times in deciding whether to queue λs on the same VM and sacrifice parallelism (hot execute), or to execute the λs on different VMs in parallel with the additional data passing cost and startup latency (cold execution).

5.4.3 Minimizing End-to-End Execution Time

We estimate the execution time τ of a stage S_{i} as follows:

$$\tau(S_{\mathrm{i}}) = DataPass(S_{\mathrm{i}-1}, S_{\mathrm{i}}) + Compute(S_{\mathrm{i}}) \tag{5.1}$$

When *VM-Storage* is selected, all λs in a given stage are forced to run on the same VM. If only one λ can run at a time, the first λ incurs a cold execution time and all subsequent λs experience hot executions.

SONIC estimates τ_{VM} as follows:

$$\tau_{VM}(S_{\text{i}}) = 0 + Cold(\lambda_{\text{i},1}) + \sum_{\text{j}=2}^{K} Hot(\lambda_{\text{i},\text{j}}) \tag{5.2}$$

Here we set $DataPass(S_{\text{i}-1}, S_{\text{i}})$ to zero since no additional latency is incurred by *VM-Storage* passing. For simplicity, we give here the upper bound, if all the λs are serialized. In practice, we estimate based on batches that are serialized and all λs within a batch can run concurrently.

For *Direct-Passing* and *Remote Storage*, we take the nature of the fanout type into account. For *Direct-Passing*, with Broadcast-type fanout, the runtime is given by:

$$\tau_{Direct_B}(S_{\text{i}}) = \frac{F \times K}{min(BW(VM_{\text{i}-1}), BW(VM_{\text{i}}))} + Cold(\lambda_{\text{i},1}) \tag{5.3}$$

Where F is the intermediate data size, K the fanout, and $BW(VM_{\text{i}})$, the bandwidth of the VM type hosting λs in the i-th stage. Notice that the bandwidth is limited by the slowest of the sending and receiving VMs and that all execution times are *cold*, yet only one execution is accounted for, since they all run in parallel. For Scatter-type fanout, the equation becomes:

$$\begin{aligned}\tau_{Direct_S}(S_{\text{i}}) &= \frac{F/K}{min(BW(VM_{\text{i}-1})/K, BW(VM_{\text{i}}))} + Cold(\lambda_{\text{i},1}) \\ &= \frac{F}{BW(VM_{\text{i}-1})} + Cold(\lambda_{\text{i},1}) \end{aligned} \tag{5.4}$$

Often cloud providers overprovision network bandwidth [127], hence we assume location of the source and destination VMs (intra- vs. inter-rack) does not affect the network bandwidth. Also, we find that the bandwidths are symmetric between VMs for all VM types; however, for remote storage, writing was faster than reading. Next, we show how we use the previous equations for our optimization.

5.4.4 Online VM and Data Passing Selection

A major challenge to optimize the Perf/$ for the entire DAG is to do local selections for *each* stage to achieve the global optimum. Consider Fig. 5.1 for example: if we used *VM-Storage* passing between `Split_Video` and `Extract Frame`, all the extracted frames will reside in a single VM. Selecting *VM-Storage* between `Extract Frame` and `Classify Frame` will force `Classify Frame` to execute serially since that single VM does not have enough memory. However, if we select *Direct-Passing* or *Remote Storage* passing between `Split_Video` and `Extract Frame`, every extracted frame will now reside in a separate VM. Therefore, we can use *VM-Storage* between `Extract Frame` and `Classify Frame` without sacrificing parallelism, lowering the DAG's execution time. Thus, greedily optimized decisions for individual stages can lead to sub-optimal global DAG performance.

To overcome the issue of sub-optimal greedy decisions, we apply the Viterbi algorithm [128] to find the globally optimized solution. SONIC uses a recursive scoring algorithm to generate all possible lambda assignments in every stage in the DAG based on the VM's compute and memory capacities, along with the λs' predicted memory footprint.

SONIC explores all possible λ-placement options under the constraints of VM resources (CPU and memory primarily) and selects the best data passing method for every pair of stages. Then, SONIC constructs a dynamic programming table with all the generated solutions. Finally, SONIC applies the Viterbi Algorithm to find the best sequence of options (*i.e.* the optimum Viterbi path) in the dynamic table. The selected solution is the one with the best Perf/$ that also meets the user bounds on execution time and cost budget. This approach relies on the fact that the execution time up until stage i is equal to the execution time to stage $\mathrm{i}-1$ + data passing time between the two stages. This makes the Markovian assumption true (next state depends only on the current state) and makes the computation tractable.

We choose the Viterbi algorithm as it is guaranteed to find the true maximum a posteriori (MAP) solution [128], [129] unlike heuristic-based searching algorithms such as Genetic Algorithms or Simulated Annealing. The runtime complexity of the algorithm is $O(P^2 \times S)$, where P is the number of feasible λ placements on VMs for a given stage and S is the number

of stages. P is upper bounded by the degree of parallelism of the stage (*e.g.* AWS sets a limit of 1,000) and in practice is much smaller considering many co-locations on VMs are infeasible. The runtime increases with the number of stages in the DAG, not the number of nodes or edges or fanout degree. This is desirable as the number of stages is small in practice, where a DAG of 8 stages is considered long for current serverless applications [99].

This reduction in complexity happens because Sonic applies the *same* data passing method to *all* the functions in a given stage.

5.4.5 Further Design Considerations

Fault tolerance.

Most FaaS providers apply an automatic retry mechanism upon execution failure (*e.g.* AWS Lambda, Google Functions, and Azure Functions) to ensure that functions are executed *at-least-once*. For this retry mechanism to be successful, functions are required to be idempotent.

To achieve idempotence, Sonic's file API assigns an ID to every intermediate file in the DAG that is used by the function that writes that file. Hence, a re-execution of this function simply overwrites the files from the previous execution.

However, as highlighted in RAMP [130] and AFT [131], idempotence in itself is not sufficient to achieve fault tolerance in serverless environments, since it does not guarantee *atomic visibility*. The problem happens when a sender λ generates only a subset of its output files, and then fails, triggering an incomplete subset of receiving λs, resulting in a corrupted state. Therefore, Sonic applies the concept of *atomic visibility* by delaying the execution of *all* receiving λs that belong to the same logical request until *all* their input files are successfully written to storage (either EBS in case of *Local-VM or Direct-Passing*, or S3 in case of *Remote Storage*). Although this delaying mechanism can potentially increase the E2E latency of the DAG, our evaluation shows that this additional latency is negligible: 6.3% for MapReduce, 3.3% for Video-Analytics, and 0.5% for LightGBM. All the Sonic results in the evaluation are with atomicity enabled.

DAGs with Content-Sensitive Structures. Our design assumes that the stages in the DAG are static and known, while the fanout degree in every stage can vary based on the input. This is analogous to state machines created by serverless orchestrators such as AWS Step-Functions [106]. Our video-analytics application is an example of an input-dependent fanout degree DAG and SONIC successfully predicts the fanout for new input sizes. If the structure of the DAG's stages depends on the input content, then our estimate of the DAG will be inaccurate. We evaluate the sensitivity of Sonic to prediction errors in Sec. 5.6.6.

Scalability of SONIC. For scalability, SONIC adopts a simple distributed design in which no state is shared across application jobs. When a new job arrives, SONIC's centralized component makes the decision whether to use an existing instance (if it is not overloaded) or to spin up a new instance to handle the new job. There is a central scoreboard maintained to keep track of the number of jobs being handled by each instance. The central component is lightweight in terms of its computational load and state. However, should it be necessary, this itself can be distributed through standard state machine replication (SMR) strategies [132].

5.5 Implementation

We implement SONIC as a data passing management layer and we use OpenLambda as our serverless platform [117]. OpenLambda is an open-source platform that relies on Linux containers for isolation and orchestration [133]. We choose OpenLambda for its flexibility—SONIC needs to control the lambda placement and needs IP addresses and administrative access to the hosting VMs. This level of control is infeasible to achieve on AWS Lambda or any other commercial offering. We implement SONIC in C# (482 LOC) and deploy it on EC2 instances, providing the same isolation guarantees as AWS Lambda across users [134].

In our setup of SONIC on OpenLambda, we use one separate container for each function (OpenLambda's design), while one VM can host multiple containers for the same application.

SONIC's data manager consists of two parts: a centralized manager that stores IP addresses and file paths, and a distributed manager, deployed in every VM, executing the SONIC-optimized data passing method. The distributed manager also measures the actual DAG parameters during the online phase and sends them to the online manager, which

updates the regression models in an incremental fashion. Moreover, since the network bandwidth can vary over time, we monitor it using Cloudwatch [135] (default of every 5 min). We use a weighted average of historic measures (as is common [136], [137]) when estimating data passing times. Thus Sonic adjusts its decisions based on bandwidth fluctuations. We use EBS storage as our storage for *VM-Storage* and *Direct-Passing* methods. EC2 EBS-optimized instances (*e.g.* our choice m5) have dedicated bandwidth for EBS I/O (minimizes network contention with other traffic), and can be rightsized for the predicted intermediate data volume.

5.6 Evaluation

5.6.1 Performance Metrics

Below text can be chopped/rewarded for space, since we already have Section 2.3 defending why we use these metrics. Moved from before and merged. Our primary performance metric is Perf/$. Since we are minimizing end-to-end (E2E) execution time (i.e., latency), this is given by: $\frac{1}{Latency(sec)} \times \frac{1}{Price(\$)}$. We also use raw latency as a secondary metric, to separate the $-cost normalization effect. For the first metric, higher is better, and for the second, lower is better. When we refer simply to performance, we mean Perf/$. When we refer to the secondary metric, we explicitly say (E2E) execution time or latency. Review E & D: I'm a bit surprised that Perf/$ is the chosen performance metric, given that the paper was motivated exclusively by performance.

The different data passing methods considered by Sonic vary in their billing cost, which is sensitive to the selected configurations for each method. *Hence, our solution should consider both latency **and** cost when selecting the best data passing method.* Consequently, we use the Perf/$ metric. We also empirically demonstrate that Sonic performs comparably to the baselines in terms of raw performance, and in many cases, outperforms the baselines (Figures 5.7, 5.8, 5.13, 5.14). Finally, there is precedence of prior work in cloud optimization using performance normalized by price [138]–[140]. Review A: Is the EBS cost, which is larger than S3, taken into account for SONIC? Sonic's *VM-Storage* and *Direct-Passing* methods add additional cost due to the extra local storage (*e.g.* EBS storage). However,

this additional price is comparable to that of remote storage such as S3, and we include it in our evaluation.

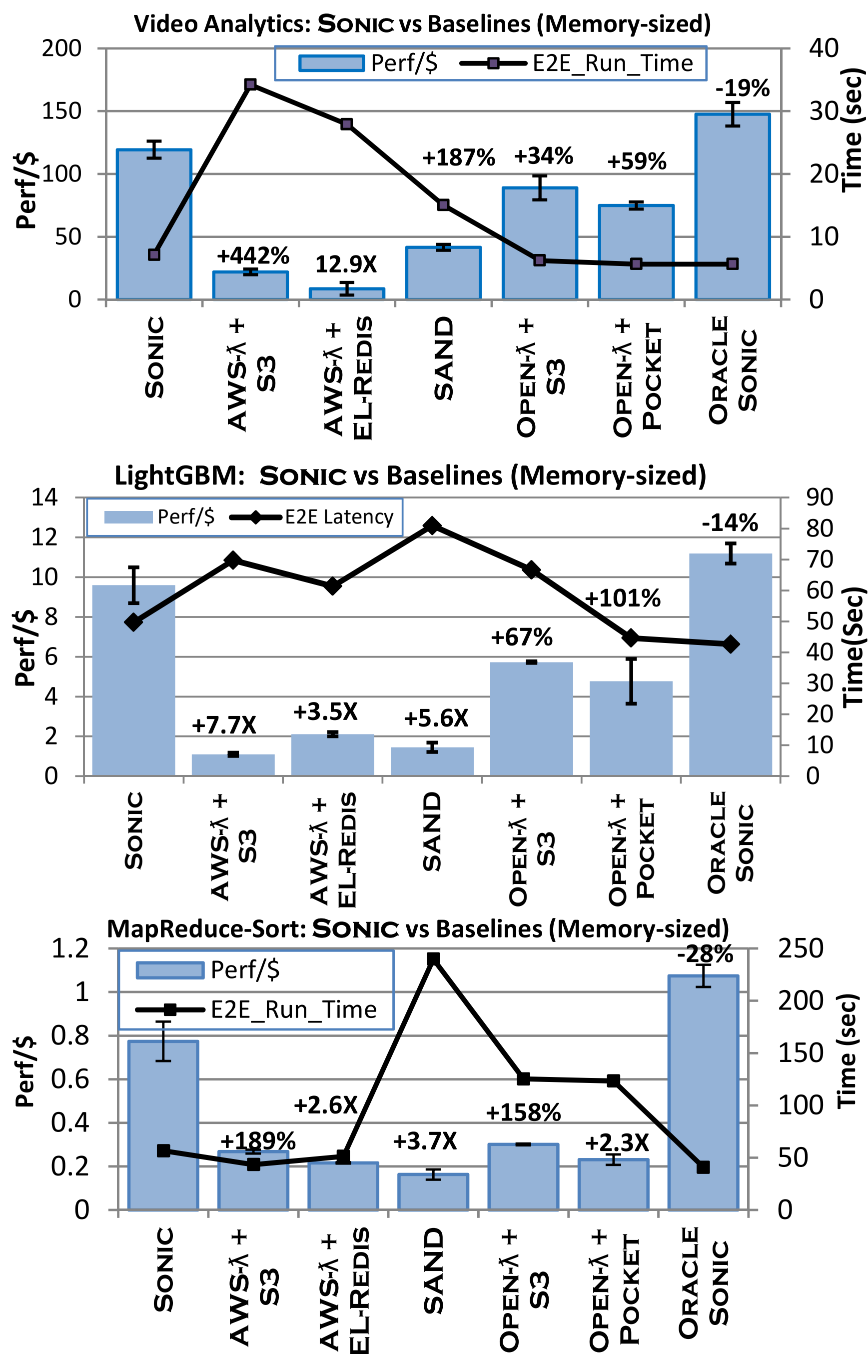


Figure 5.7. Performance of SONIC and the baselines for our three applications (memory-sized). Two performance metrics are shown: Perf/$ (bars; left axis) and end-to-end execution time (lines; right axis). Relative improvements in Perf/$ due to SONIC are at the top of each bar for the corresponding baseline.

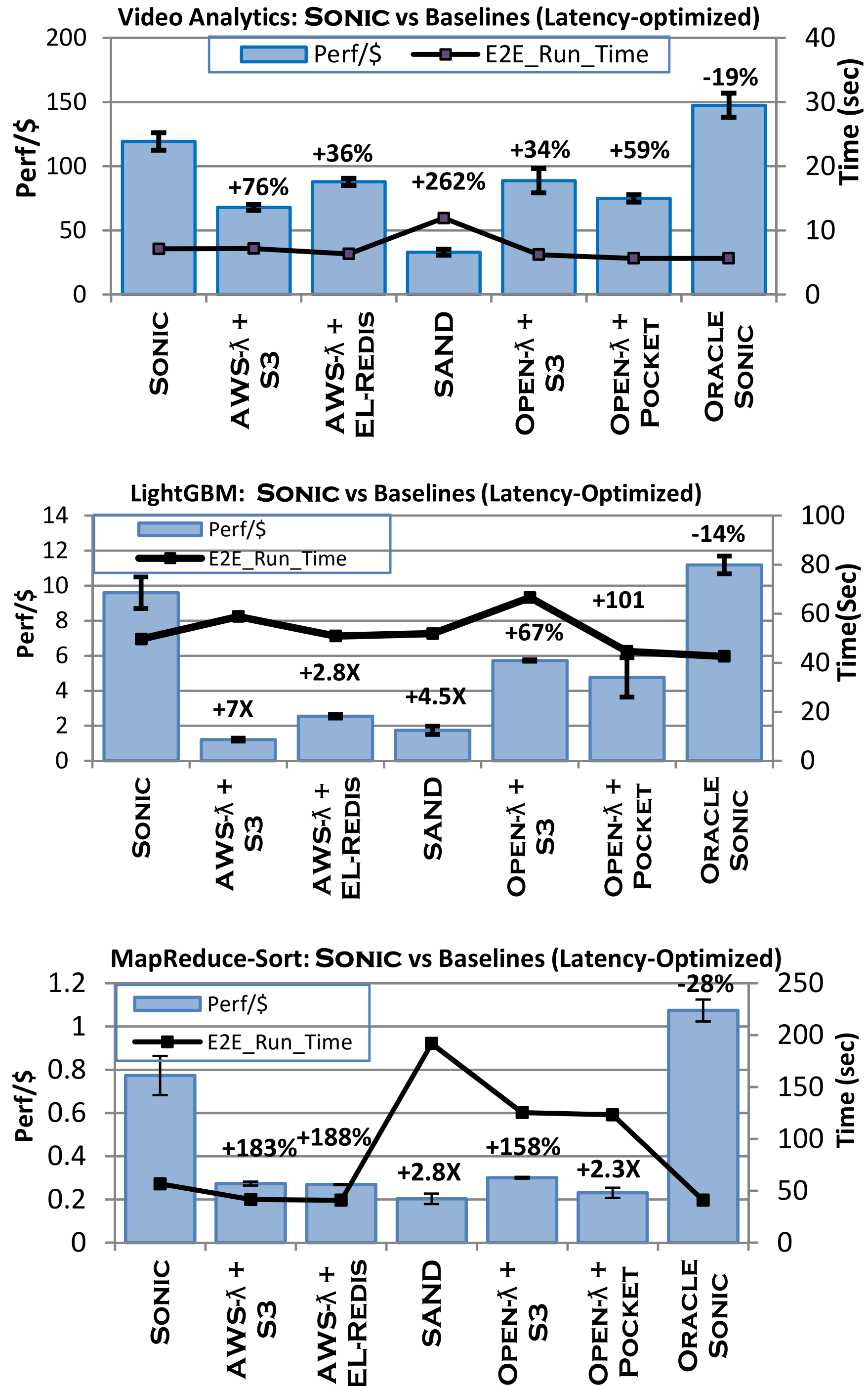


Figure 5.8. Performance of SONIC and the baselines for our three applications with the latency-optimized configuration.

5.6.2 Baselines and Methodology

We compare SONIC to the following baselines:

1. **OpenLambda + S3** [117]: This is the OpenLambda framework deployed on EC2 with

S3 as its remote storage. A new VM is created to host each λ in the DAG. The smallest VM that has enough memory to execute the λ is selected.

2. **OpenLambda + Pocket** [114]: This is a variant of the OpenLambda framework with Pocket (deployed in EC2) as the remote storage. We use Pocket's default storage tier (DRAM) with *r5.large* instance types.

The DRAM storage tier strikes the balance between performance and cost and provides the best Perf/$ (Table 4 in [114]). Comparing the price for one DRAM node to one SSD node in Pocket, DRAM is also the cheapest tier. We vary the number of Pocket nodes for each application to reach the best Perf/$. We include the price of the storage nodes only, excluding master and metadata nodes. We use EC2's per-second level pricing.

3. **SAND** [118]: This baseline leverages data locality by allocating all lambda functions on a single host with rich resources and performing data passing between chained functions through a local message bus.

4. **AWS-λ**: The commercial FaaS platform using two different remote storage systems: S3 and ElastiCache-Redis.

5. **Oracle** SONIC: This is SONIC with fully accurate estimation of DAG parameters and no data passing latency (mimicking local running of all functions). Although impractical, this serves as an upper-bound on performance for any of the three data-exchange techniques selected by SONIC.

Similar to prior OpenLambda evaluations [141], we deploy OpenLambda (and SONIC) on AWS EC2 General Purpose instances (*m5.large*, *m5.xlarge*, *m5.2xlarge*) for their balance between CPU, memory, and network bandwidth. We use the same EC2 family with SAND for fairness. *m5* instances have a network bandwidth of upto 10 Gbps, which we rely on for both *Direct-Passing* and *Remote-Storage*.

All costs follow pricing by Amazon for 01/2021, N. California (Region).

5.6.3 Applications

We use three analytics applications popular as serverless applications and that span the variety of DAG structures.

Video Analytics: Fig. 5.1 shows the DAG for the Video Analytics application, which performs object classification for frames in a given video. It starts with a lambda that splits the input video into chunks of fixed length (10 sec in our case). Then, a second lambda is called for every chunk to extract a representative frame. Next, a third lambda uses a pre-trained deep learning model (MXNET [142]) to classify the extracted frame. It outputs a probability distribution across 1,000 classes over which MXNET is trained. Finally, all the classification results are written to long-term storage.

LightGBM: This application trains decision trees, combining them to form a random forest predictor using `LightGBM` Python library [143]). First, a λ reads the training examples and performs PCA. Second, a user-specified number of λs train the decision trees in parallel (every λ randomly selects 90% for training, 10% for validation). A third λ collects and combines the trained models; then tests the combined model on held-out test data.

Handwritten images' databases: NIST (800K images); MNIST (60K images) used as inputs [144], [145].

MapReduce Sort: This application implements MapReduce sort with serverless functions. In the first stage, K parallel lambdas (*i.e.* mappers; K = user parameter) fetch the input file from a remote data store (*e.g.* S3) and then generate the intermediate files. Next, K parallel lambdas sort the intermediate files and write their sorted output back to storage.

5.6.4 End-to-End Evaluation

We compare the E2E Perf/$ and latency of Sonic vs. other baselines for our three applications. For Video Analytics, we use a video of length 3 min and size of 15MB, which generates a fanout degree of 19 (we vary the video size in § 5.6.6). For LightGBM, we use an input size of 200MB and fanout degree of 6, generating a random forest of 6 trees (we vary

the input size in § 12). For MapReduce Sort, we use an input size of 1.5GB and a fanout degree of 30.

All the results of SONIC include its overheads (11ms for the DAG parameter inference phase; 120ms for the Viterbi optimization phase). We perform online profiling and training with 35 jobs, by which we reach convergence for all applications with a low average Mean Absolute Percentage Error (MAPE) $\leq$ 15%. We show the accuracy of SONIC's predictions in Section. 5.6.6.

Memory configurations. SONIC infers memory requirements automatically from online profiling. Here, we describe how we select the memory allocation for each baseline. AWS Lambda and other serverless offerings scale compute resources relative to a user-specified memory allocation. We run the baselines under two different configurations, which we call "memory-sized" and "latency-optimized". These are respectively shown in Fig. 5.7 and Fig. 5.8. For the memory-sized configuration, we give each lambda just enough memory that it needs to execute.

We determine each λ's memory requirement by measuring the actual memory used when executing the DAG once with all λs using the maximum memory limit (3GB for AWS-λ). For the latency-optimized configuration, we progressively increase the memory allocation as long as a reduction in latency is observed. We report the results for the run with the lowest latency. For OpenLambda, we use SONIC's memory footprint predictor to select the cheapest VM that fits each lambda. We draw several conclusions. First, although AWS-λ allows users to rightsize the allocated memory for their applications, it does not always lead to the best Perf/$ or latency. For example, with Video Analytics and memory-sized configuration, SONIC achieves 442% and 12.9$\times$ better Perf/$ over AWS-λ with S3 and ElastiCache-Redis respectively. However, with (memory) over-provisioning, the latency-optimized configuration improves AWS-λ performance significantly, reducing the gains of SONIC to 76% and 36% with S3 and ElastiCache-Redis respectively (similar observation is shown w.r.t. raw latency). The reason is that AWS-λ allocates all other resources (*e.g.* CPU capacity, network bandwidth, etc.) proportionally to the selected memory requirement [146]. Therefore, as the allocated memory is increased, the latency decreases and the cost also increases. But the latency decreases faster than the cost increases, thus Perf/$ increases.

However, beyond a certain point of over-provisioning, the latency does not decrease further, and thus the Perf/$ begins to decrease.

For MapReduce Sort application, we do not see a significant improvement in Perf/$ for AWS-λ baselines with the latency-optimized configuration when using S3 as the remote storage. This is because the memory footprint of this application is close to the 3GB limit to begin with leaving very little room for over-provisioning. However, for LightGBM application, AWS-λ + ElastiCache-Redis outperforms AWS-λ + S3 due to the lower latency. SONIC still achieves 3.5X and 2.8X over AWS-λ + ElastiCache-Redis in memory-sized and latency-optimized configurations respectively, as ElastiCache-Redis increases the cost significantly.

Compared to SAND, SONIC achieves 187% better Perf/$ with 2× lower latency in the memory-sized case for Video Analytics application. The gain increases to 5.6× and 3.7× for LightGBM and MapReduce Sort applications, respectively. This is again due to the higher memory footprints of these two applications compared to Video Analytics. This reduces SAND's ability to run more λs in parallel and forces a high degree of serialization. This explains the spike in latency for SAND in both applications.

Compared to OpenLambda, SONIC achieves 34% and 59% improvement in Perf/$ with S3 and Pocket, respectively, for Video Analytics.

The performance with Pocket suffers compared to vanilla OpenLambda (*i.e.* with S3) due to Pocket's higher-cost storage (*e.g.* r5.large) without proportional benefit. Note that for OpenLambda, its performance turns out to be identical for the memory-sized and latency-optimized configurations as increasing the memory beyond SONIC's predicted values for each function gives no latency benefit.

Finally, Oracle-SONIC outperforms SONIC, as expected, but not hugely — within 19%-28% across all applications. Recall that Oracle-SONIC has perfectly accurate predictors and assumes no data passing latency.

5.6.5 Scalability

Here, we evaluate SONIC's ability to scale to concurrent invocations of a mix of the applications and larger data sizes.

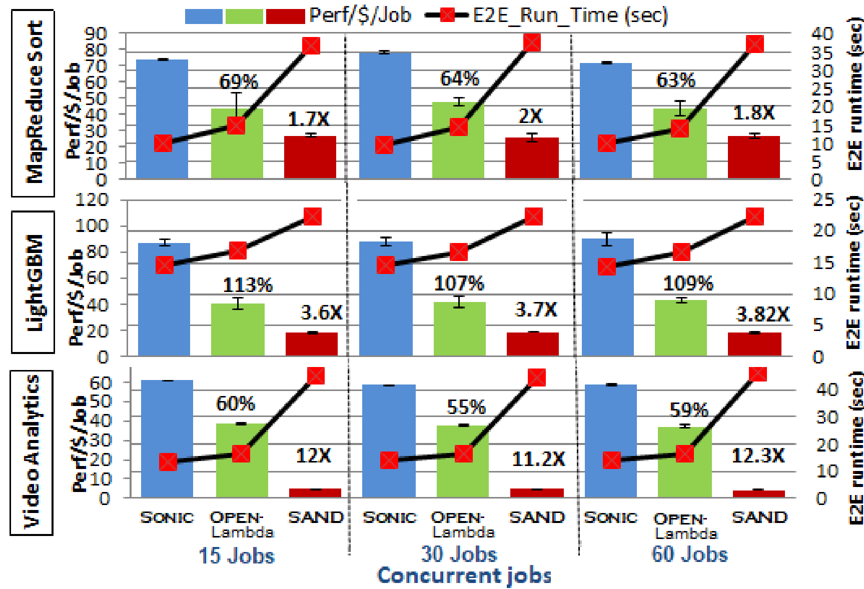


Figure 5.9. Scalability of Sonic, OpenLambda+S3, and SAND with equal portions of our three apps running concurrently. Sonic maintains its improvement over the entire scale, in both Perf/$/job and raw latency.

Make the legend Perf/$/job and make it all three colors (blue, green, maroon — from left to right). Ashraf: If I remove the legend it will be hard to tell which metric is w.r.t bars and which is w.r.t the line with markers

Varying Degree of Concurrency

We compare Sonic to SAND and OpenLambda+S3 serving a mixture workload of our three applications, with equal portions of invocations (jobs) per application. We use a cluster of 52 VMs of type *m5.large* with 2 compute cores per VM. We select SAND as it always prefers to keep data local, while OpenLambda+S3 always uses *Remote-Storage* data passing. OpenLambda+S3 is also the closest baseline to Sonic in terms of performance (Fig. 5.7 and 5.8).

We vary the level of concurrent invocations from 15 (5 per app) to 60 (20 per app). With 60 concurrent app invocations, the cluster executes a total of 480 functions in parallel and all compute cores are fully utilized (except for SAND). We show the Perf/$/job and E2E runtime for every app in Figure 5.9[2]. We notice that the gain of Sonic over both baselines

[2]↑We normalize by the number of jobs because naturally the total cost increases with the number of jobs completed and the normalization brings out the important trend that the metric is flat across the scales, implying perfect scalability.

is consistent across the different degrees of concurrency, which shows SONIC's ability to seamlessly scale with the number of concurrent invocations. It is of course not surprising that the VM infrastructure scales up — the question was would SONIC scale up as well. This experiment answers that question in the affirmative (within the scales of the experiment), for SONIC as well as for the two baselines. This is expected from the design of SONIC where most of its components are stateless and different instances are spun up to handle more input jobs (Section 5.4.5). Since SAND schedules the entire job to execute on a single VM, it cannot utilize all the compute cores and hence suffers from very high latency. In contrast, OpenLambda uses *Remote-Storage* passing between functions, which allows scheduling functions on different VMs and utilizing the available compute resources. SONIC's hybrid approach achieves both lower E2E execution time along with high resource utilization, and therefore, achieves better Perf/$ and raw latency than all baselines.

Varying Intermediate Data Size

In Fig. 5.12, we show the impact of changing the intermediate data size on SONIC's performance vis-à-vis two baselines.

SONIC achieves lower E2E latency and higher Perf/$ across all input sizes by predicting the corresponding DAG parameters and selecting the best co-location of lambdas and data passing methods. Second, the normalized Perf/$ of SONIC and OpenLambda+S3 increases with higher input sizes. The reason is that the compute:data passing time ratio for this application increases with higher input sizes. This, in turn, is because the data passing time increases linearly with the input size, whereas its computation time grows faster than linear (PCA has quadratic compute complexity [147] and it dominates the total computation time).

Recall that the Oracle assumes no data passing latency but it counts computation time. Accordingly, the higher the compute:data passing time ratio, the lower the gap between Oracle and the baselines (except SAND that has no data passing component).

5.6.6 Microbenchmarks

Here we evaluate Sonic's online training accuracy, sensitivity to input content, prediction noise, and cold-start times.

Prediction Accuracy with Online Refinement

As discussed in Sec. 5.4.2, our solution profiles jobs to characterize the relation between input size and the data passing relevant parameters. This training phase is done online while serving production jobs, and we use remote storage data passing (Sonic's default) until convergence is achieved. We show Sonic's accuracy in prediction of execution parameters for new input sizes across the three applications. First, we execute the DAG of each application with jobs of varying input sizes while we measure the DAG's execution parameters (*i.e.* λ's memory footprint, λ's compute time, data volume on every edge, and fanout degree in every stage). Next, we divide the collected data into train and test sets. We fix the test set size while we vary the number of jobs used for training to show the reduction in error w.r.t. training with more jobs for every application. In each training run, we generate the regression models for all execution parameters in the DAG. For example, PCA's runtime regression estimated equation is: $Y = 0.0024\ X^2 + 12.764$, and PCA's estimated memory equation is: $Y = 16\ X + 185.5$, where X is the input size in MBs.

For Video Analytics, we collect 60 YouTube videos with lengths that vary uniformly between 1 min and 1 hour and belong to 5 different categories with equal representation of each: News, Entertainment, Nature, Sports, and Cartoon. We similarly execute the LightGBM and MapReduce Sort applications with 60 jobs each. For LightGBM, we use the MNIST database of handwritten digits [144]. The data has 60,000 training images and 10,000 test images. To execute the DAG with varying input sizes, we sub-sample the training data and vary the sampling rate between 5% and 100% uniformly. For MapReduce Sort, we generate randomly shuffled data and vary the size of the data between 3M records (255MB) and 12M records (1.5GB). We also vary the number of Mappers and Reducers uniformly between 5 and 50.

Increasing the number of jobs used in training expectedly reduces the prediction error for all applications. With 35 jobs we reach convergence for all predictions with a low average Mean Absolute Percentage Error (MAPE) $\leq 15\%$ as shown in Figures 5.10 for Video-Analytics. This low prediction error is essential for SONIC to determine the best lambda placement information and data passing decisions for new jobs with new input sizes. Optionally, users can set a higher level of acceptable error to reduce the number of training jobs.

We notice that Video Analytics incurs the highest prediction error among the three applications. This is because our collected videos vary significantly in their bitrate, which impacts the relation between the input size and the video's length. Recall that SONIC is content-agnostic, it does not consider any information that is dependent on the content of the data when it makes its prediction. Although this negatively impacts the prediction accuracy for content-dependent applications, it allows SONIC to generalize to a wide range of applications without the need for special processing for each type of application. Furthermore, policies for public cloud providers often prohibit any visibility into the client applications. We evaluate SONIC's sensitivity to input content in § 5.6.6.

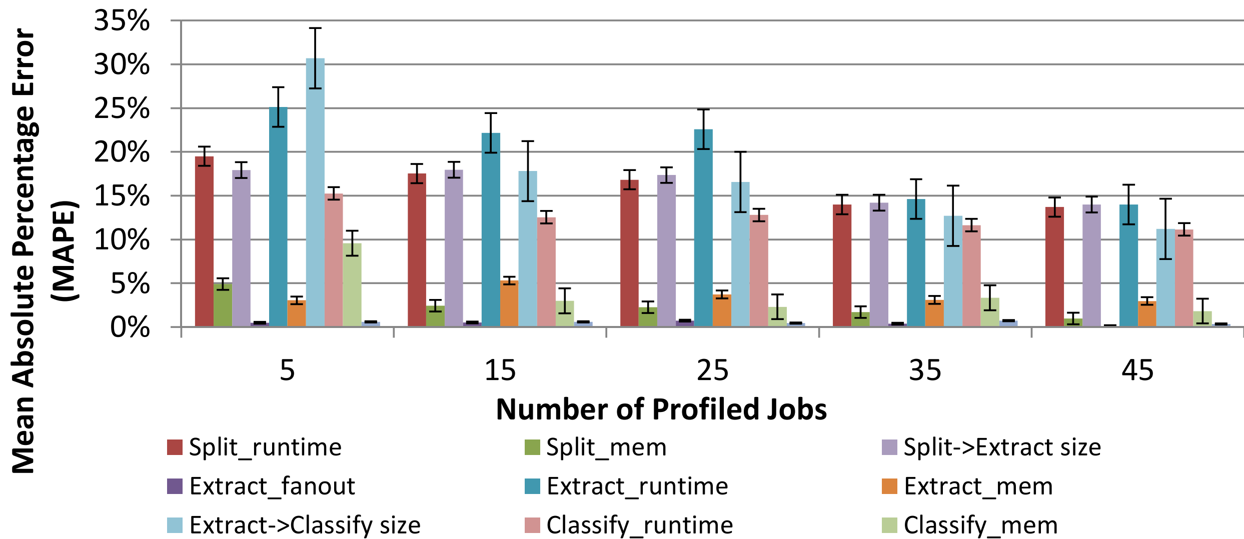
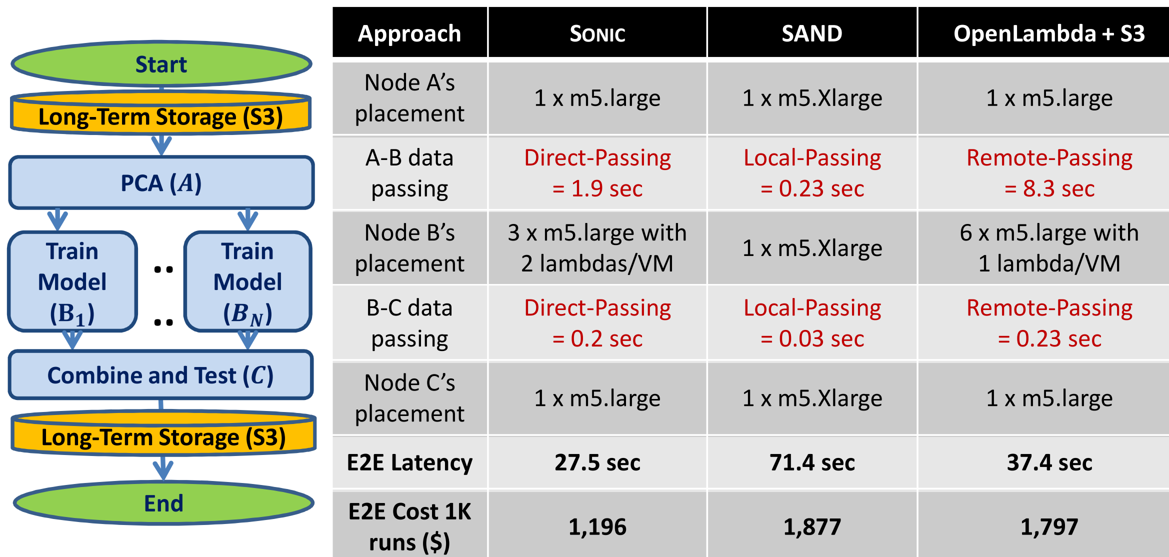


Figure 5.10. Error in parameter estimation for Video Analytics application. Convergence point is reached with *e.g.* 35 jobs. Split_mem, Extract_mem, and Classify_mem represent memory footprints for Split, Extract and Classify functions respectively. All parameters are predicted using polynomial regression models, which take the application's input size in MBs as input.



Approach	Sonic	SAND	OpenLambda + S3
Node A's placement	1 x m5.large	1 x m5.Xlarge	1 x m5.large
A-B data passing	Direct-Passing = 1.9 sec	Local-Passing = 0.23 sec	Remote-Passing = 8.3 sec
Node B's placement	3 x m5.large with 2 lambdas/VM	1 x m5.Xlarge	6 x m5.large with 1 lambda/VM
B-C data passing	Direct-Passing = 0.2 sec	Local-Passing = 0.03 sec	Remote-Passing = 0.23 sec
Node C's placement	1 x m5.large	1 x m5.Xlarge	1 x m5.large
E2E Latency	**27.5 sec**	**71.4 sec**	**37.4 sec**
E2E Cost 1K runs ($)	**1,196**	**1,877**	**1,797**

	PCA	Train	Combine
m5.l	**VM-Count = 1** **Exec-Time = 9.4** **Price (x1K) = $686**	**VM-Count = 3** **Total Latency = 15.9 s** **Price (x1K) = $480**	**VM-Count = 1** **Total Latency = 2 s** **Price (x1K) = $24**
m5.xl	VM-Count = 1 Exec-Time = 9.4 Price (x1K) = $1,370	VM-Count = 2 Total Latency = 15.2 s Price (x1K) = $640	VM-Count = 1 Total Latency = 1.97 s Price (x1K) = $48
m5.2xl	VM-Count = 1 Exec-Time = 9.4 Price (x1K) = $2,744	VM-Count = 1 Total Latency = 14.6 s Price (x1K) = $840	VM-Count = 1 Total Latency = 1.96 s Price (x1K) = $72

Direct Passing (PCA → Train); Direct Passing (Train → Combine)

Figure 5.11. Example showing the benefits of Sonic over baselines. The table at the bottom shows possible λ placements for each stage in the LightGBM application and their corresponding latency and cost. We highlight the best sequence of decisions that achieves the best Perf/$ for the entire DAG.

Sonic's Performance Improvement Root Causes

Here we highlight the root causes for Sonic's improved performance over baselines SAND and OpenLambda + S3. We show an example DAG for LightGBM application along with the data passing and lambda placement decisions made by each approach in Figure 5.11. We also show Sonic's selected optimum Viterbi path in the table at the bottom. We run the LightGBM application with a fanout degree of 6 and show the E2E latency and Cost for all baselines. First, SAND leverages data locality between all stages and hence uses the same single VM of size m5.Xlarge for the entire application. This causes the fanout stage

(TrainModel) to experience serialized execution as this single VM does not have enough resources to execute all invocations in parallel and hence increases the E2E latency to 71.4 sec. Compared to OpenLambda + S3, we notice that passing data between PCA and TrainModel is very slow with remote passing as it takes 8.3 sec. However, if direct passing is used (as done by SONIC), the data passing time becomes 1.9 sec only. We also notice that SONIC places each pair of lambdas in one VM using a total of 3 VMs. This makes direct passing faster as it only needs to transfer 3 copies of the transformed training data. Recall that this application has a broadcast fanout and all lambdas in TrainModel stage get the same copy of PCA's output file. In conclusion, with SONIC's optimized data passing and placement decisions, the E2E latency is reduced by 27% over OpenLambda + S3 and by 62 % over SAND, and the Cost is reduced by 33% over OpenLambda + S3 and 36% over SAND.

Sensitivity to Input Content

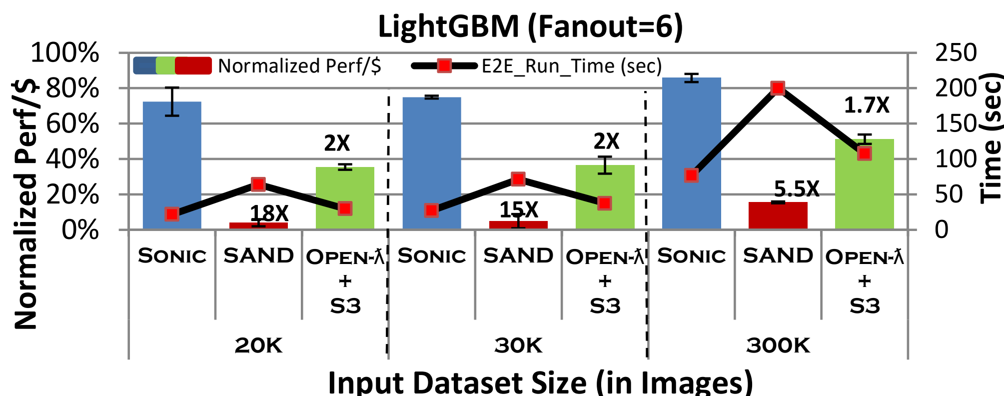


Figure 5.12. Normalized Perf/$ of SONIC vs SAND and OpenLambda+S3 with varying input sizes. We fix the Fanout-degree=6 and change the number of images used in training the Random-Forest model.

SONIC uses only the input size information, as opposed to content awareness, to predict DAG parameters for a new job[3]. For example, for Video Analytics (Fig. 5.1) some of the parameters like the intermediate data size are sensitive to the video size (bytes), which

[3]↑Although this hurts SONIC's prediction accuracy for content-dependent DAGs, it allows generalizing without application-specific processing. Further, public cloud providers often are not allowed to look into client data.

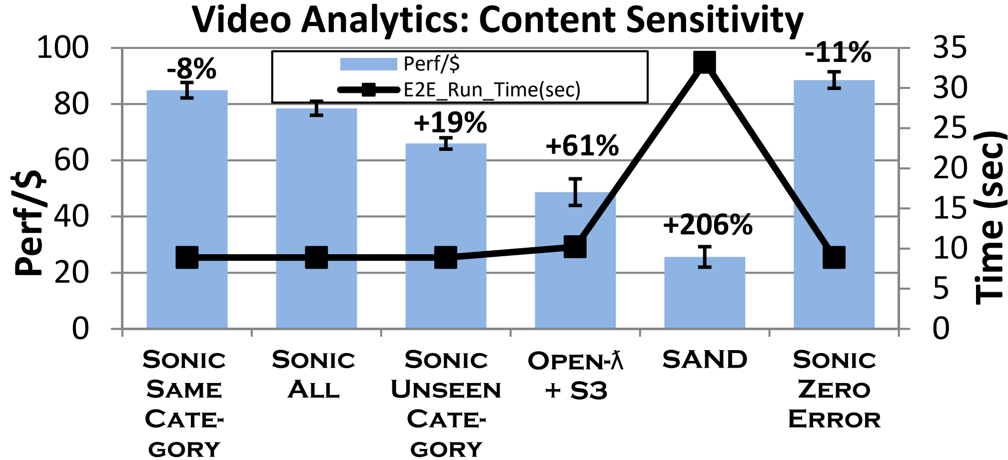


Figure 5.13. Effect of training SONIC on YouTube video categories similar or dissimilar to test. % over the bars represent the SONIC's (second bar from left) gain over that baseline.

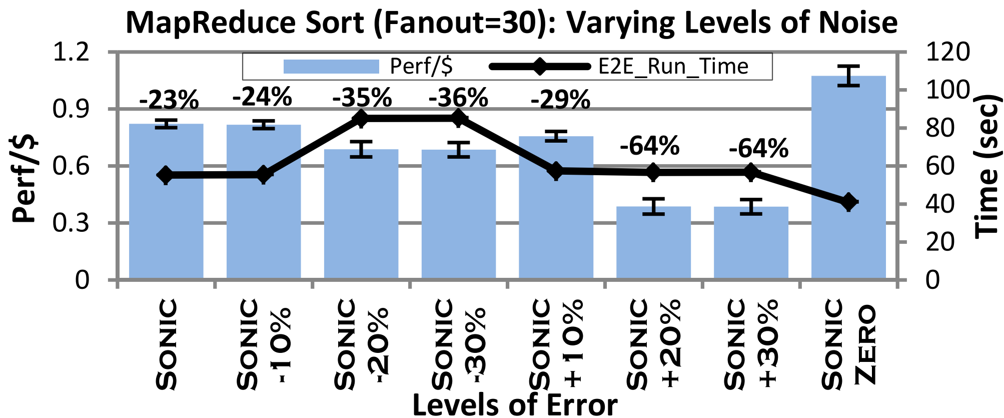


Figure 5.14. Impact of noise in SONIC's memory footprint predictions. Errors of less than ± 10% have small effect and over-prediction has higher effect than under-prediction. The values over the bars are w.r.t. SONIC with zero error (rightmost).

depends on video bitrate specification. We want to examine how SONIC's performance would be impacted on test videos different from training. In Fig. 5.13, we show the performance gain for three variants of SONIC, testing on a 396-sec video from the Sports category. First, we train SONIC on 60 videos from the same Sports category, which shows the best performance among the 3 variants. Second, we show SONIC's performance with training videos from 5 different categories (60 in all, split equally), which shows an 8% performance reduction vis-à-

vis the first. The third variant is trained with 60 videos from the *News* category, which has a 25% lower bitrate than the *Sports* category on average. This difference in categories causes a further performance reduction by 19% due to the error in predicting the fanout degree (40%) and intermediate data size between the `Split_Video` and `Extract_Frame` functions (21%). All three variants still show a significant gain over SAND and OpenLambda baselines. As expected, the higher the difference in critical features of the training and testing data (that can impact the DAG's parameters), the lower is Sonic's performance. Critical features are those that affect the compute time or the data passing volume, *e.g.* video bitrate. One solution to this limitation is to cluster the jobs based on the critical features and train a separate prediction model for each cluster.

Tolerance to Prediction Noise

We examine Sonic's sensitivity to prediction noise. We use the MapReduce Sort application with 30 each of map and reduce functions and apply varying levels of synthetic noise to our memory footprint predictions. We show the impacts of over-predicting (*i.e.* the predicted memory footprint is *higher* than the actual), and under-predicting in Fig. 5.14. For calibration, the natural error of Sonic in prediction of memory parameters is 7%. We draw several conclusions. First, error levels of less than ±20% have little impact since with low levels of noise, the (categorical) decisions by Sonic for lambda-placement and data passing are unchanged. Second, under-predicting (the bars with -ve errors) has lesser impact on Sonic than over-predicting. Under-predicting causes Sonic to allocate fewer VMs (1 VM per 3 lambdas in this experiment) than without synthetic noise (1 VM per 2 lambdas). This causes the execution of only two lambdas in parallel while queuing the third lambda, increasing the job's E2E execution time. On the other hand, over-estimation of the memory causes Sonic to allocate more VMs than what the job actually needs (1 VM per lambda). The increase in latency with under-prediction is partly compensated for by the reduction in the $ cost, while with over-prediction, the increase in the $ cost dominates over the reduction in latency.

Varying Cold-Start Overheads

In this experiment, we evaluate the effect of varying cold:hot execution times. We use a synthetic application of one stage containing 10 parallel functions and vary the function's startup:steady state compute ratios. We compare SONIC to two static baselines, SAND and OpenLambda+S3, in Fig. 5.15. SAND always prefers data locality and hot execution over parallelism. We notice that this approach is beneficial for lambdas that have a gap of 5× or more between cold and hot execution times. However, this solution is counter-productive when the gap between cold and hot executions is lower, unnecessarily forcing lambdas to run sequentially. The exact opposite happens with OpenLambda — it is competitive with SONIC for cold to hot execution ratios of 2× or less but suffers increasingly as the ratios become higher as it always incurs cold-start costs. SONIC achieves close-to-optimal performance across the entire range of cold-to-hot execution ratios due to its ability to estimate the execution times under cold and hot executions and to select the best lambda placement and data passing approach dynamically. In practice, we find that the ratio varies in the range $[1, 3.6]$ (highest for Video Analytics' Classify frame due to a heavy NN model); prior work has shown that the ratio can be as high as 9.6× (Figure 21 in [141]). We also evaluate SONIC with varying fanout and ratios of compute time to total execution time (=compute time+data passing time). SONIC's gain over OpenLambda is more significant at lower compute ratios as data exchange dominates, while it is more significant over SAND at higher fanouts (data locality hurts parallelism).

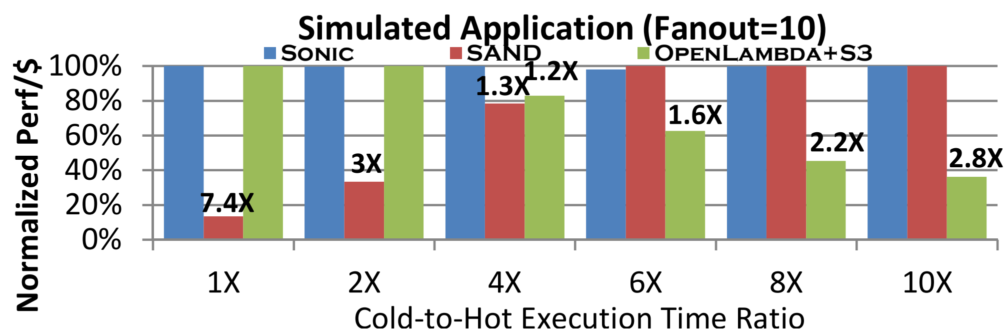


Figure 5.15. Effect of varying ratios of cold to hot execution times on SONIC, SAND, and OpenLambda+S3. Normalized Perf/$ is calculated by dividing the Perf/$ by the max across the three techniques.

5.7 Related Work

Data-passing in serverless environments: We are not the first to identify data-passing latency as a key challenge for chained lambda execution [101], [115], [117], [148], [149]. Pocket [114] and Locus [99] implement multi-tier remote storage solutions to improve the performance and cost-efficiency of ephemeral data sharing in serverless jobs. SONIC can leverage these remote storage systems (*e.g.* we have evaluated SONIC with Pocket) while automatically optimizing data-passing performance with *VM-Storage* (as in SAND [118]) and *Direct-Passing* methods, which minimize data copying. Pocket also requires hints from the user about parameters of the DAG, which we infer using our modeling approach. as we discussed, can you also tout the direct-passing technique to be new to the serverless ecosystem and introduced by SONIC; I feel that is a big deal and we have not really claimed that novelty anywhere

Prior systems have shown that serverless functions can communicate directly using NAT (network address translation) traversal techniques. ExCamera [103] uses a rendezvous server and a fleet of long-lived ephemeral workers to enable direct communication. However, it needs both endpoint lambdas to be executing at the time of the data transfer, which SONIC does not. gg [150] is a framework for burst-parallel applications that supports multiple intermediate data storage engines, including direct communication between lambdas, so gg and ExCamera have already introduced direct passing so I guess not a novelty; I now recall ExCamera from our discussion which users can choose from. In contrast, SONIC abstracts and adaptively selects the optimal data-passing mechanism.

The need for processing state within serverless frameworks is being increasingly recognized [99], [151]–[153], *e.g.* for fine-grained state sharing or coordination among processes in ML workflows. This trend will emphasize the importance of efficient data passing among functions like SONIC provides. Automated tuning systems of clusters configurations have been proposed in [28], [138], [154]. However, these systems use black-box machine learning optimization and rely on hundreds of offline profiling runs to build accurate performance models. Cloudburst proposes using a cache on each lambda-hosting VM for fast retrieval of

frequently accessed data in a remote key-value store [155], adding a modicum of statefulness to serverless workflows.

SONIC does not cache data, but still exploits data locality with its lambda placement.

Efficiency of serverless executions. There is flourishing work to make serverless executions more efficient. One strategy optimizes cold-start latencies, which will influence SONIC's function placement decisions as in § 13 (*e.g.* SOCK [141], SEUSS [156], and Firecracker [157].)

Another strategy optimizes in the isolation *vs.* agility spectrum (*e.g.* Firecracker [157], MVE [158] (supporting hugely concurrent services as in popular games), Spock [159], and Fifer [160] (hybrids of serverless and other cloud technologies for microservices). The data-passing selection in these works can benefit from SONIC. Costless [109] optimizes lambda fusion and placement, reducing the number of state transitions. This contribution is orthogonal and beneficial to SONIC.

Cluster computing frameworks: Many distributed computing frameworks, such as Spark [161], Dryad [162], CIEL [163], and Decima [164] use DAGs of jobs to schedule tasks. With some engineering effort, SONIC can be used to support the data passing on these DAGs. However, SONIC stays close to the spirit of serverless in that it requires a minimal number of user hints or configuration options.

5.8 Conclusion

Optimizing the cost and performance of analytics jobs on serverless platforms requires minimizing the data passing latency between chained lambdas. The optimal data passing method depends on application-specific parameters, such as the input data size and the degree of parallelism. We presented SONIC, a system that *jointly* optimizes the inter-lambda data exchange method and lambda placement.

SONIC performs online profiling to determine the relation between the application's input size and its DAG parameters. Afterward, SONIC applies an online Viterbi algorithm, to globally minimize the application's end-to-end latency, normalized by cost. SONIC achieves lower ($ cost-normalized) latency against four competitive baselines for three popular server-

less applications. Moreover, Sonic is able to adjust the best data passing method based on infrastructure changes such as network bandwidth fluctuations. In ongoing work, we are designing Sonic to handle conditional control flows in the application DAG through content-aware prediction.

6. ORION AND THE THREE RIGHTS: SIZING, BUNDLING, AND PREWARMING FOR SERVERLESS DAGS

6.1 Abstract

Serverless applications represented as DAGs have been growing in popularity. For many of these applications, it would be useful to estimate the end-to-end (E2E) latency and to allocate resources to individual functions so as to meet probabilistic guarantees for the E2E latency. This goal has not been met till now due to three fundamental challenges. The first is the high variability and correlation in the execution time of individual functions, the second is the skew in execution times of the parallel invocations, and the third is the incidence of cold starts. In this paper, we introduce Orion to achieve this goal. We first analyze traces from a production FaaS infrastructure to identify three characteristics of serverless DAGs. We use these to motivate and design three features. The first is a performance model that accounts for runtime variabilities and dependencies among functions in a DAG. The second is a method for co-locating multiple parallel invocations within a single VM thus mitigating content-based skew among these invocations. The third is a method for pre-warming VMs for subsequent functions in a DAG with the right look-ahead time. We integrate these three innovations and evaluate Orion on AWS Lambda with three serverless DAG applications. Our evaluation shows that compared to three competing approaches, Orion achieves up to 90% lower P95 latency without increasing $ cost, or up to 53% lower $ cost without increasing P95 latency.

6.2 Introduction

Serverless computing (FaaS) has emerged as an attractive model for running cloud software for both providers and tenants. Recently, serverless environments are becoming increasingly popular for video processing [104], [165], machine learning [115], [166], and linear algebra applications [167], [168]. The requirements of these applications can vary from latency-strict (*e.g.* Video Analytics for Amber Alert responders [169]) to latency-tolerant but cost-sensitive (*e.g.* Training ML models [170]). Accurate latency estimation is essential

to meet the requirements for both, as the cost in FaaS platforms is based on resource usage and runtime. The workflow of these serverless pipelines is usually represented as a directed acyclic graph (DAG) in which nodes represent serverless functions and edges represent data flow dependencies between them.

Serverless platforms experience high performance variability [109], [111], [114], [118], [171], [172] due to three primary reasons: First, some function invocations have cold starts. Second, there is skew in the execution time of various functions because of different content characteristics that the functions operate on. Third, there exists skew in the execution time due to variability in infrastructure resources (*e.g.* network bandwidth for an allocated VM). Because of this variance in performance, predicting the mean (or median) execution time of individual functions is not sufficient to meet percentile-specific latency requirements (*e.g.* P95) for serverless DAGs. Rather, a distribution-aware modeling technique is essential to capture this variability and provide accurate latency SLOs.

Key Idea. We propose ORION, a novel technique for performance modeling of serverless DAGs to estimate the end-to-end (E2E) execution time (synonymously, E2E latency). We leverage this model to enable system optimizations such as allocating resources to each function to reduce E2E latency while keeping $ costs low and utilization high. The different components of ORION are shown in Figure 6.1. We derive insights about serverless DAGs from analysis of production traces at *Azure Durable Functions* [173]. This analysis drives our performance model and the design features.

First, we observe the inherent performance variability in serverless DAGs and therefore represent the latency of a single function, as well as that of the entire DAG, as a distribution rather than a single value. For example, Figure 6.5 shows the variance in execution times for the top-5 most frequently invoked DAG-based applications. The execution times of invocations of the *same* DAG vary significantly, and the P95 latency is 80X of the P25 latency, averaged over the 5 applications. Thus, our performance model profiles the latency distribution for each function in the DAG and builds a performance model to capture the impact of varying the resource allocation to that function on its latency distribution. Afterward, we estimate the DAG E2E latency distribution by applying a series of two statistical combination operations, `convolution` and `max`, for in-series and in-parallel functions, respectively.

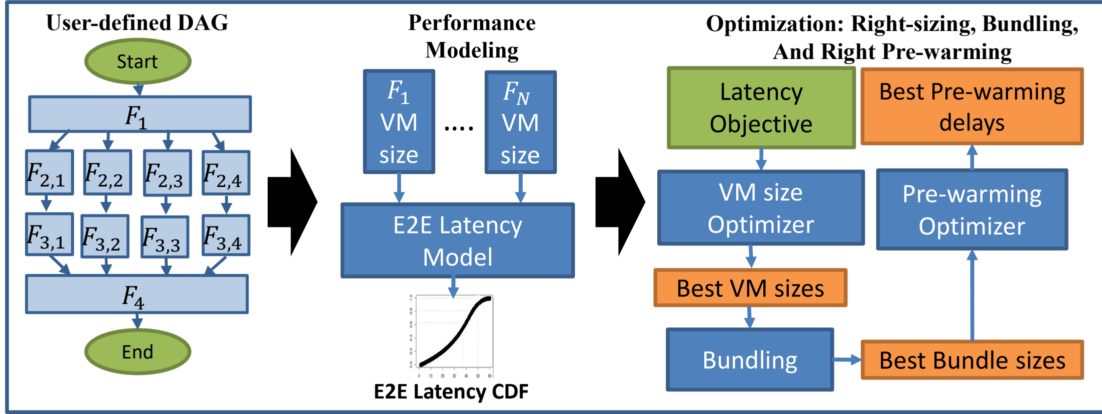


Figure 6.1. ORION Overview. ORION profiles the DAG, estimates E2E latency CDF using *CONV* and *MAX*, and performs three system optimizations — right-sizing, bundling, and right pre-warming.

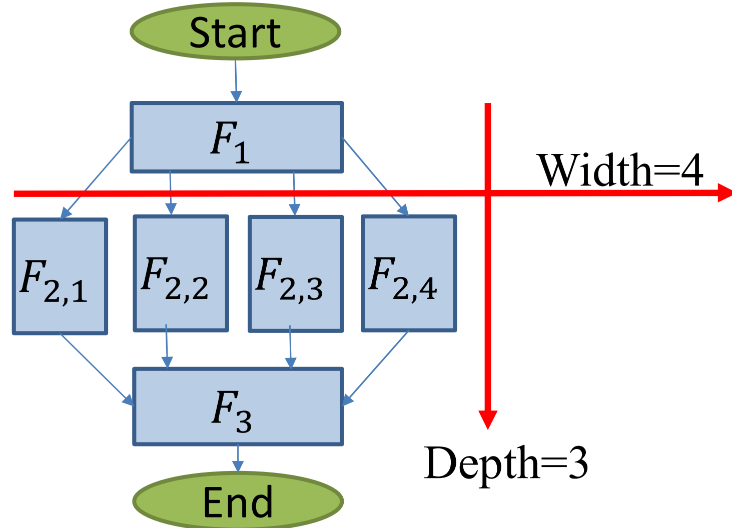


Figure 6.2. Illustration of DAG Depth (*i.e.* number of in-series stages) and Width (*i.e.* Maximum fanout degree).

Moreover, we observe it is essential to consider the correlation between the workers across stages and within the same stage to accurately estimate the joint distributions. Our performance model does not require expensive profiling, as is needed for the leading technique, Bayesian Optimization [72], [174], [175].

We then use our performance model to design three optimizations for serverless DAGs. (1) **Right-sizing**: Finding the best resource configurations for each function to meet an E2E latency objective (*e.g.* 95-th percentile latency $< X$ sec) with the minimum cost. (2)

Bundling: Identifying stages where co-locating multiple parallel instances of a function together to be executed on a single VM will be beneficial. The benefit arises when there is computation skew among the parallel workers caused by different content inputs. (3) **Right pre-warming**: The VMs to execute the functions in the DAG are pre-warmed just right, ahead of time, so that cold starts can be avoided while keeping provider-side utilization of resources high. With these three optimizations, ORION accurately meets latency SLOs while reducing execution cost (Figure 6.3).

ORION can be deployed by either the cloud service provider or by the end consumer. For the former, the use case is to provision its resources better to meet client SLOs. For the latter, the driving force is the appropriate resource provisioning to minimize E2E latency while minimizing execution cost.

Evaluation and Insights. We evaluate ORION on three serverless applications on AWS Lambda: two variants of Video Analytics [114], an ML Pipeline [101], and an NLP Chatbot [176] application. Our evaluation, comparing to three approaches: Best-Memory [177], [178], Speculative-Execution [159], and CherryPick [72], shows that the benefits of ORION persist across the different applications with different DAG structures, skews in execution times, and invocation frequencies. Our evaluation brings out the following insights:

(1) Latency correlation across stages *and* within a stage is important (Tables 6.1 and 6.2). Even when correlations are weak, not taking them into account can introduce significant error in the latency estimations. Further, to make the solution scalable, we have to compute the E2E latency estimation using just the right degree of correlation.

(2) Selecting the best VM sizes for serverless functions in a DAG is challenging (Table 6.3). This is because different resources in a VM scale up differently with their size. For example, for AWS-λ, CPU cores go from fractional to a maximum of six, network bandwidth only increases till a level, and disk capacity stays constant.

(3) Bundling multiple parallel instances of a function within a single VM helps when there is content skew and the functions are scalable, *i.e.* they can make use of additional resources (Figure 6.17). Here also, the degree of bundling has to be carefully determined so as not to cause resource contention.

(4) Using the DAG structure and the function latency model, we can estimate the right time

to pre-warm VMs and thus mitigate cold starts (Figure 6.16). With pre-warming, lower P95 latency is achieved with higher resource utilization.

In summary, the main contributions of Orion are the following: (1) Workload characterization for serverless DAGs seen by *Azure Durable Functions*. (2) A performance model for E2E latency of serverless DAGs; (3) A method for assigning the right resources for serverless DAGs to meet the E2E latency requirements within a reduced $ cost; (4) An application-independent way to bundle multiple function invocations to mitigate skews. (5) A method for deciding when to start pre-warming VMs for functions in a serverless DAG to minimize initialization latency while providing acceptable resource utilization.

Orion is open sourced and we release its code, the workload characterization data, and the evaluation applications at: https://github.com/icanforce/Orion-OSDI22

6.3 Motivation

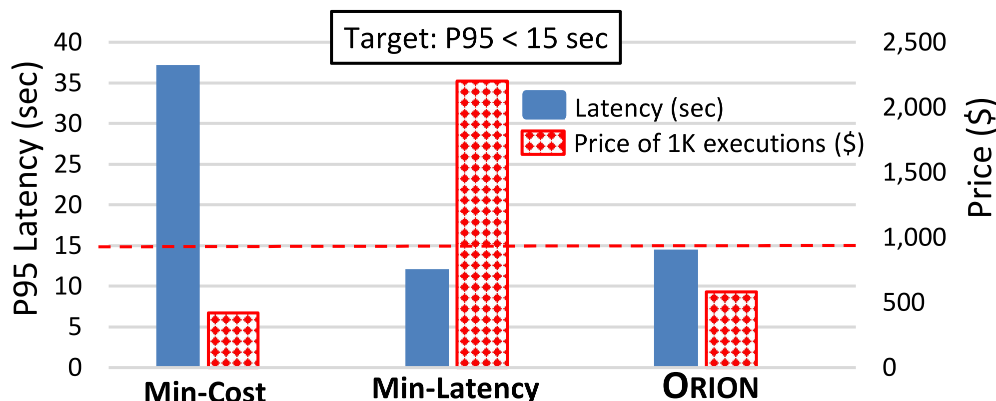


Figure 6.3. Orion improves both latency and cost of Video Analytics DAG. Executions with min VM size (*i.e.* Min-Cost) and max VM size (*i.e.* Min-Latency) are for reference, and all latencies are for warm executions.

6.3.1 Workload Characterization

Definitions. A DAG is a chain of two or more serverless function stages that execute in-series. A stage consists of one or more parallel invocations (a.k.a. instances) of the same serverless function. DAG *depth* is the number of stages in the DAG. DAG *width* is the max

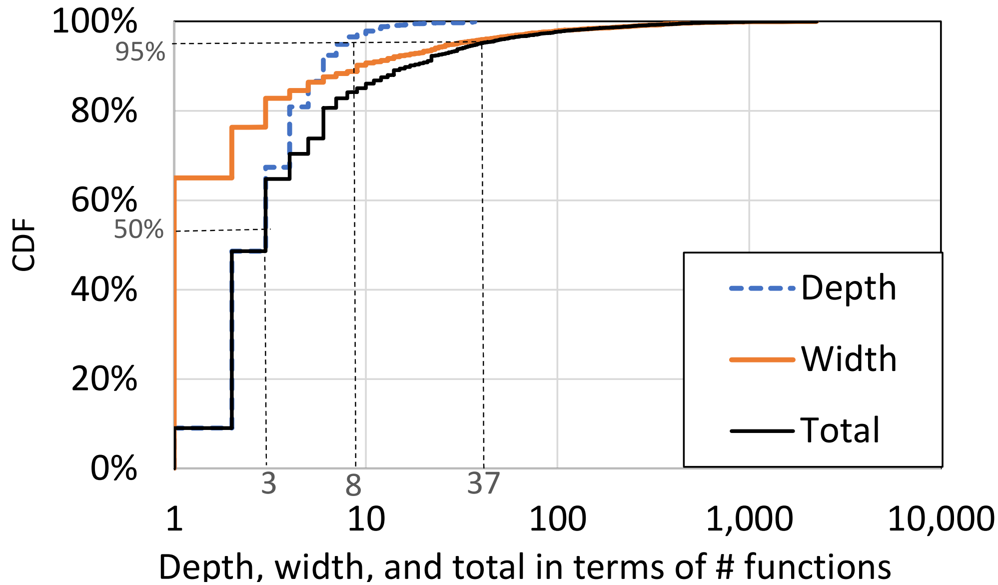


Figure 6.4. Characterization of depth (number of stages), width (degree of parallelism), and total number of nodes. Depth is low (P50 = 3; P95 = 8). 65% of DAGs are linear chains (no fanout) and width reaches 37 at the 95th percentile.

number of parallel worker functions (a.k.a. fanout degree) across all stages in the DAG. A DAG with $width = 1$ means it is a chain of sequential function invocations, whereas a DAG with $width > 1$ means it has at least one parallel stage. We show an illustration of DAG depth and width in Figure 6.2. Finally, we define skew in a parallel stage as the ratio of the execution times of the slowest to the fastest worker function.

Analysis. In this section, we characterize the workload of serverless DAGs from *Azure Durable Functions*. We collect a subset of the logs of DAG executions from six datacenters — three located in the US, two in Europe, and one in Asia from 10/19/21 to 10/25/21 (1 week). The workload we analyze includes 20M-30M DAG executions/day. From our characterization, we draw the following conclusions, which in turn motivate various design decisions of ORION.

(1) DAG Structure and Execution Time. We study the depth and width of serverless DAGs and their distributions in Figure 6.4. We account for the DAG invocation frequencies — if a DAG is invoked N times, its width and depth are included N times in the CDFs. First, we notice that DAG depth is low, with a median of 3 stages and a P95 of 8 stages.

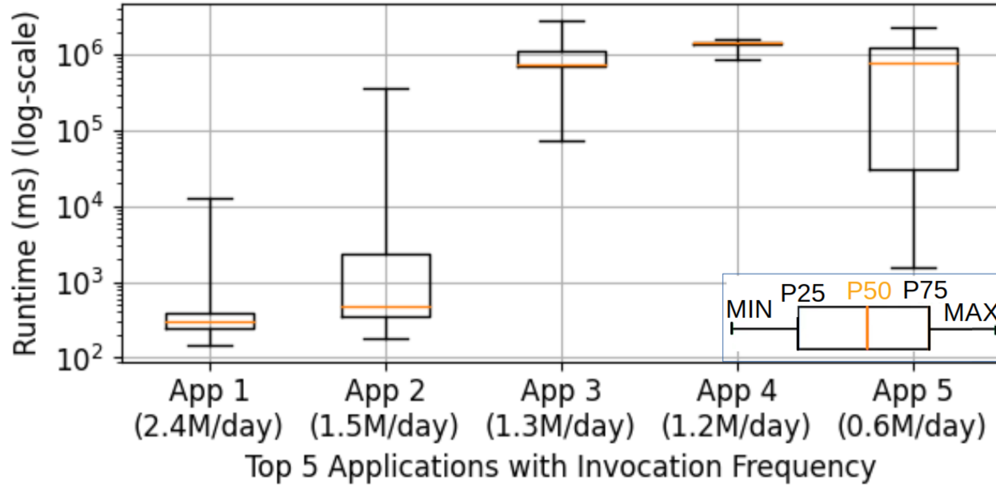


Figure 6.5. Latency distributions for the top-5 most frequent applications executed by *Azure Durable Functions* over a period of 1 week. We notice that the execution time varies significantly across different invocations of the same application.

Second, although 65% of the DAGs are linear chains (no fanout), the width can grow to as high as 37 in the 95th percentile. We also study the execution time of DAGs and find that they can range from 10 ms to 112 min, with a median of 3.7 sec and a mean (weighted by invocation frequencies) of 48 sec. This motivates the need for considering DAG structure while minimizing the E2E latency.

(2) DAG Invocation Frequency and Relation to Cold Starts. Figure 6.7 shows the frequency of invocations/day for each DAG and the corresponding percentages of cold starts. We notice that the frequency of invocations is heavily skewed, with the top-5 most frequent DAGs accounting for 46% of all invocations. Thus, the optimized executions of these frequently invoked DAGs result in higher cost savings. We also notice that the percentage of cold starts decreases with higher invocation frequency. For example, DAGs invoked ≥ 100 times/day have very low percentages of cold start with a median of 0.35%. However, most of the DAGs are rarely invoked: 80% of the DAGs have an invocation frequency of < 100 times/day, and these experience a high percentage of cold starts with a median of 50%. This shows an increase in the proportion of infrequent serverless applications (DAG-based in our case) compared to a prior study [179], which showed that 55% of the serverless applications are invoked less than 100 times/day.

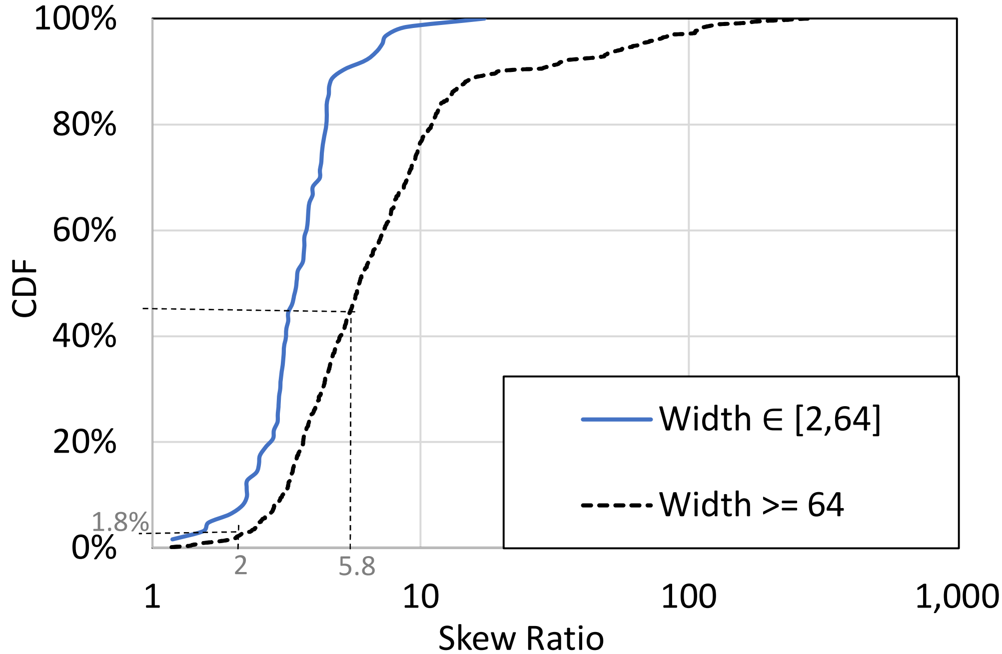


Figure 6.6. Skew CDF for stages with different width ranges. 98.2% of the DAGs have a skew $\geq$ 2X, and skew increases for wider DAGs.

Hence, using keep-alive policies (as done by most major FaaS platforms) for those DAGs will not be sufficient and pre-warming becomes essential to mitigate cold starts. Even for the DAGs that are not the most frequently invoked, keeping E2E latency low is a desirable feature.

(3) Variance in DAG Execution Time. Figure 6.5 shows a boxplot for the execution time of the top-5 most frequently invoked DAGs (which contribute 46% of all invocations). We notice that the variance in execution times across different invocations of the same DAG is high. For example, the P95 latency is 80X the P25 latency, averaged over the five DAGs. We also notice that the *distribution of execution times can be heavily skewed.* For example, P50 is not centered between MIN and MAX, or between P25 and P75. Hence, it is essential to represent E2E latency of serverless DAGs as a distribution when modeling their performance to capture this variability.

(4) Degree of Skew. Figure 6.6 shows the skew distribution for parallel stages for different ranges of DAG widths. We notice that skew among parallel worker functions is at least $2\times$

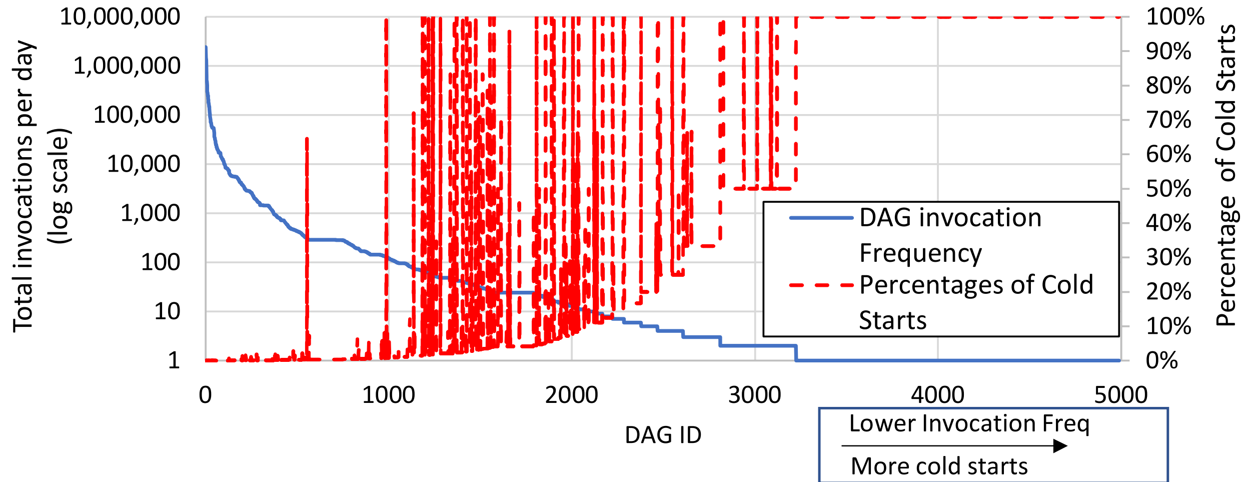


Figure 6.7. DAG invocation frequency (Blue solid) and its impact on the percentage of cold starts (Red dashed). DAGs with low invocation rate (*e.g.* once per day) always experience a cold start, whereas very frequent DAGs (*e.g.* $\geq$ 500 invocations per day) have very rare cold starts.

for 98.2% of the DAGs and increases as the width increases. This motivates the need for a mechanism to mitigate latency skew of parallel worker functions to reduce the E2E latency.

6.3.2 Performance Modeling

Modeling Latency as a Distribution rather than a Single Statistic. To estimate the E2E latency and cost of a serverless DAG, it is essential to model the latency of each component as a distribution. For example, the latency of the image classification function (`Classify-Frame`) in the Video Analytics DAG can vary by up to 2× when processing different frames, even when keeping the VM-size fixed to 1 GB. Although similar performance variability can be observed in server-centric platforms, our model is geared to serverless platforms due to their ability to scale resources according to demand virtually unboundedly and hence showing negligible queuing times [171]. Now consider a simple stage of two `Classify-Frame` functions running in parallel. Let X and Y be random variables representing the latency of each. The E2E latency of the two workers combined is given as $P(Z \leq z) = P(X \leq z, Y \leq z)$, which depends on the slowest of the two and hence knowing their median or even their P99 latencies is not sufficient to estimate their combined CDF.

In fact, we need the *entire* distribution of *both* components to estimate the E2E latency distribution. Moreover, simply using statistical tail bounds is not suitable for our purpose. For example, Chebyshev's inequality uses the mean and variance to establish a loose tail bound and it is not known how to combine tail bounds for in-series and in-parallel functions with their correlations.

Modeling Correlation. To accurately estimate the combined latency distribution for in-series or in-parallel functions, we need to capture the correlation between their execution times. *Ignoring correlation by assuming statistical independence leads to over-estimating the combined distribution for in-parallel functions, while it leads to under-estimation for in-series functions.* With perfect correlation, the N parallel functions will all finish at the same time. At the other extreme, if they are independent, the last function to finish determines the overall latency. With perfect correlation, if the latency of the first function is long, the latency of second function is also long increasing the combined latency. In our evaluation, we give quantitative evidence of these effects (§ 6.6.3) and show that a performance model that is distribution-agnostic (*e.g.* SONIC [172]), or correlation-agnostic (*e.g.* [180]), fails to provide accurate E2E latency estimates.

6.4 Design

We first describe the performance model and E2E latency estimation. Then, we describe how we use the performance model to perform our three optimizations.

6.4.1 Modeling E2E Latency Distribution

Modeling Functions Runtimes. We represent the runtime of a function as the sum of its initialization and execution times. Since both phases have a high variance, we represent them as separate distributions. This separation allows us to estimate the gains from each optimization. Allocating the right resources and bundling mainly impacts the execution times, whereas pre-warming reduces initialization times.

Combining Latency Distributions. Given a latency distribution for every function, Orion applies a series of statistical operations to estimate the DAG E2E latency distribution.

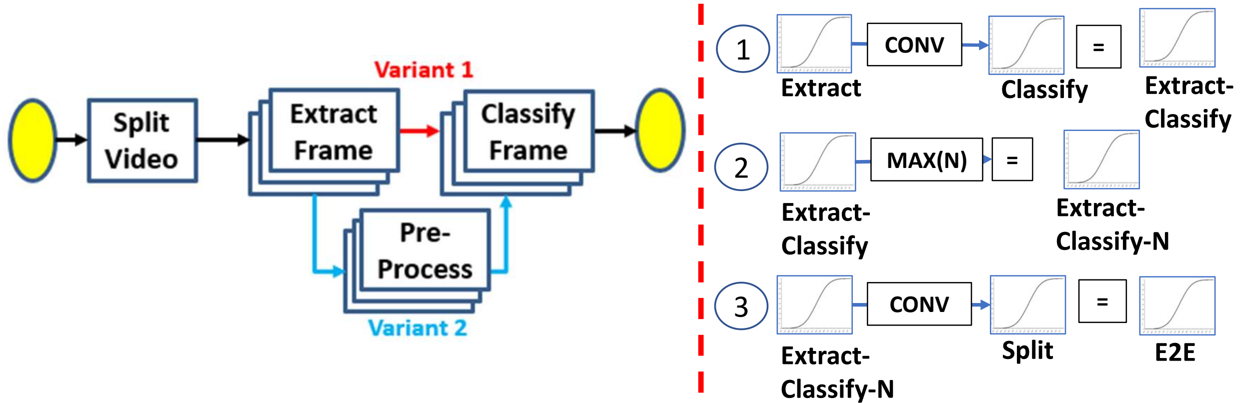


Figure 6.8. Video Analytics DAG. `Split` function downloads input video and splits it into chunks. Each chunk is passed to an instance of `Extract` function — extracting a representative frame, sent either to `Classify` function (Variant #1), or sent to *Pre-process* function and then `Classify` (Variant #2). The steps of estimating E2E latency distribution for Variant #1 are on the right.

For two in-series functions with latency distributions represented as random variables X and Y, we use `Convolution` to estimate their combined distribution as: $P(Z = z) = \sum_{\forall k} P(X = k, Y = z - k)$.

If X and Y are independent, we simplify the computation: $P(Z = z) = \sum_{\forall k} P(X = k) \cdot P(Y = z - k)$.

The latter is simpler to estimate since it only requires the marginal distributions of the two components, which can be profiled *separately*, rather than *jointly*.

On the other hand, if the two functions execute in parallel, then their combined latency distribution will be defined by the max of the two. Therefore, we use the `Max` operation to combine their CDFs as follows: $P(Z \leq z) = P(X \leq z, Y \leq z)$.

Similar to the `Convolution` operation, a simpler form can be used when X and Y are independent: $P(Z \leq z) = P(X \leq z) \cdot P(Y \leq z)$, which uses the marginal CDFs of the two functions, rather than their joint CDF.

Handling Correlation Among Functions.

We consider two types of statistical correlation in the DAG: in-series and in-parallel correlation. For example, our Video Analytics application (Figure 6.8) has high correlation between *Pre-process* and *Classify* stages, and also high correlation between parallel *Extract* invocations or parallel *Classify* invocations. (Table 6.1). By analyzing the correlation be-

tween the stages as well as the correlation between the parallel invocations in the same stage, ORION identifies both types of correlation. We consider a Pearson correlation coefficient value $>$ experimental parameter θ, as an indication of correlation. This then determines whether to apply the independent or dependent formulation for the `CONV` or the `MAX` operation. In our experiments, we find that the performance of ORION is relatively insensitive for $\theta \in (0.2, 0.5)$ and we run all the experiments with $\theta = 0.4$.

To determine the length of correlation chains (pairwise, etc.), ORION uses conditional entropy measurements of the execution time and compares the reduction in entropy by including additional terms. Thus, if marginal entropy of stage Y is $H(Y)$ and:

$$H(Y) \gg H(Y \mid X_{\mathrm{i}}) \approx H(Y \mid X_{\mathrm{i}}, X_{\mathrm{j}}) \approx H(Y \mid X_{\mathrm{i}}, X_{\mathrm{j}}, ..., X_s) \tag{6.1}$$

where X_{i} denotes the random variable of invocation i's execution time, then ORION infers that correlations across stages are at most pairwise. We find that correlations in all our application DAGs *across* stages are at-most pairwise. Our formulation, however, can handle any degree of correlation, not just pairwise. Only the amount of profiling data needed will increase with higher degrees of correlation.

In case of high correlation between N parallel invocations in the same stage, the `max` operation can be expanded by the chain rule:

$$\begin{aligned} P(Z \leq z) = & P(X_1 \leq z) \\ & .P(X_2 \leq z \mid X_1 \leq z) \\ & .P(X_3 \leq z \mid X_2 \leq z) \\ & .... \\ & .P(X_N \leq z \mid X_1 \leq z, X_2 \leq z, \ldots, X_{N-1} \leq z) \end{aligned}$$

which is further simplified in case of pairwise correlations by only conditioning on one invocation, hence all conditional terms reduce to: $P(X_{\mathrm{i}} \leq z \mid X_{\mathrm{i}-1} \leq z)$.

Since all components within a stage are identical, we estimate the above equation as follows:

$$P(Z \leq z) = P(X_{\mathrm{i}} \leq z).[P(X_k \leq z \mid X_{\mathrm{i}} \leq z)]^{n-1} \quad , k \neq \mathrm{i} \tag{6.2}$$

Therefore, we use two distributions to model the `max` for *any* number of parallel components — the marginal distribution and the conditional distribution.

In practice, all individual components are used to estimate the marginal distribution and all pairs of components are used to estimate the conditional distribution, as all marginal distributions are identical and so are all conditional distributions.

Using this performance model, ORION designs three optimizations for serverless DAGs, which we describe next — ***Right-sizing*** in § 6.4.2, ***Bundling*** in § 6.4.3, and ***Right pre-warming*** in § 6.4.4.

6.4.2 Allocating the Right Resources

The target of this optimization is to assign the right resources for each function in the DAG so that the entire DAG meets a latency objective with minimum cost. Normally, the user picks the VM-size for each function, and the VM-size controls the amount of allocated CPU, memory, and network bandwidth capacities. What makes this problem challenging is twofold — the scaling of multiple orthogonal resources is coupled together and the scaling of different resources is not linear. As an example, the `Classify-Frame` function in the Video Analytics DAG has a small memory footprint (540 MB). However, increasing its VM-size above 1,792 MB (as that size comes with a full vCPU [181]) reduces latency. This is because larger sizes come with a fraction of a second core up to six cores, which this function utilizes. The first step in this optimization is to map a given configuration candidate (*i.e.* a vector of VM-sizes, with one entry per stage) to the corresponding latency distribution. To achieve this, we build a per-function performance model that maps VM-sizes to latency distributions. Next, we combine these distributions to estimate the E2E latency distribution.

Per-function Performance Model. For each function in the DAG, we collect the latency distributions for the following VM sizes: *min* (the minimum VM size needed for this function

Algorithm 1 Best-First algorithm to identify the best VM sizes for functions in a DAG given a user-defined latency objective.

Input: Latency-Percentile=P, Latency-Objective: T_O
Output: Best VM sizes=S_{best}

```
 1: ## Initialize priority queue pq, performance model GetLatency, cost
    model GetCost, StepSize = 64 MB
 2: ##Set start state S_0 to minimum VM size for every function in DAG
 3: pq.insert(s_0)
 4: while pq is not empty do
 5:     S_next = pq.pop()
 6:     ## Create N new states by adding StepSize to each function
 7:     ## Set the priority of each state and add to pq
 8:     for i = 0-> |S_next| do
 9:         S_new = S_next
10:         S_new.VMsize[i] = S_next .VMsize[i] + StepSize
11:         S_new.latency = GetLatency(S_new, P)
12:         S_new.priority = -1 × S_new.latency × GetCost(S_new)
13:         pq.insert(S_new)
14:         ## Check if latency objective is met
15:         if S_new.latency ≤ T_O then
16:             return S_best = S_new
17:         end if
18:     end for
19: end while
20: ## If no explored state meets the latency objective
21: return State S_best with closest latency to T_O
```

Figure 6.9. Algorithm 1

to execute), 1,024 MB, 1,792 MB, and *max*. We pick 1,024 MB as it is the point of network-bandwidth saturation (increasing VM-size beyond it does not provide more bandwidth), and 1,792 MB as it comes with one full CPU core. Hence, this initial profiling divides the configuration space into 3 regions: [Min, 1024), [1024, 1792), and [1792, Max].

In order to estimate latency distribution for intermediate VM-sizes, we use percentile-wise linear interpolation. For example, the P50 for 1408 MB is interpolated as the average between the P50 for 1,024 MB, and the P50 for 1,792 MB settings. This generates a *prior* distribution for these intermediate VM-size settings. To verify the estimation accuracy in a region, ORION collects a few test points using the midpoint VM-size in that region to measure its actual CDF (*i.e.* the *posterior* distribution) and compares it with the *prior* distribution. If the error between the *prior* and *posterior* CDFs is high, ORION collects more data for the region midpoint and adds it to its profiling data. In summary, this approach divides the space of a potentially non-linear function into a set of approximately linear functions, and hence, more complex functions get divided into more regions, with a profiling cost overhead. In practice, we find that 5 to 6 regions accurately model the latency distributions for all functions in our applications.

Optimizing Resources for a Latency Objective. Since the performance model estimates the E2E latency distribution of the DAG, we can use it to choose a configuration (*i.e.* the set of VM sizes) to execute a DAG in order to meet a user-specified latency objective while reducing $ cost. We search for the configurations using a heuristic based on Best-First Search (BFS) [182]. The pseudo-code is shown in Algorithm 6.9.

The algorithm starts by creating a priority queue, in which all the new states are added. A state here represents a vector of VM sizes, one for each stage. Each new state expands the current state in one dimension (with a step-size of 64 MB) and the start state S_0 has the minimum VM size for every function. The priority is set to be the difference between the target latency and each state's estimated latency multiplied by the $ cost of the new state (lower value means higher priority). Our chosen heuristic is suitable for our problem because latency is a monotonically non-increasing function of resources allocated to a function. The worst-case complexity of BFS is $O(n * log(n))$, where n is the number of states.

6.4.3 Bundling Parallel Invocations

Stragglers can dominate an application's E2E latency [183]–[186]. Here we show how to bundle multiple parallel invocations in one stage within a larger VM, rather than the current state-of-practice of executing each in a separate VM. This promotes better resource sharing, thus mitigating skew.

To understand how bundling works, consider the example shown in Figure 6.10 for a stage of 4 parallel worker functions experiencing load imbalance. Specifically, the load for workers #2 and #4 is low and both require only one time step of execution. In contrast, the load for workers #1 and #3 is higher and requires 7 and 3 time steps of execution, respectively. Also, assume that the workers are scalable and can actually make use of additional resources made available. If we execute each worker function on a separate VM, say with 1 core each, the E2E latency is dominated by the slowest worker and the entire stage will take 7 time steps. However, if we bundle the workers together in a single VM with 4 cores, the E2E latency reduces to 3 time steps only. This is because the straggler workers get access to more resources when the lightly loaded workers finish their execution. Notice that the cost remains the same in both cases because they consume 1 core × 12 time steps or 4 cores × 3 time steps for the entire stage.

We make a few observations about the applicability of bundling. *First*, bundling is useful in reducing the latency if the execution skew is due to *load imbalance*, which arises from processing bigger partitions of data, or inputs that require more computation. We detect load imbalances due to content by subtracting latency CDF #1 from #2: (#1) When the function is executed multiple times with the same input. (#2) When the function is executed multiple times with different inputs. Moreover, the higher the correlation between workers, the lower the gap between their execution times, and hence, the lower is the benefit from bundling.

Second, for bundling to be useful, the function has to be *scalable* to benefit from the additional resources. We identify a function's scalability using our performance model to estimate the impact on the function's latency CDF when given more resources (§ 6.4.2).

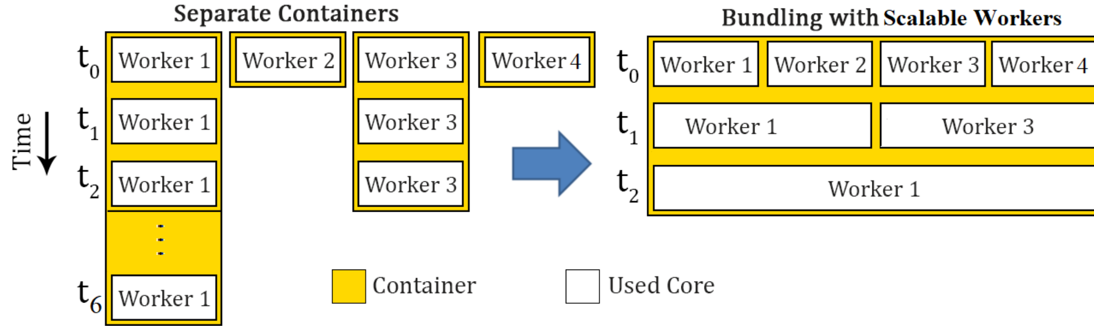


Figure 6.10. **(Left) Separate VMs**: workers #2 & #4 finish early, while workers #1 & #3 take longer. **(Right) Bundling**: after workers #2 & #4 finish, workers #3 & #4 get more resources reducing stage latency.

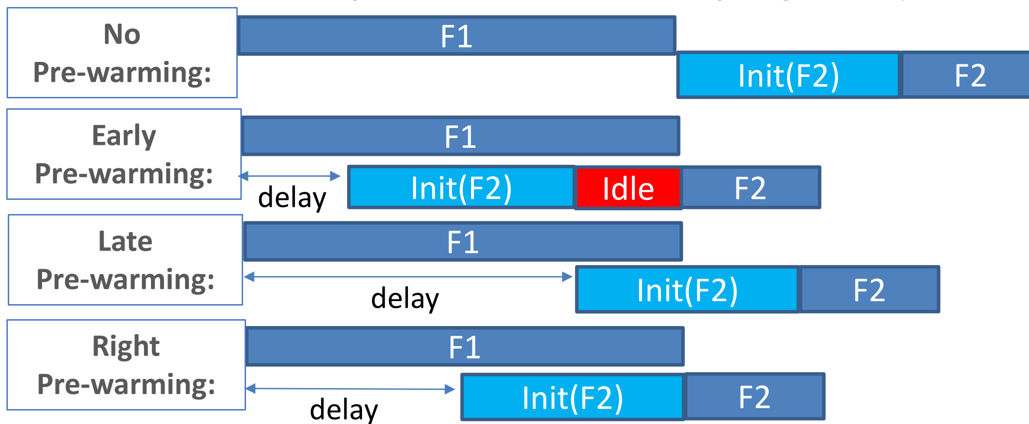


Figure 6.11. Impact of different pre-warming decisions on the E2E latency and utilization for a chain of two in-series functions. Without pre-warming, the E2E latency increases due to added initialization time of function F2. Both early and late pre-warming are not desirable.

We benefit from the fact that the community has developed many highly scalable libraries, *e.g.* [187], which are widely used in serverless applications.

Third, our example in Figure 6.10 assumes there will be *no contention* between bundled workers. However, in practice, we find that this contention can be high, especially for network-bound or IO-bound functions, as these resources do not scale linearly with the VM size. For example, all VM sizes in AWS Lambda get the same disk space of 512 MB and network bandwidth scales only for VM sizes until 1,024 MB.

Based on these three requirements, ORION identifies the best bundle size in two steps. First, ORION identifies functions that experience execution skew and are scalable using the performance model. Second, ORION searches the space of bundle sizes through multiplicative

increase (*i.e.* bundle sizes of 1, 2, 4, etc.). At each step, ORION collects very few profiling runs (we use 10) to capture contention. The search terminates when bundling more workers causes contention and hence increases the E2E latency. ORION strives to spread stragglers across different VMs, by performing a "redistribute" operation if needed, so that each straggler has excess resources to speed up its execution. Accordingly, when ORION detects when there is an excess of stragglers within a bundle, it performs a "re-distribute" operation. Since skew often shows up with temporal locality, we spread the parallel functions among the available bundles in a round-robin manner. For example, for the Video Analytics application, load typically varies gradually across consecutive frames.

ORION's security considerations: ORION does not currently bundle functions in different stages for security purposes. Moreover, all the invocations to be bundled together belong to the same user and the same DAG invocation. Additionally, in cases when the stages have very different resource requirements, it becomes counter-productive to come up with one VM size that fits multiple stages. We defer the possibility of bundling invocations across different functions as future work.

6.4.4 Pre-warming to Mitigate Cold Starts

We describe our approach to mitigating cold starts, leveraging the DAG structure of the application. We describe how to identify when to start pre-warming the VMs for each stage in the DAG, in order to balance the E2E latency and the utilization of the computing resources. This step is performed after we perform the previous two optimizations: Right-sizing and Bundling. Figure 6.11 shows conceptually the impact of different pre-warming delays on E2E latency and utilization. At the extreme, a delay of zero for every stage minimizes the E2E latency but also minimizes the utilization. The other extreme is no pre-warming at all, which is the state-of-practice. First, we define `pre-warming delay` for a stage S as the time elapsed between the start of the DAG execution and the beginning of initialization of the VMs for that stage. For a given DAG of N stages, we want to select a vector $\vec{d} = [d_1, d_2, ...d_N]$ representing the pre-warming delays for each stage in the DAG. For the first stage in the DAG, we have the degenerate case and set its delay (d_1) to zero.

This is because pre-warming requires predicting when the DAG will be invoked, which is challenging in the general case. The optimal delay vector, given a performance model $\mathcal{P}$, is defined as follows:

$$\begin{aligned} \vec{d^*} = \underset{\vec{d}}{\arg\min} \;\; & \textit{E2E-Latency}(\mathcal{P}, \vec{d}) \\ \text{subject to} \quad & Util(\mathcal{P}, \vec{d}) \geq \text{Target Utilization} \end{aligned} \tag{6.3}$$

The selected vector is the one that minimizes the DAG E2E latency while achieving the target resource utilization as set (and dynamically adjusted) by the provider. Both the utilization and the E2E latency are estimated by our performance model $\mathcal{P}$. The metric $Util(\mathcal{P}, \bar{d})$ measures the utilization for a given delay vector using the performance model, and is estimated as $\frac{BusyTime(VM)}{BusyTime(VM)+IdleTime(VM)}$. $BusyTime(VM)$ includes both initialization and execution times, while $IdleTime(VM)$ is the time between when the initialization completes to when the function starts executing. We again use Best-First Search (BFS) to select vector $\vec{d^*}$ as follows. We start by setting all values of $d_i = 0$. In each iteration, we add a delta (100 ms) to the delay factor d_i that yields the best improvement in utilization over the current state without increasing the E2E latency. The algorithm terminates when adding delta to any delay factor does not improve utilization but increases E2E latency.

6.4.5 Further Design Considerations

Deployment. ORION is designed to serve as a DAG optimization layer. Although the primary use case is to be deployed by the provider, ORION can also be deployed by end-users. In the latter, users are able to select the VM sizes for their functions. For this, the end-user will need to profile her code and also send pre-warming requests to the provider at the right times, as identified by ORION. However, users do not need to change their function code for Bundling. Instead, ORION identifies the bundle size for each parallel stage in the DAG and executes multiple invocations together. The cloud provider still decides the mapping of specific VMs to function bundles.

Naturally, ORION's performance model is trained faster for functions with higher invocation frequency as they provide natural training data points. As discussed in 6.3.1, frequently

invoked DAGs dominate the total set of DAG invocations. For example, the 5%-most frequent DAGs have an invocation rate of 2.3K per day. Hence, it will take us less than 3.5 hours to gather 300 training samples per function. We can accelerate this, and also handle less frequently invoked DAGs, by inserting synthetic but realistic DAG invocations to generate training data points. It is also possible that we will have to re-train our models from time to time when the workload characteristics have changed significantly, or less commonly, the application DAG or the infrastructure characteristics has changed significantly. This incremental training is *not* a computationally heavy task as it involves updating only *parts* of the distribution curves. Maintaining the latency data for performing such updating is also *not* a memory-heavy task.

6.5 Implementation

We implement ORION in C# and Python 3.8 with 2,100 LOC. We execute the serverless DAG applications in AWS Lambda and use Amazon S3 for data passing between the functions. We use AWS Step Functions [188] to orchestrate the DAG. Function bundling is implemented without any code change by using a wrapper around the (developer-provided) entry point to each function. We use the Python multiprocessing library [189] to execute bundled invocations together.

Runtime Overhead. In theory, the worst-case runtime of Algorithm 6.9 increases exponentially with the number of stages in the DAG. However, we find that ORION's BFS algorithm has a very low overhead in practice: Each iteration in Algorithm 6.9 takes [3,7.5] msec, and the best solution takes between 6 and 88 iterations while exploring $< 1\%$ of all possible states. The number of iterations depends on the latency target, the steepness of the latency-VM size relation, and the step size (we use 64 MB). For finding the best pre-warming delays, BFS takes between [0.4,3] seconds across all applications.

Scalability. We evaluate the scalability of ORION in Figure 6.19 with respect to increasing the number of stages. We synthetically replicate the last (and most time consuming) stage of the Video Analytics application to create a DAG of up to 8 stages. The overhead is defined as the inference time divided by the application lifetime, and it ranges between 0.12% for

3 stages to 0.07% for 8 stages. Also, increasing the number of stages, the prediction error increases, but slowly. Specifically, with up to 8 stages, P50 error is stable, and P90 and P95 increase slowly but never reach 15%. With wider DAGs, our inference time remains unchanged as the fanout degree is used as a parameter in our estimation of MAX (Eq. 6.2).

Pre-warming. Ideally, implementing pre-warming in AWS Lambda requires our control over assigning invocations to warm containers or VMs. Since we do not have such control, we rely on AWS Lambda's container reuse to implement pre-warming. Specifically, we perform pre-warming by sending a dummy call to a function, then send the actual call right after the response from the dummy call is received. Since AWS Step Functions does not allow such orchestration, we implement our own pre-warming-aware orchestration using two threads that communicate in a producer-consumer pattern. The producer thread sends dummy calls for pre-warming, and then generates tokens for the consumer thread that sends the real functions calls.

6.6 Experimental Evaluation

We evaluate Orion running on AWS Lambda. First, we describe the three serverless DAG applications used in our evaluation. Then, we show an E2E evaluation compared to three alternatives. Next, we show a set of microbenchmarks to evaluate each component of Orion. Finally, we provide a unit experiment on Azure Functions, a platform that allows less configurability for an external mechanism like Orion.

6.6.1 Serverless DAG Applications

Video Analytics. This application, adopted from Pocket [114], analyzes an input video by extracting representative frames from the video and classifying each frame. The application stages are shown in Figure 6.8. The first variant of this application directly calls a third function, *Classify-Frame*, which uses a YOLO [190] pre-trained DNN model to classify object(s) in the frame into 1,000 classes. The second variant calls an intermediate pre-processing function, *Pre-process*, which applies a sharpening filter to improve image quality before classification. We refer to this variant as "Video Analytics w/ Preprocess" (VA-Pre, for short).

Finally, all classification results are uploaded to remote storage. For VA-Pre, there is a high correlation between the times of the *Pre-process* and the *Classify* functions. We use 600 YouTube videos (300 for profiling, 300 for testing), each of length 1 min, belonging to the "Nature" and "News" categories.

ML Pipeline. This application is a machine learning pipeline (adopted from Cirrus [115]).It consists of three stages: dimensionality reduction (PCA), model training, and testing (*Combine*). The second function, *Train-Model*, runs in parallel and each instance trains a decision tree model using the *LightGBM* Python library [191]. In this stage, a user-specified number of functions is triggered (we use 64 trees in our evaluation), and every function trains a different decision tree. The third function, *Combine*, combines the trained models into a random forest and evaluates its joint accuracy on a held-out test dataset. We use the MNIST [144] database of handwritten digits that has a total of 60K images. We execute the application with 600 runs (300 profiling, 300 test), and in each run, we use 5K images to train the ML model, and 15K for testing.

Chatbot. This application trains a domain-specific Natural Language Understanding (NLU) model, whose task is to identify the accurate "intent" of a user-spoken utterance. We use the Chatbots Intent Recognition Dataset, available on Kaggle [176], which consists of 22 intents and 455 utterances. As before, we evaluate with 300 profiling and 300 test runs. The first lambda in this application parses the dataset and constructs bag-of-words representation for all utterances. Next, a stage of parallel lambdas trains One-vs-Rest classifiers with one lambda per intent. The models are then uploaded to remote storage for real-time intent detection.

The three applications cover important characteristics of serverless DAGs. Specifically, Video Analytics and Chatbot have scatter communication pattern, whereas ML Pipeline has a broadcast pattern. They also cover different fanout degrees (22 for Chatbot, 32 for Video Analytics, 64 for ML Pipeline) and their execution times resemble the average latency of DAGs in our workload characterization (§ 6.3.1). Moreover, Video Analytics and ML Pipeline are both compute bound, whereas Chatbot is network bound.

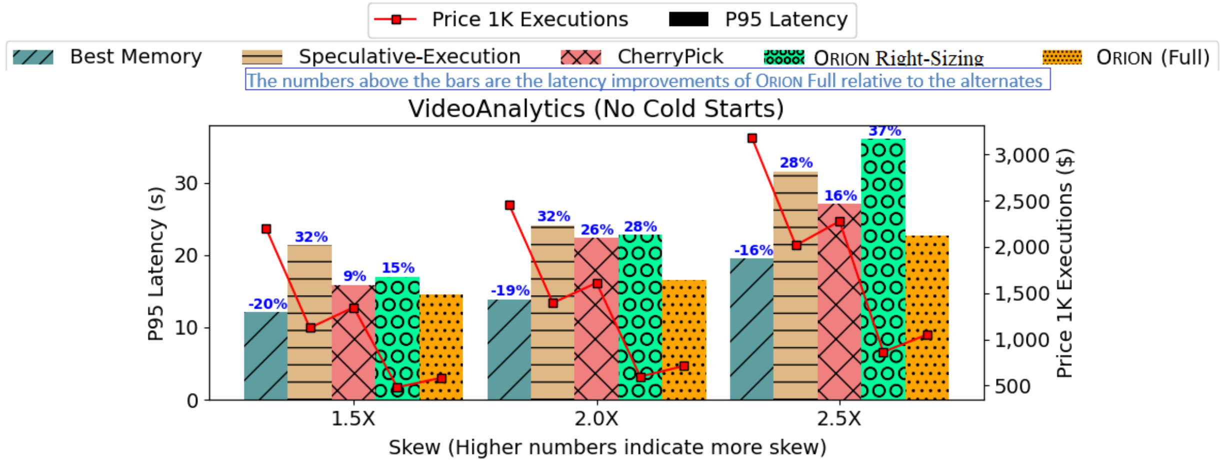


Figure 6.12. Skew is varied by changing the detection probability threshold as: 2%, 10%, and 15%, with lower values resulting in more detected objects and higher skews [192].

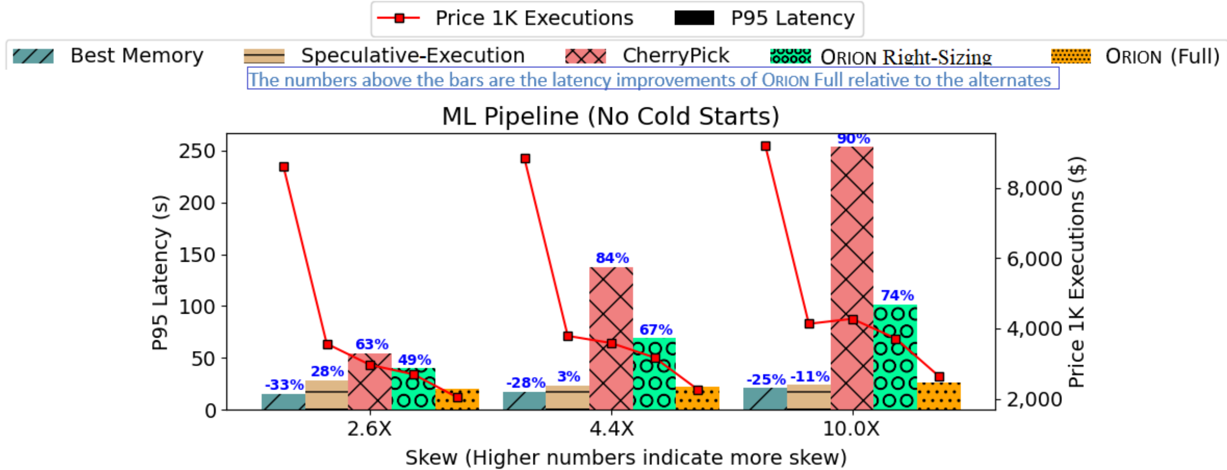


Figure 6.13. Skew is varied by changing the maximum value for each hyper-parameter, *e.g.* we vary the max number of trees as: 50, 100, and 200, and these map to 2.6×, 4.4×, and 10× skews.

6.6.2 Orion and Competing Approaches

We compare our E2E latency and cost to multiple resource allocation, skew mitigation, and pre-warming approaches:

(1) **Best-Memory:** This is a resource allocation approach that uses our performance model and progressively increases the VM size for every function in the DAG till the latency objective is met. This mimics the standard VM autoscaling that is employed in many cloud

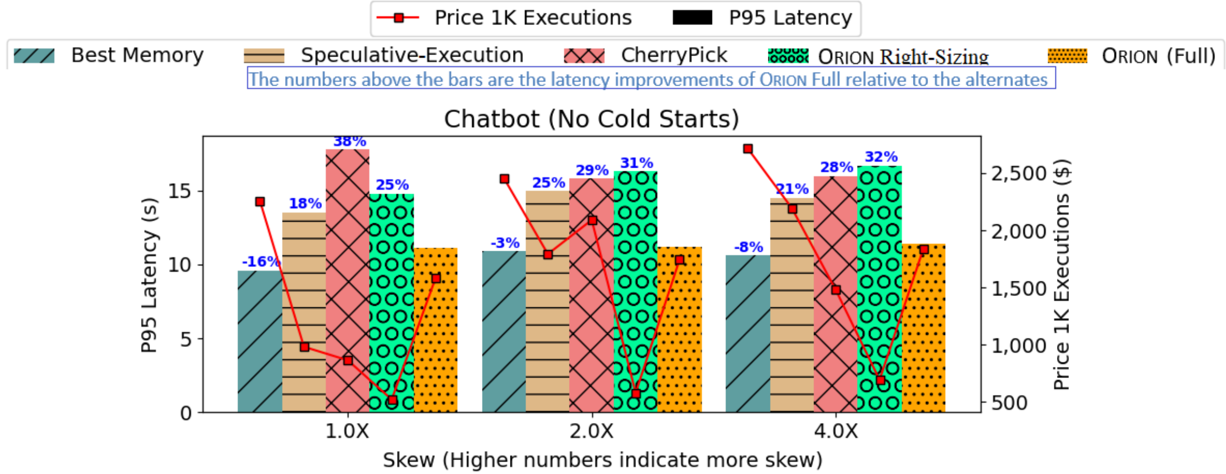


Figure 6.14. Skew is altered to 1×, 2×, and 4× by changing the number of training epochs as: 100, 500, and 200.

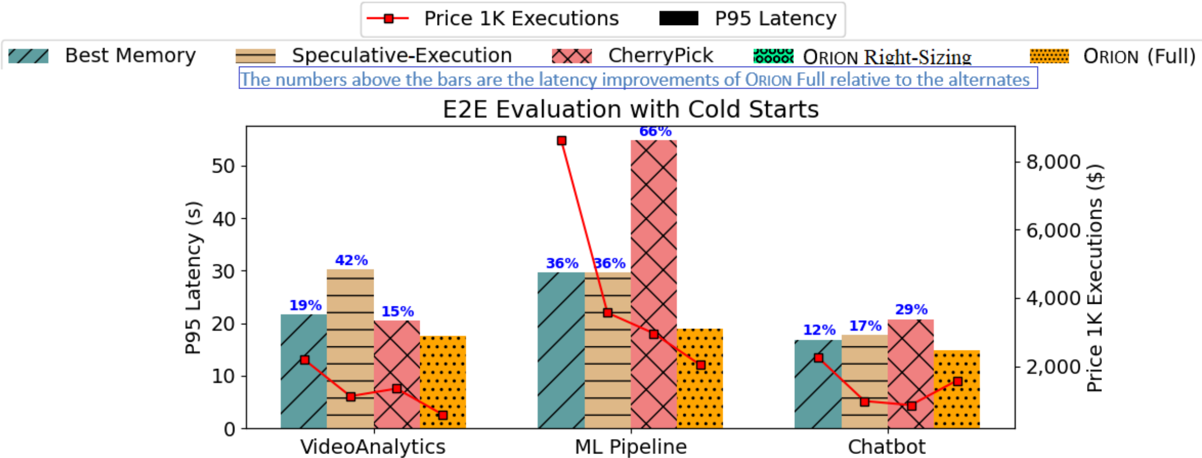


Figure 6.15. E2E evaluation with cold starts. ORION achieves the lowest latency and cost compared to all baselines.

scheduling solutions [177], [178]. All invocations run in separate VMs of the same size.

(2) **CherryPick [72]:** CherryPick uses Bayesian-Optimization (BO) to find latency-optimized memory configurations. BO relies on an *acquisition* function to propose new points to sample next. This makes BO a distribution-agnostic baseline as each VM size is profiled once, then a new size is selected by the acquisition function. We set the loss function in BO to be the difference between the achieved latency and the user-specified latency target. Since CherryPick is distribution agnostic, it cannot detect execution time skew and hence performs no bundling. We choose CherryPick as it (BO with Gaussian processes) was recently

demonstrated as the most competitive approach in the category [175] and recent approaches have used it for configuring serverless functions [174].

(3) **Speculative-Execution:** This is a skew mitigation approach that identifies stragglers at runtime and executes a duplicate invocation on a different VM. Speculative execution is widely used in MapReduce, Hadoop, and Spark systems for skew mitigation to reduce tail latency [184], [193]–[195]. We adapt this baseline from Spock [159] (specifically the technique called "conservative autoscaling in predictive mode"). Since the skew is caused by the input's content, the new invocation will likely take as long as the first one unless it is assigned more resources. Accordingly, we modify the technique by assigning the "Max" resources (10 GB) for the new invocation.

(4) ORION **Right-Sizing:** This variant of ORION performs Right-Sizing only, and not the other two optimizations.

(5) ORION **Full:** This is our complete solution, which includes Right-Sizing, Bundling, and Pre-Warming.

6.6.3 End-to-End Evaluation

We show the P95 latency (primary Y-axis, shown using bars) and cost (secondary Y-axis, shown using lines) of each approach in Figures 6.12, 6.13, and 6.14 for the three applications. The numbers above the bars are the latency improvements of ORION Full relative to the alternates. First, we set the latency objective to the minimum achievable latency, which is identified by executing all functions with *max* VM size, while computation skew is minimized. For each application, we vary the skew in a controlled manner through application-specific parameters. For each solution and for each experimental point, we execute each application 300 times and highlight the gains of ORION's right sizing and bundling. In this part, we take care to eliminate all cold starts for the experimental data points. Later, we show the impact of cold starts and the additional gain due to ORION's pre-warming design in Figure 6.15. Compared to Best-Memory, ORION has a slightly higher latency since Best-Memory assigns high resources to all workers, including stragglers. However, this baseline increases the cost significantly by assigning identical resources for each stage and parallel running workers

in separate VMs, which over-provisions the resources to meet the latency objective. Orion provides [33%,71%] lower cost by assigning the right resources for each function and bundling parallel workers. Compared to Speculative-Execution, we notice that Orion has consistently lower latency and cost across all skews. For example, with the lowest skew, Orion shows [18%, 32%] lower latency and [46%, 57%] lower cost for the three applications. This is because Speculative-Execution detects straggling workers (using a user-specified threshold) and re-executes them on new VMs with the max size. This causes an additional delay due to the wasted execution time. It also increases cost as it sometimes mistakenly re-executes workers that would finish shortly after the threshold. Orion's bundling does not require any user-specified threshold to detect straggling workers, assigning them more resources once co-located workers finish and release their resources.

For CherryPick, since it is distribution-agnostic, we modify the BO algorithm so that 100 points are profiled for each point selected by BO's acquisition function to measure the latency percentiles. We run CherryPick for 100 iterations total (a generously high number compared to the original work and follow-on works), resulting in 10K profiles for each application. Notice that Orion requires only 300 profiling runs to model the E2E latency distribution, reducing the profiling burden of CherryPick by 97%. Compared to CherryPick, we notice that Orion Full consistently provides lower latency and cost, except for Chatbot where CherryPick has higher latency but lower cost. Specifically, with the highest skew, Orion Full shows [16%, 90%] lower latency and [38%, 53%] lower cost for Video Analytics and ML Pipeline. For Chatbot, this application has a lower bundle size than the others, reducing the gain from Orion's bundling mechanism. Compared to Orion Right-Sizing, adding bundling significantly reduces the latency across the three applications. However, bundling causes a slight increase in the cost for Video Analytics by 22% (relative to No-Bundling), while it causes a decrease in cost for ML Pipeline by 30%. The reason is that the ML Pipeline experiences higher skews (up to 10X), and for higher skew, Bundling is more beneficial. We also notice that the reduction in latency increases with higher skews. For Chatbot also, bundling reduces the latency compared to no bundling, but increases the cost by 163%. This is because the Chatbot application is more network bound and not compute bound than the other two, and hence the best bundle size is only 2, vs [6,8] for the other applications.

However, cost with bundling is still 33% lower than Best-Memory, which is the closest to us in latency among all baselines.

Mitigating Cold Starts with Pre-warming. So far, we have compared ORION to the baselines with only warm executions. Now we show the gain of our pre-warming technique and how useful it is in reducing cold starts. Figure 6.15 shows ORION's latency and cost vs other baselines in the case of cold start for every function in the DAG. We notice that all baselines are impacted by cold starts and their latencies increase, whereas ORION's pre-warming technique is able to mitigate the impact of cold starts. For example, Best-Memory shows an increase of E2E latency over ORION by 19%, 36%, and 12% for Video Analytics, ML Pipeline, and Chatbot, respectively. Similarly, Speculative-Execution suffers from cold starts twice, once for the first execution with the small VM, and once more for the second execution with the max VM size. Hence, ORION's improvements in latency over Speculative-Execution increase to 42%, 36%, and 17% for the three applications. To summarize, ORION's three optimizations of Right-sizing, Bundling, and Right pre-warming provide lower E2E latency and cost over all competing approaches. In the next section, we show a set of microbenchmarks to separately evaluate the performance of each component of ORION.

Evaluation of Performance Model

Capturing Correlation between Functions.

Here we evaluate how much correlation exists in our target applications. We calculate the Pearson's correlation coefficient between in-series functions (*e.g.* between *Split-Video* and *Extract-Frame*), and between in-parallel functions (*e.g.* between multiple instances of *Extract-Frame*). We show the correlation scores in Table 6.1 for Video Analytics.

The correlation scores between in-series components are close to zero (0.036 on average for Video Analytics, 0.06 for ML Pipeline, and 0.04 for Chatbot), while the correlation scores between functions in the same stage are high for Video Analytics (0.55), while low for ML Pipeline (0.052) and for Chatbot (0.03). For Video Analytics w/ Preprocess, `Pre-process` has a high correlation with in-series `Classify` functions (0.65). Therefore, we apply the dependent `conv` operation between `Pre-process` and `Classify`, while we use the

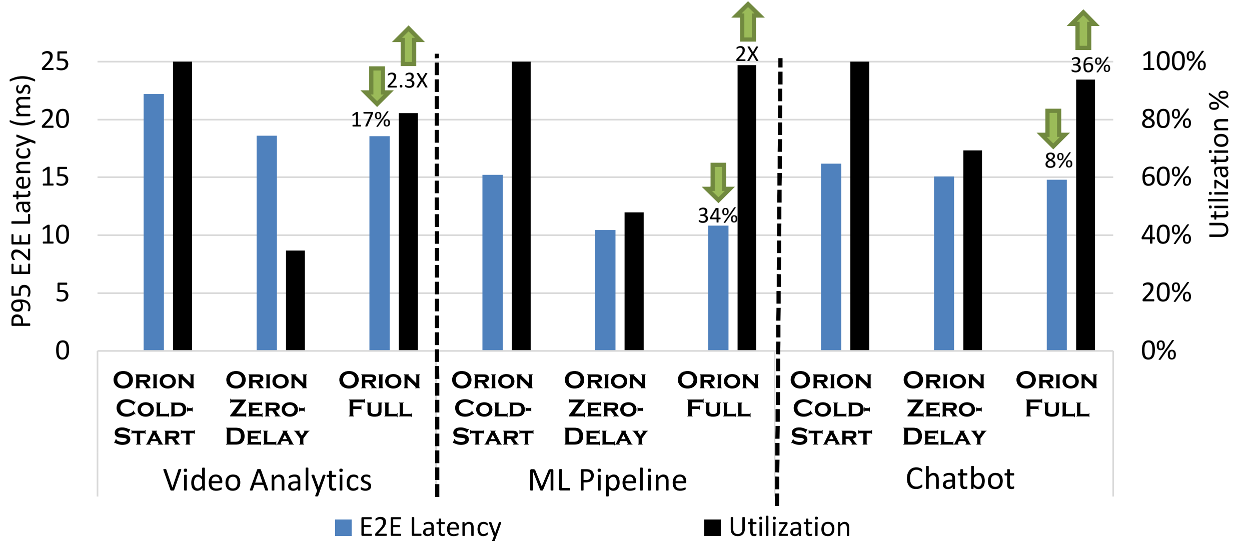


Figure 6.16. Impact of pre-warming on latency and utilization. We use VM and bundle sizes selected by ORION and compare different execution strategies w.r.t. cold starts. Percentages over the bars of ORION show the improvements in P95 Latency (over ORION Cold-Start) and Utilization (over ORION Zero-Delay pre-warming).

independent `conv` operation for all other in-series functions for all applications. Additionally, we incorporate correlation when performing `max` operation (if correlation is detected) by using the conditional distribution.

Estimating E2E Latency Distribution. Here we evaluate the accuracy of ORION in predicting the E2E latency distribution for the entire DAG. We compare to two baselines — distribution-agnostic (as mentioned earlier, any BO-based technique like CherryPick falls in this category) and correlation-agnostic (*e.g.* [180]). The results are shown in Table 6.2 for Video Analytics.

ORION estimates the E2E latency distribution for the applications with low error rate (<15%), much lower than those of both baselines. We find through drill down of our estimation error that: (i) our estimated length of correlation chains as pairwise (§ 6.4.1) is accurate and hence does not lead to much error (ii) the dominant source of error lies in the interpolation of the CDFs for each function for the *unseen* memory configurations. This is despite our design, where if the interpolation causes too much error, the memory region is split into two and further data points are collected (§ 6.4.2). These observations hold across all three

Table 6.1. Correlation between execution times of functions in the Video Analytics DAG. In-series correlation is low but in-parallel correlation is high.

VM-Sizes (in MB)	**In-series Correlation**			**In-parallel Correlation**	
Split, Extract, Classify	Split ⇕ Extract	Extract ⇕ Classify	Preprocess ⇕ Classify (VA-Pre)	Extract	Classify
192, 192, 576	0.09	0.04	0.45	0.05	0.43
1024, 1024, 1024	0.07	0.02	0.61	0.34	0.44
1792, 1792, 1792	-0.07	-0.04	0.69	0.48	0.58
3008, 3008, 3008	0.05	-0.01	0.88	0.65	0.51

Table 6.2. Video Analytics: Error rates for ORION's E2E latency estimation. Abbreviations: S→Split, E→ Extract, and C→Classify

Video Analytics						
VM Sizes (MB)	ORION		Distribution Agnostic		Correlation Agnostic [180]	
S, E, C	**P50**	**P95**	**P50**	**P95**	**P50**	**P95**
512 , 1280 , 1536	14.0%	13.0%	40.0%	15.6%	78.7%	47.5%
768 , 1280 , 2240	14.0%	12.0%	35.4%	11.6%	67.7%	38.3%
1536 , 512 , 1536	13.0%	11.0%	39.7%	16.7%	79.4%	49.9%
1792 , 1792 , 576	6.4%	11.8%	11.6%	-39.0%	49.7%	-18.2%
6000, 6000, 6000	14.5%	10.7%	24.2%	3.3%	56.9%	30.5%
MAPE	**13.0%**	**12.0%**	**32.0%**	**21.0%**	**68.0%**	**39.0%**

applications. Error rates are higher in Video Analytics relative to ML Pipeline because the execution time is content sensitive for the former. Our technique does *not* create content-specific models since we (and any provider-side tool) cannot have visibility into user data due to privacy concerns. The ***Distribution-Agnostic*** baseline uses the median execution times and predicts the median execution times for unseen configurations by interpolation. This baseline has a high error rate in the range of [-39%, 40%] for Video Analytics, [-5%,108%] for ML Pipeline, and [-6%, 66%] for Chatbot. The ***Correlation-Agnostic*** baseline from [180] also has a higher error rate in the range of [-18%, 79%] for Video Analytics, [-5.4%,103%] for ML Pipeline, and [80%, 111%] for Chatbot. Note that the majority of the errors for the ***Correlation-Agnostic*** baseline are over-estimation, which is caused by ignoring the

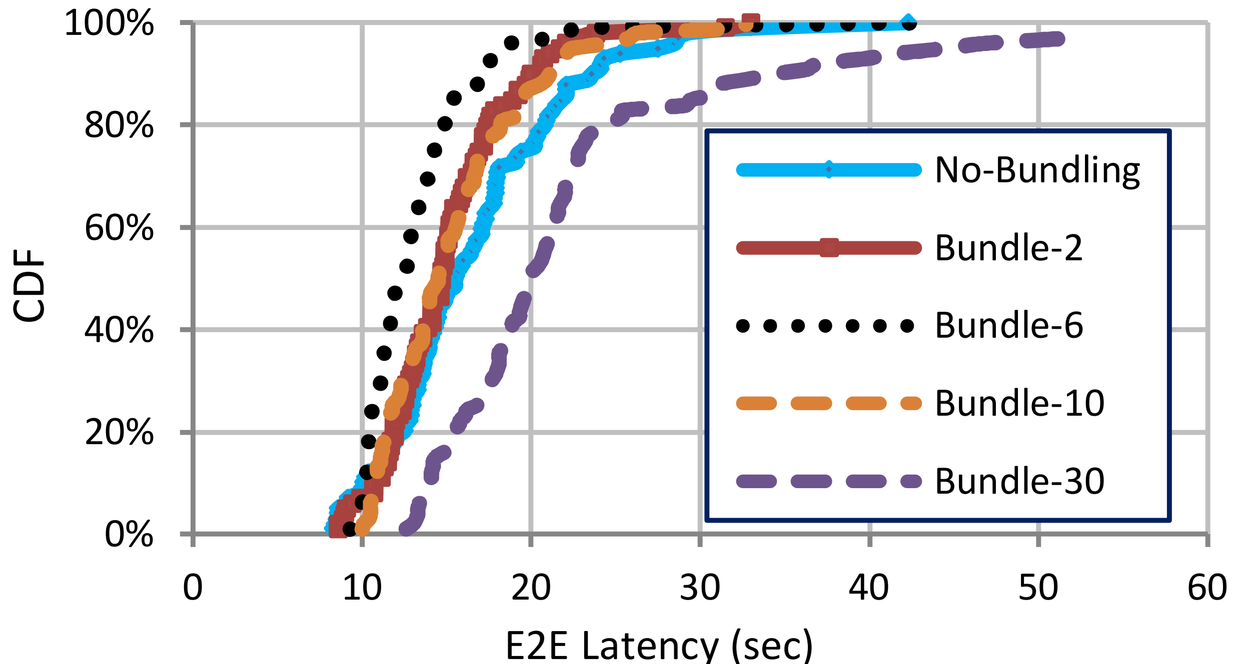


Figure 6.17. Video Analytics: Impact of varying bundle sizes. No-bunling has high latency due to computation skew. The optimal bundle size here is 6, and using a bundle size of ≥ 10 causes contention and the latency increases.

correlation between parallel workers. In conclusion, it is important to take into account the latency distributions and not simply a point estimate and to account for the correlation across stages and across workers within a stage, even when the correlations are quite weak (Table 6.1).

Optimizing Resources for a Target E2E Latency

We profile the applications to build the E2E performance model in ORION for all three applications as mentioned in § 6.6.3, then set 6 latency targets per application. ORION proposes the DAG configuration (*i.e.* VM size for each function in the DAG) to meet each latency target at while reducing cost. We validate ORION's accuracy by executing the application with the proposed configuration and comparing the achieved latency to the user requirement.We notice that ORION's proposed configurations are very close to the latency requirement in all applications, with error rate of [-2.75%, 4.93%] for Video Analytics,

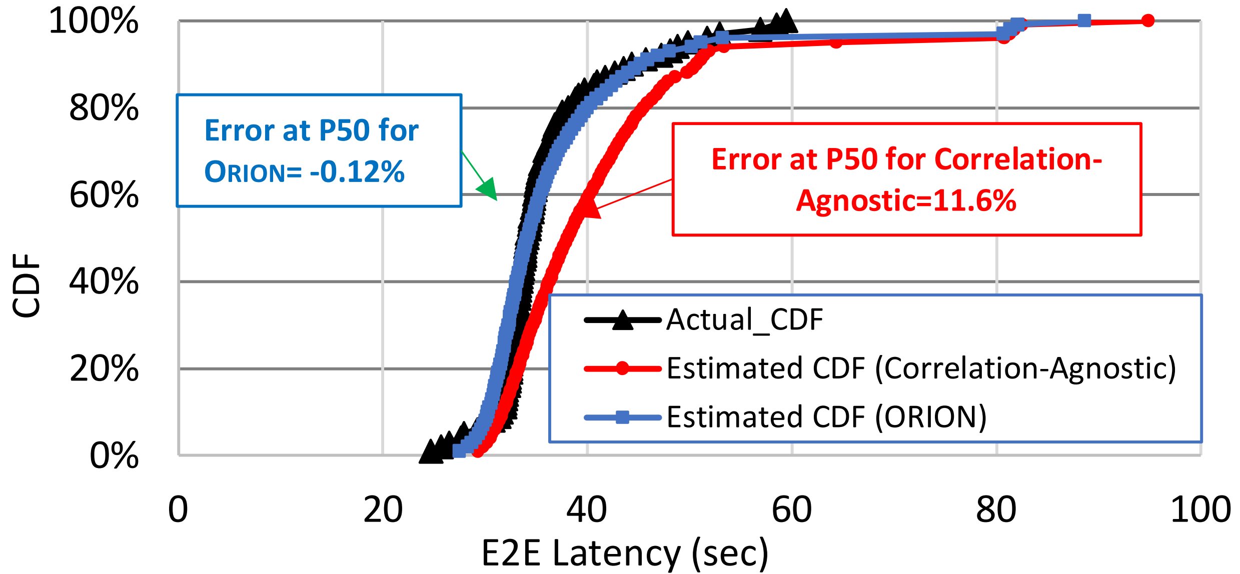


Figure 6.18. ORION's estimated latency CDF vs Actual CDF for Video Analytics application deployed in Azure Functions. Ignoring in-parallel correlation leads to higher errors for the Correlation-Agnostic baseline.

[-1.37%, 2.6%] for ML Pipeline, and [-3.5%, 3.7%] for Chatbot. Table 6.3 lists detailed configurations for Video Analytics. We notice that expectedly, ORION tends to assign more resources as the latency percentile increases (*i.e.* P50 $\rightarrow$ P95) or as the latency requirement decreases (50 sec $\rightarrow$ 30 sec). Also ORION decides to increase the allocated resources for a subset of functions and by different amounts, based on the latency requirement. For example, for ML Pipeline, ORION increases the VM-size of PCA from 768 MB to 832 MB to achieve a latency requirement of (P90 $\leq$ 50 sec). However, ORION decides to increase the VM-size of Combine from 1,408 MB to 1,472 MB to achieve a latency requirement of (P90 $\leq$ 40 sec). This shows ORION's BFS adjusts the *Best* function to increase its resources according to the estimated latency of the current state.

Impact of Varying Bundle Size

We evaluate the impact of varying bundle sizes on the E2E latency CDF and cost (Figure 6.17). First, we run our Video Analytics application with the best VM size selected by BFS but without bundling. This is an application that is both CPU bound and scalable,

Table 6.3. ORION's E2E latency-optimized VM sizes. ORION meets the latency objective with a low error rate in the range of [-2.75%, 4.93%]

Video Analytics			
User Requirement	ORION's configs (MB) Split,Extract,Classify	Achieved Latency	Error Rate
P50 ≤18 s	192, 192, 640	18.3 s	1.5%
P90 ≤18 s	256, 192, 768	17.8 s	-1.0%
P95 ≤18 s	384, 192, 768	17.5 s	-2.8%
P50 ≤17.5 s	192, 192, 704	18.4 s	4.9%
P90 ≤17.5 s	448, 192, 768	17.22 s	-1.6%
P95 ≤17.5 s	640, 192, 768	17.4 s	-0.5%
P50 ≤17 s	256, 192, 768	17.8 s	4.4%
P90 ≤17 s	832, 256, 1024	16.9 s	-0.57%
P95 ≤17 s	832, 256, 1024	17.3 s	2.0%

and thus a good candidate for demonstrating the effect of bundling. Next, we progressively increase the bundle size and the VM size proportionally. For example, if the best VM size selected by BFS is 1,792 MB (1 core), we use a VM of size 1,792 × 2 when we bundle pairs together, and so on. We notice that increasing the bundle size from 2 to 6 workers reduces the latency; however, increasing the bundle size beyond that (to 10 and 30) causes an increase in the latency. This is because the maximum number of cores available in AWS Lambda is 6 and hence, at the higher bundle sizes (10 or 30), each worker is getting less than its required resource.

Thus, the design of ORION to choose the best bundle size is essential to optimize latency by avoiding contention.

6.6.4 Generalizability to Microsoft Azure

To test if ORION generalizes to other FaaS providers, we evaluate our model using Azure Functions as the serverless environment. Azure Functions supports a few plans, but the most popular one is the *Consumption Plan*. In this plan, users are charged for the exact amount of resources consumed by their functions at runtime, whereas all other plans have a flat

rate pricing model. Although we have no control over the resources assigned to individual functions when selecting the *Consumption Plan*, we wanted to measure the accuracy of ORION's E2E latency estimates compared to the actual latency observed with this plan. We show ORION's estimated CDF and actual CDF in Figure 6.18. We use our Video Analytics application with the earlier-mentioned 600 YouTube videos.

We use our E2E performance model to estimate the CDF for the entire DAG. For fair comparison to AWS-Lambda, we also rely on remote-storage (*i.e.* Azure Blob Storage) for data-passing between the functions. We also show the estimated CDF when correlations among functions are ignored — this corresponds to the "Correlation-Agnostic" baseline from our earlier experiment (§ 6.6.3). We notice that ORION predicts the E2E latency with very low error rates (-0.12% for P50, 1.9% for P90, and 2.5% for P95 latencies). The Correlation-Agnostic baseline has significantly higher errors (11.6% for P50, 14.4% for P90, and 29.2% for P95). Thus, the baseline suffers more for higher percentiles.

6.7 Pre-warming Policy Simulator

To better understand different pre-warming policies without being constrained by privileges granted by the cloud provider, we build a policy simulator, implemented in Python 3.8 with 1,058 LOC. The simulator takes as input the latency CDFs for stages in the DAG. Policies are implemented through a state machine with different actions being taken in each state (such as FUNC_START, FUNC_END, FUNC_PREWARM, etc.). The output of the simulator are the E2E latency CDF of the DAG and the overall resource utilization. We open source the simulator for future exploration of serverless DAGs [196].

Simulation Results. Figure 6.20 shows the utilization achieved by a policy with optimal pre-warming using an Oracle that knows the exact runtimes of each function invocation. The input DAG has 2 stages with width of 10 for each stage. The X-axis denotes the skew on the runtime of the first stage. The Y-axis denotes the percentage of variance on the delay chosen by the Oracle for pre-warming functions of the second stage — so if the value is $X\%$ and ORION calculated deterministic delay is Y, then the Oracle can pick a delay in the range $[Y - X\% \text{ of } Y, Y + X\% \text{ of } Y]$. Thus, the range of values the Oracle can choose from is capped

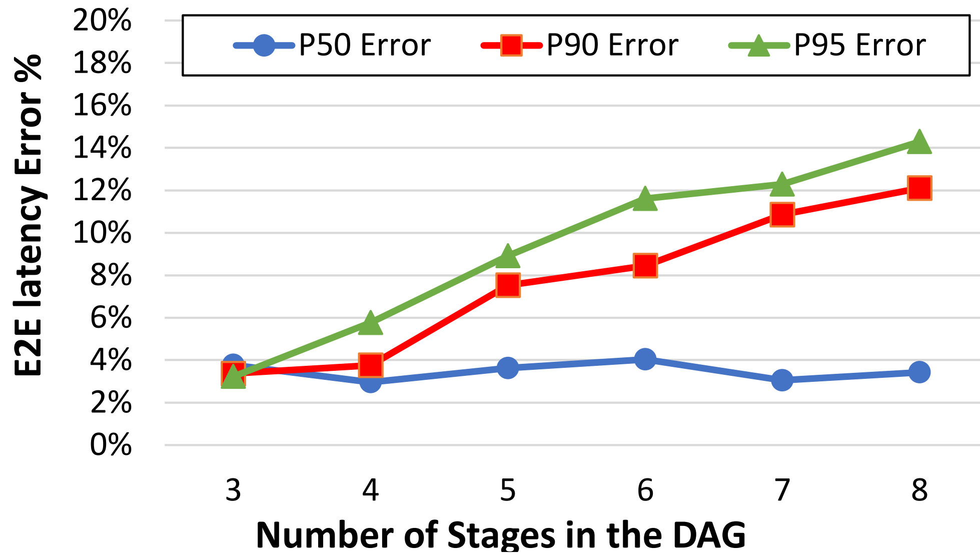


Figure 6.19. ORION's error with varying number of stages. More stages increase the error for the tail, while the median stays stable.

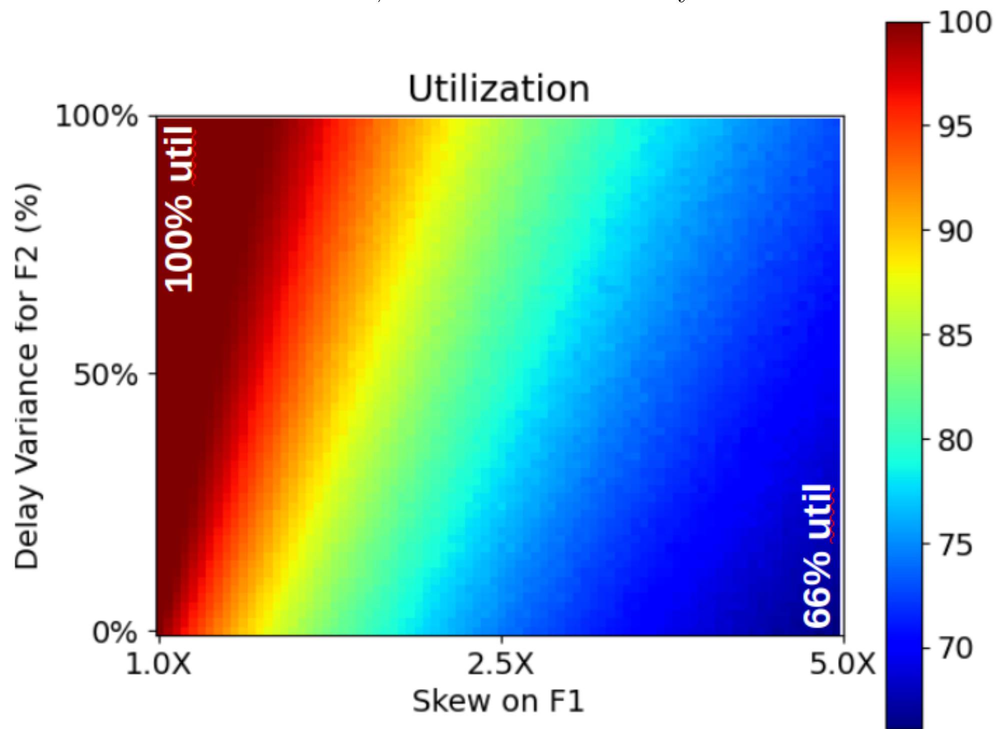


Figure 6.20. Simulation of an Oracle pre-warming policy where utilization improves with the width of distribution from which the pre-warming delays are chosen. ORION's strategy corresponds to the 0% variability, *i.e.* deterministic delay.

even if the Oracle determines the optimal pre-warming time for a specific function invocation lies outside of the range. The lowest point on the Y-axis is the optimal deterministic delay determined by ORION for all function invocations in the second stage. We find that the E2E latency is unaffected (not shown) by increasing the size of the range on higher skews, but utilization increases. This is because the policy is able to pre-warm with the ideal delay and hence does not incur any idle time. This shows the theoretical best achievable utilization since we use an Oracle. However, implementing this Oracle has two challenges: (1) Predicting per-function *exact* latency is impractical. (2) Selecting a delay factor for each function invocation rather than each stage increases search space exponentially with DAG width.

6.8 Related Work

Minimizing cost and/or execution time for serverless chains is the target of a few recent studies. For example, Sequoia [197] makes the observation that current serverless platforms treat functions within a DAG separately, without making use of the DAG structure. SONIC [172] reduces the communication latency between in-series serverless functions by optimizing the data passing strategy. SONIC selects from among the data passing strategies: direct passing, remote storage, and local VM-storage, where only the latter two can be implemented directly in AWS Lambda. Caerus [198] stresses the importance of optimizing latency and cost jointly for serverless DAGs, and achieves this by identifying pipeline-amenable data dependencies between stages to find ideal task launch times. Xanadu [199] and Kraken [200] tackle the problem of cascading cold starts in a dynamic DAG. Neither can determine the optimal pre-warming time to mitigate cold starts.

Overall, no prior work in this category considers execution time variance and its impact on cost or utilization.

Latency and Cost Prediction for Serverless Functions. A few prior studies have targeted predicting the execution time and cost for serverless functions. For example, [201] predicts (a point estimate) and optimizes resources for a *single* serverless function by build-

ing regression models from a host of synthetic functions. The authors in [180] also observe a variance in execution time in serverless environments, and hence, apply mixture density networks to predict the distribution of the function cost. However, their Monte-Carlo simulation mechanism is very sample inefficient.

Orion uses a more direct method by applying statistical operations to combine the distributions of individual functions and thus, to infer the E2E latency distribution. A number of prior works target reducing the cost of serverless DAGs by optimizing the intermediate data transfer between functions, such as, Costless [109], Sonic [172], Locus [99], and Pocket [114]. They solve an orthogonal problem to ours, namely, reducing the cost of intermediate data transfer. Orion does *not* introduce a new mechanism for intermediate data transfer, nor does it limit or specify the method for state transfer between functions. We use state-of-practice remote storages, such as AWS S3 and Azure Blob Storage. However, Orion would integrate seamlessly with the mentioned systems as the read/write times are included in the latency profiles used in Orion 's model.

Scheduling in Serverless Computing. Photon [202] optimizes *single-stage* serverless functions by doing the equivalent of bundling in Orion, but not for skew mitigation. Its main motivation is to reduce the memory footprint of parallel invocations of a function, while its design sophistication is meant to address security concerns of bundling (out of scope for Orion). One work that targets meeting latency SLAs for serverless DAGs is Atoll [203].

It takes a complementary approach to ours—partitioning a cluster to lower scheduling overheads, and proactively starting up containers and then routing function requests to the appropriate containers.

Resource Optimization in the Cloud. Black-box configuration tuning systems such as CherryPick [72], Selecta [73], OptimusCloud [138], and Ernest [204] target optimizing the cloud resources for a wide range of applications by selecting the right VM type and size, which vary in the amount of allocated resources. However, these systems treat the application as a single component, and thus, do not take the DAG workflow information into account. Further, they are not directly applicable to serverless applications.

Cold Starts Mitigation. Many prior works identified cold starts as a major performance bottleneck in FaaS platforms. Accordingly, several solutions have been proposed such as keeping containers alive [205], leveraging checkpoint/restore operations [206], or using *Pause* containers [207]. Although these solutions reduce the initialization time significantly, there is still a significant user-observable initialization time. ORION hides this initialization time through pre-warming and decides the right time to start pre-warming to minimize idle time, hence keeps resource utilization high.

6.9 Discussion

Profiling and Modeling Overheads. ORION requires monitoring the execution of the application for a number of runs to accurately capture the latency distribution for each function in the DAG. In our evaluation, for all the applications, a total of 300 profiling runs was found sufficient for accurate 95-percentile latency estimates. Initially, and before convergence is reached, the data collection is performed as a background process while the DAG executes with user-provided configurations. An important design consideration for ORION is that this data collection does not have to happen purely offline and in batch mode. Rather, that is complemented with online data collection and incremental model refinement, which is a lightweight task. When predicted and observed latencies differ significantly (as can happen if the workload or the application changes), we restart the data collection phase to capture the changes in the latency distributions.

Bundling and Performance Model Interaction. Bundling changes the DAG structure (by reducing the fanout degree), and hence, the performance prediction model needs to be updated. Therefore, this becomes an iterative process, with each iteration being *Performance model building* $\Rightarrow$ *Resource optimization* $\Rightarrow$ *Bundling*. In practice, we find that a single iteration, or at most two iterations, leads to convergence.

Impact of The Three Optimization. The three optimizations of ORION can have a negative impact on performance, resource utilization, or $ cost if not performed carefully. First, over-provisioning the VM size for all workers to mitigate execution skew (as done by the Best Memory baseline in our evaluation) unnecessarily increases the $ cost (Figures

6.12, 6.13,& 6.14). Second, excessive Bundling (bundle size > right bundle size) can lead to resource contention and increase of the latency (Figure 6.17). Third, early pre-warming (delay < right delay) decreases resource utilization, whereas late pre-warming increases latency (Figure 6.16). This motivates the need for an accurate performance model to accurately perform these three optimizations. In terms of cost, we notice that users do not pay for initialization times, hence pre-warming does not impact cost. However, the provider should treat a pre-warming request as a hint since a true invocation is always more important.

Applicability of Performance Model.

ORION is tailored to model the performance for serverless DAGs. In general, the response time of a job includes queuing and execution times. Cloud providers operate large serverless platforms, providing virtually infinite capacity, reducing queuing delays to primarily cold-start latencies [208]. Further, serverless platforms typically limit the execution time of each invocation [209] favoring modular reusable functions. The combination of short queuing and execution times enables ORION to model E2E latency, without the need to predict variable (and long), heavy-tailed queuing times that appear in other environments [210]–[212].

Mitigating Infrastructure-caused Delays. In serverless platforms, two types of stragglers can be observed: (1) Stragglers that experience longer execution times due to their input content (e.g., larger data portions or more complex inputs such as video frames with many objects). (2) Stragglers that appear due to infrastructure causes (e.g., network fluctuations). Bundling mitigates the first type of stragglers. The second type is well studied in the literature, and solutions such as *Speculative Execution* [184] work well in practice. Nevertheless, Bundling has a positive side effect of using fewer VMs/containers, reducing the likelihood of occurrence for infrastructure stragglers.

Supporting Dynamic DAGs. In a dynamic DAG, the execution flow is identified at runtime, say based on input data. Such DAGs appear in microservice-based applications [200], among others. ORION, as well as other provider-side tools, cannot have visibility into user data due to privacy concerns. Hence, ORION cannot support dynamic DAGs where the path is determined based on request content. **Future Work.** Our bundling approach increases VM size proportionally with the bundle size. For example, assuming a single function invocation use a VM of size VM_{single}, we bundle N invocations in a VM with a size

of $N \times VM_{single}$. There is, however, room to explore choosing other VM sizes beyond linear scaling. Furthermore, combining two or more in-series functions together to execute in a single VM can improve performance compared to invoking those function in separate VMs (e.g. due to avoiding remote storage communication). We plan to explore the performance benefits of these ideas.

6.10 Conclusion

We proposed Orion as a novel optimization technique for serverless DAGs. It presents four design innovations: a distribution and correlation-aware performance model for E2E latency, a resource optimization strategy, a design for bundling multiple invocations of a function within a stage to mitigate execution time skews, and a pre-warming strategy to mitigate cold starts. We evaluate Orion on AWS Lambda on three serverless applications with different DAG structures, skews in execution time, and communication patterns. We compare Orion to three competing approaches and show significant improvements in E2E latency, $ cost, or both. We highlight the following insights: (1) It is challenging to decide on the right resource configurations that accurately meet latency SLOs for serverless DAGs. (2) It is important to bundle parallel workers together to mitigate skew, yet it is challenging to pick the right bundle size that avoids resource contention. (3) We can leverage the DAG structure information along with latency CDF estimates to find efficient pre-warming delays that minimize E2E latency without degrading utilization.

7. WISEFUSE: WORKLOAD CHARACTERIZATION AND DAG TRANSFORMATION FOR SERVERLESS WORKFLOWS

7.1 Introduction

Serverless workflows are becoming increasingly popular for many applications as they are hosted on a scalable infrastructure with fine-grained resource provisioning and billing [114], [171], [179]. A serverless workflow contains two or more serverless functions orchestrated as a DAG (Directed Acyclic Graph). Commercial providers offer orchestration services to facilitate the design and execution of serverless DAGs (*e.g.* AWS Step Functions, Azure Durable Functions, and Google Cloud Workflows). The serverless platform, however, executes each function in the DAG in a separate VM without making use of the DAG structure and the data transfer among functions, which leads to significant increases in the end-to-end (E2E) latency and cost [118], [213], [214].

To understand the challenges of supporting serverless workflows, we study production workloads of serverless DAGs of *Azure Durable Functions* [173] over two weeks [1]. Our analysis shows that the vast majority of DAG executions are for recurring DAGs: the top 5% most frequent DAGs constitute 94.6% of all DAG invocations, with an invocation rate of at least 1.6K times per day.

With this high rate of invocations, the cloud provider can monitor the DAG execution parameters and quickly identify the bottlenecks to optimize the DAG execution. We identify two major performance bottlenecks: (1) *Communication latency between in-series functions*, which stems from the intermediate data that is typically passed through remote storage. (2) *Computation skew among in-parallel invocations of the same functions*, since each parallel invocation processes different content. Our analysis of the production DAGs shows that 46% of the DAGs have a high communication latency, and 48% of the DAGs have a high computation skew of 2× or more between parallel workers. Both factors are a direct result of the current state-of-practice where each function in the DAG runs in a separate VM.

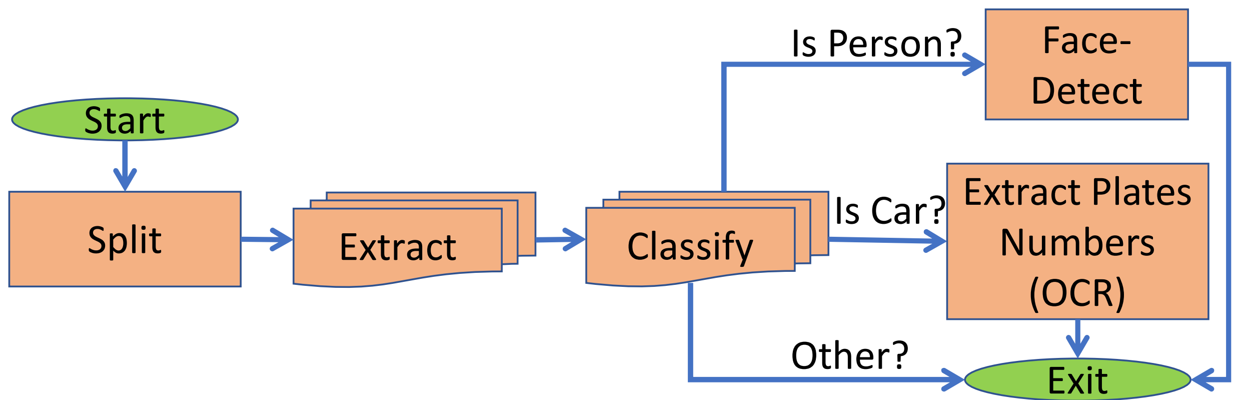
Key Ideas: We propose WiseFuse, which is an automated approach to generate an optimized execution plan for serverless DAGs. We show an example of WiseFuse's optimiza-

[1]↑We release a subset of the production traces and this is available from .

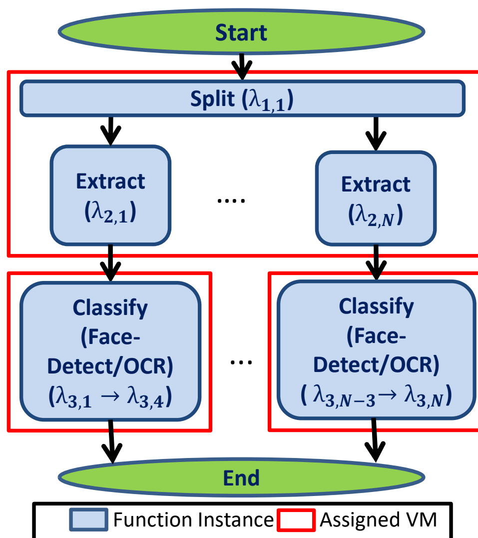
tions in Figure 7.2, and the solution overview is schematically shown in Figure 7.3. Users provide WiseFuse with a DAG definition that includes individual functions as nodes and their data dependencies as edges, which is typical in today's commercial offerings. We introduce three optimizations. (1) **Fusion**: Combining in-series functions together as a single execution unit. (2) **Bundling**: Executing parallel invocations of the same function together in the same VM. (3) **Resource Allocation**: Assigning the best size to each VM hosting a function or a group of the DAG functions. First, we show that Fusion allows for optimized communication between cascaded functions due to better data locality between sending and receiving functions. Second, we show that Bundling mitigates execution skew among the colocated function invocations and therefore reduces latency. Finally, assigning the best sizes for each VM hosting one or more DAG functions allows WiseFuse to reduce the E2E latency and $ cost.

Challenges: Determining a good execution plan for a serverless DAG poses three technical challenges. *First*, serverless functions experience a high runtime variability, even when executed as standalone functions [109], [111], [114], [118], [171], [214]. This variability in runtime can be either in communication time (*i.e.* while downloading/uploading data from remote storage), or in processing time. Hence, we need to model the communication time and processing time of individual functions as well as for the entire DAG as distributions, rather than as single-point estimates. *Second*, we also need to estimate the impact of Fusion or Bundling on the E2E latency distribution. The effect is non-monotonic, *e.g.* bundling of a certain number of functions decreases E2E latency but excessive bundling increases it, and the same argument applies to Fusion. *Third*, the search space of all possible execution plans is massive, due to the large number of possible DAG widths and VM size choices, making exhaustive grid search infeasible. Therefore, we need to develop an efficient algorithm to optimize the execution plan.

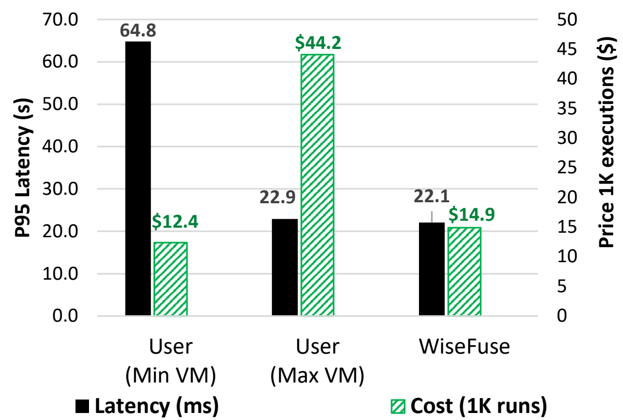
Our Solution: To overcome these challenges, we make three technical contributions in WiseFuse. *First*, we create a distribution and correlation-aware performance model to capture the variability in performance for each function. Our model breaks down the function's runtime into download, processing, and upload components, while taking into account the correlation between in-series and in-parallel workers. By profiling the latency distri-



(a) Video Analytics pipeline: User-defined DAG.



(a) WiseFuse's execution plan.



(b) WiseFuse's gains.

Figure 7.2. Example serverless DAG of Video Analytics application (7.1a) and the corresponding optimized execution plan generated by WiseFuse (7.2a). The gains in latency and cost over user-defined DAG, using either minimum or maximum VM size configurations (7.2b).

butions of the three components for each function, we can identify stages that experience high communication latency (*i.e.* high latencies in intermediate data download/upload) and hence can benefit from Fusion. We also identify stages with parallel invocations of the same function that experience execution skew (*i.e.* long tail operation), and hence can benefit from Bundling. *Second*, we search for a good bundle size, and the right VM size to assign to each bundle. Using the performance model, we also estimate the impact of joint Fusion and Bundling of functions on the E2E latency and cost. *Finally*, we use the above two contributions to develop a searching strategy that performs a joint optimization of the combination of Fusion and Bundling operations.

To get a sense of the impact of these three contributions combined, consider the DAG for a video analytics pipeline in Figure 7.1a and WiseFuse's execution plan in Figure 7.2a. Compared to the user-defined DAG (without performing any Fusion or Bundling action), WiseFuse achieves 64% lower latency compared to allocating the Min VM size for each function, and achieves 66% lower cost over allocating the Max VM size for each function.

Evaluation: We evaluate WiseFuse using three popular serverless applications with different DAG structures and show significant improvements in E2E latency and cost compared to four approaches from recent work. For example, for the video analytics application, WiseFuse achieves 62% lower latency than Photons [202], which colocates parallel invocations together in the same VM to maximize memory utilization. WiseFuse also achieves 39% and 47% lower latency than communication optimization frameworks, Faastlane [213] and [214].

Contributions: We make the following contributions in this paper:

(1) We characterize production FaaS workloads for serverless workflows (*i.e.* DAGs) of *Azure Durable Functions* over two weeks. From our analysis, we pinpoint two major performance bottlenecks: (a) high latency when transferring data between in-series functions. (b) Lack of resource sharing between in-parallel invocations of a function, leading to processing skew.

(2) We highlight two types of optimizations that can be performed by the cloud provider while executing a serverless DAG, *Fusion* and *Bundling*. We show that performing the two independently is counterproductive and we jointly determine a good combination of Fusion and Bundling actions.

(3) We implement a DAG-aware execution plan optimizer that meets a user-given latency objective while reducing cost. The execution plan specifies which functions should be fused or bundled, as well as the appropriate VM sizes.

The rest of the paper is organized as follows: Section 7.2 motivates the optimizations in WiseFuse using workload characterization of Azure Durable Functions. Section 7.3 gives an overview of our design, encompassing Fusion (grouping of in-series functions), Bundling (grouping of parallel invocations of a function), DAG transformation, usage model, and finding the best configuration. Section 7.4 gives the implementation details for WiseFuse. In section 7.5, we evaluate WiseFuse on AWS Lambda against competing approaches — an approach that does a limited form of Bundling (Photons), and approaches that perform a variant of Fusion (and Faastlane), and the user-defined DAG baseline configured either with the largest VMs to obtain low latency or with the smallest VMs for lower cost. Further, we use three applications for the evaluation — Video Analytics, Approximate SVD, and ML Pipeline, and use microbenchmarks to evaluate the individual contribution of Fusion and Bundling. We evaluate WiseFuse on another cloud provider using the Approx SVD application to see if our techniques generalize. We discuss related Work in Section 7.6, and discuss several aspects of WiseFuse (*e.g.* updating the performance model, scheduling, and security considerations) in Section 7.7. We present our conclusions in Section 7.8.

7.2 Workload Characterization

We analyze a subset of *Azure durable functions* production workloads over two weeks. Compared to a previous characterization of FaaS workloads [179], we focus here on characterizing serverless DAGs rather than single functions. Our analysis highlights important findings regarding serverless DAGs in the real world and motivates the optimizations of WiseFuse. We collected data on DAG executions for 14 days, between October 18 and October 31, 2021, from six data centers, three in the US, two in Europe, and one in Asia. We are releasing publicly an anonymized subset of this data to spur research in serverless DAGs [215].

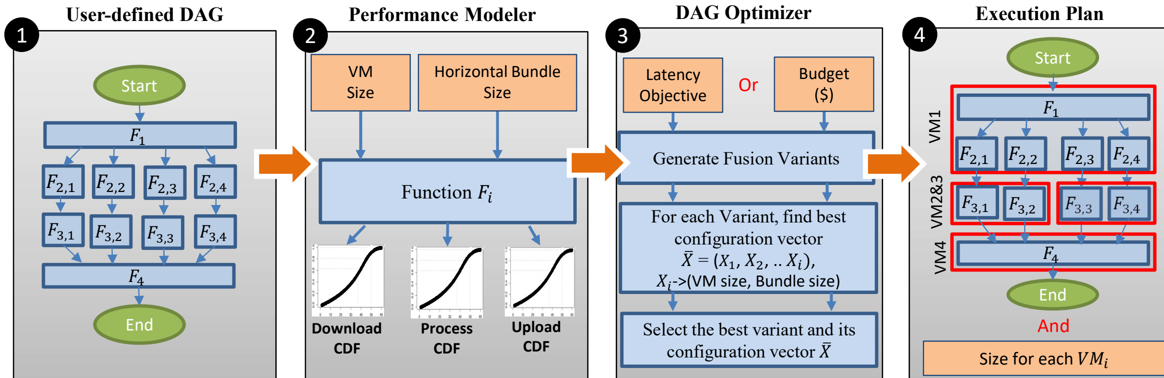


Figure 7.3. Overview of WiseFuse design. WiseFuse optimizes the execution plan for the user-defined DAG, which includes combinations of Fusion (*vertical coupling of in-series stages*), Bundling (*horizontal coupling of parallel workers in a stage*) actions, the size of each VM host a function or a group of DAG functions.

We give a general characterization of serverless DAGs in terms of invocation frequency and structure, then we provide specific characterizations to motivate the importance of Fusion and Bundling. Additionally, we use microbenchmarks to measure the data transfer latency for different data sizes at three major FaaS providers.

7.2.1 General DAG Characterization

Definitions A DAG is a sequence of stages S_1, S_2, ... S_d. The order of the stages in the DAG represent the order of execution (*i.e.* stage S_i executes before S_{i+1}). A stage S is a set of functions that execute in parallel F_1, F_2, ..., F_n, and $n = 1$ for a stage with a single function. All functions within a single stage execute the same code, but can have different inputs (*e.g.* scatter fanout stages) or same inputs but with different hyper-parameters (*e.g.* broadcast fanout stages).

Daily Invocations Figure 7.4 shows the daily DAG invocations collected data between October, 18–31, 2021. We notice that DAGs are consistently invoked at a high rate of 34M-55.8M invocations per day. By analyzing earlier data, we notice that the total number of DAG invocations per day has grown by 6× over the past 2.5 years, suggesting that this workload is growing rapidly.

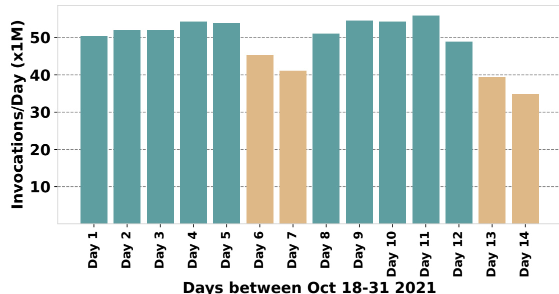


Figure 7.4. Daily DAG invocations over 2 weeks. Weekend days are in yellow.

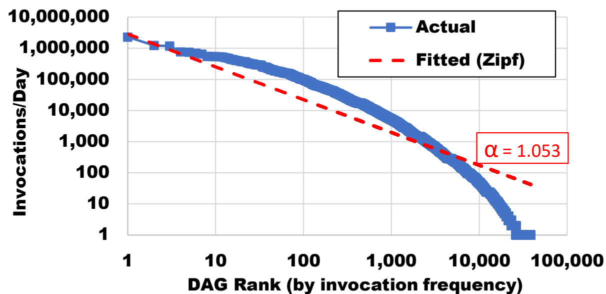


Figure 7.5. Daily invocations per DAG vs DAG rank (by frequency). Top 5% DAGs have 94.6% of invocations.

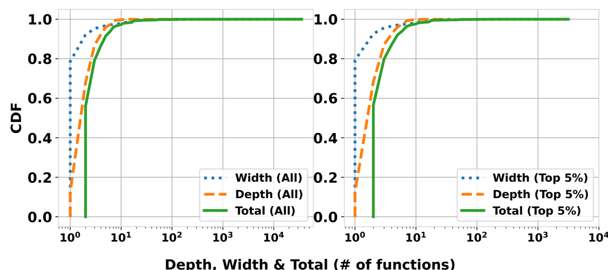


Figure 7.6. Distribution of DAG depth (*i.e.* stages), width (*i.e.* degree of parallelism) and total nodes (left for all DAGs, right for top 5%).

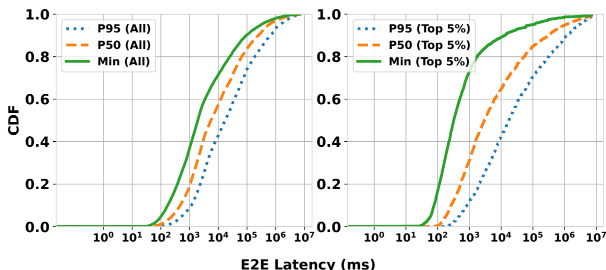


Figure 7.7. CDFs for P95, P50, min of execution time per DAG (left for all DAGs, right for top 5%). P50 of all DAGs is 5.6 sec, whereas it is 3.1 sec for the top 5%.

Invocations per DAG Now we study the frequency of invocations for each DAG. We collect the total invocations of each DAG across all 14 days and show the average daily invocations in Figure 7.5. We notice that the invocation frequency varies significantly, and the invocation frequency follows a Zipf distribution, which is well-known for modeling rank-frequency relations. For the fitted Zipf distribution, the goodness of fit Z-test achieves a p-value of 0.052 (*i.e.* above 95% confidence interval).

Notice that the majority of DAG invocations are for recurring DAGs: The top 5% most frequent DAGs constitute 94.6% of all DAG invocations, and their invocation rate is $\geq$ 1.6K/day. Therefore, improving the performance and cost of these DAGs yields large gains for both users and the cloud provider.

DAG Structure We now look at the structure of the DAGs. We collect the number of stages (*i.e.* depth) and fanout degree (*i.e.* width) in DAG executions. We use the maximum number of parallel functions across all fanout stages to determine the width of the DAG. Note that if the DAG is invoked K times, it contributes K data points for the width and depth distributions. Figure 7.6 shows the CDFs for the DAG widths, depth, and total number of functions. The median number of functions in the DAG is 3, whereas the median decreases to 2 for the top 5% most frequent DAGs. We also notice that 60% of all DAGs have a width $\geq$1, for which Bundling can potentially be applied. For the top 5% most frequent DAGs, the ratio of DAGs that can be bundled decreases to 44% and the remaining 56% DAGs are linear chains. DAG depth grows faster than the width till a crossover point ($\approx$80th percentile). At the tail, DAG depths are usually short and the max depth is 47, whereas DAG widths are longer and the max width is 10.9K. This is an important observation that we leverage in WiseFuse's design to reduce the search space, as described in Section 7.3.6.

E2E Latency for Serverless DAGs Here we show the distribution of the latency of all DAGs invocations as well as for invocations from the top 5% DAGs. We plot the CDFs for Min, P50, and P95 latency in Figure 7.7. We notice that the median execution time for all DAGs (*i.e.* median of medians) is 5.6 sec, whereas it decreases to 3.1 sec for the top 5% DAGs. We also notice that the variance in execution time between invocations of the same DAG is high (not captured in the figure). Specifically, the ratio between P95 and P50 is 3$\times$ on median, and the ratio between P50 and Min is 4.8$\times$ on median. Hence, it is essential to model the E2E latency of the DAG as a distribution to capture this variance. Additionally, we contrast the DAG invocation latency to the single function invocation latency from a prior study [179] that reported the median latency for serverless functions is 670 ms. Thus, serverless DAGs are longer running than individual functions, stressing the importance of optimizing DAG E2E latency and cost for users as well as for FaaS providers.

DAG Resource Consumption The cost of single DAG invocation is calculated as the product of consumed resources (memory is the observable parameter) by the execution time. To analyze the amount of resources consumed per DAG, we capture the resource consumption of its invocations, providing a distribution per DAG. We plot percentiles of that distribution for each DAG in Figure 7.8 in GB-sec units. For half of all DAGs, the median invocation

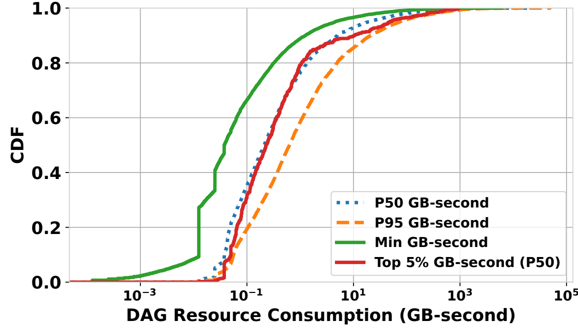


Figure 7.8. For 50% of DAGs, the median invocation consumes more than 210 MB-sec. For top 5% most frequent DAGs, this number is lower, at 230 MB-sec.

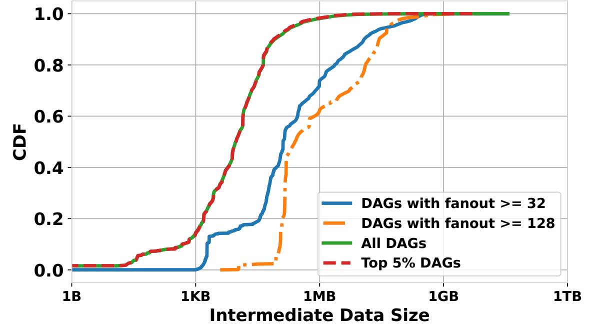


Figure 7.9. CDF of total intermediate data files passed across the entire DAG. Higher intermediate data sizes are observed with the top 5% most frequent DAGs.

consumes more than 210 MB-sec, and the median P95 is higher than 630 MB-sec. The max consumed memory per DAG (not observed in the figure) is 1 GB-sec on median. Moreover, for the top 5% most frequent DAGs, a lower median of 230 MB-sec is observed. To understand how resource consumption impacts the $ cost, we give an example for pricing in Azure Functions, AWS Lambda, and Google Cloud Platform (GCP). For a DAG that consumes 1 GB-sec per invocation, the cost of 1M invocations is $16 on Azure Durable Functions, $16.6 on AWS Lambda, and $16.5 on GCP. This motivates the need for allocating the right resources for functions in the DAG to reduce its latency *and* cost.

7.2.2 Data Transfer Between Stages

Volume of Intermediate Data Here we analyze the total volume of intermediate data being transferred between all stages in a DAG. We show the distribution of intermediate data sizes in Figure 7.9. We notice that 85% of the transferred intermediate data are of sizes $\geq$ 1KB and the median is 8KB. When we focus on DAGs with parallel stages, we notice that the size of intermediate data increases with higher fanout degrees. For example, the median data size for DAGs with stages of 32 workers or more is 710KB, which is 88× the median intermediate data size for all DAGs. Finally, the size of intermediate data for the top 5% DAGs is not significantly different from that for all DAGs, with a median of 8.4KB.

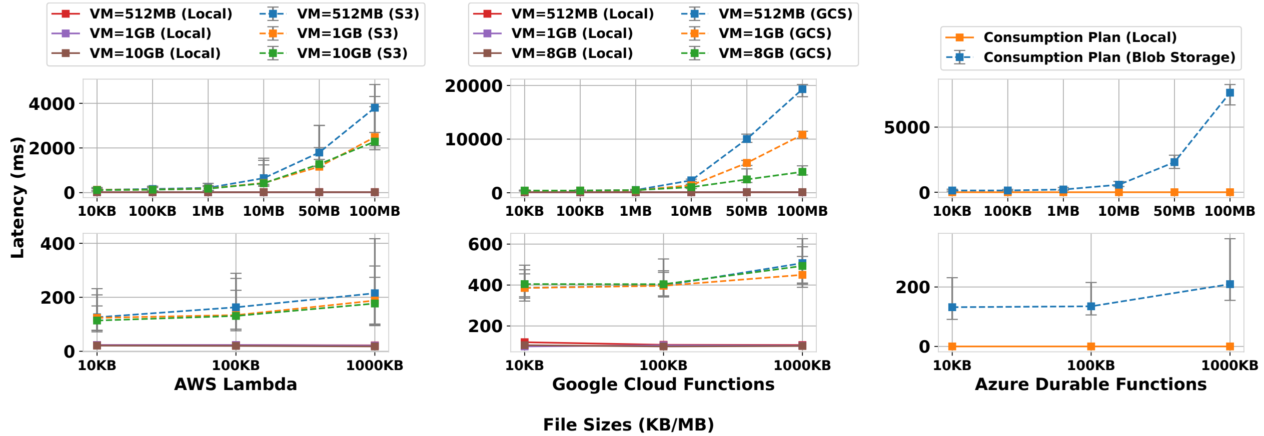


Figure 7.10. Data exchange latency on AWS Lambda, Google, and Azure comparing remote storage to local communication. Markers represent the median, and error bars represent Min to P95 latency range.

Therefore, Fusion is essential to reduce latency and cost for all DAGs in general, and it is more beneficial for wider DAGs in particular.

Latency for Transferring Intermediate Data We run a benchmark on AWS Lambda, GCP, and Azure Durable Functions to characterize the latency for exchanging data between serverless functions. We use a simple linear chain of a single sender and a single receiver function. The sender generates a random array of bytes of a particular size and uploads it as a file to remote storage (S3 for Lambda, Google Storage for GCP, and Blob Storage for Azure), then the receiver downloads the file from remote storage. We measure the E2E latency for the chain and subtract the generation and reconstruction times, hence the remaining time becomes the upload/download times. Figure 7.10 shows the median, Min, and P95 latency when exchanging different data sizes through remote storage. We vary the VM sizes for both sender and receiver functions between 512 MB and 10 GB (Lambda's Max) or 8 GB (GCP's Max), whereas we use the consumption plan for Azure which assigns resources automatically to the sender and receiver functions.

We make the following observations: (1) For AWS Lambda, increasing the VM size from 512 MB to 1 GB yields a significant reduction in data passing latency for file sizes > 1 MB. However, increasing the VM size further (> 1 GB) does not reduce the communication latency indicating that the network bandwidth saturates at this point. On the other hand, for

GCP, increasing the VM size increases the network bandwidth but sub-linearly (16× increase in VM size causes a 4× speedup in data transfer). Hence Fusion becomes essential to reduce data exchange latency and cost in both platforms. (3) Data exchange latency has a high variability, with a ratio of 8× between the P95 and the median, and a ratio of 3.4× between the median and the minimum latency for AWS Lambda. For GCP and Azure, a lower variability is observed (ratio between P95 and median is between 18% and 80%), but with a lower network bandwidth compared to AWS Lambda. Hence, it is essential to represent the upload and download times as a distribution to capture that variability. (4) Using Fusion, the data exchange latency is reduced significantly for all file sizes and for all platforms (as observed by comparing *local* to *S3, GCS, or Blob Storage* variants.). Moreover, gains are higher for bigger file sizes making Fusion more beneficial for data intensive applications.

7.2.3 Skew among Parallel Workers

We analyze the degree of runtime skew observed among all parallel stages in the DAG. For each stage, we calculate the degree of skew as the ratio between the runtime of the *slowest* worker to that of the *fastest* worker. If the DAG has multiple parallel stages, we take the maximum skew across all parallel stages. Recall that the E2E latency of the stage (and the entire DAG) is dominated by the slowest worker in the stage. We show the distribution of DAG skew in Figure 7.11. We notice that the median skew is 1.9×. When we focus on DAGs containing parallel stages with higher fanouts, the median skew rises to 15× and 132× for DAGs containing stages with 32 and 128 parallel workers, respectively. For the top 5% DAGs, the median is close to 1 (no skew) but 32% of the top 5% DAGs have a skew greater than 2×. Thus, skew among parallel workers is significant and that motivates the need for efficient skew mitigation optimizations for serverless DAGs, which we achieve by Bundling.

7.2.4 Impact of DAG Skew and Intermediate Data Size on Latency

Finally, we analyze the impact of skew and intermediate data size on the distribution of DAG E2E latency. We split all DAGs into two groups with respect to their average skew (skew < 100 and skew ≥ 100). Similarly, we split all DAGs into two groups with respect to

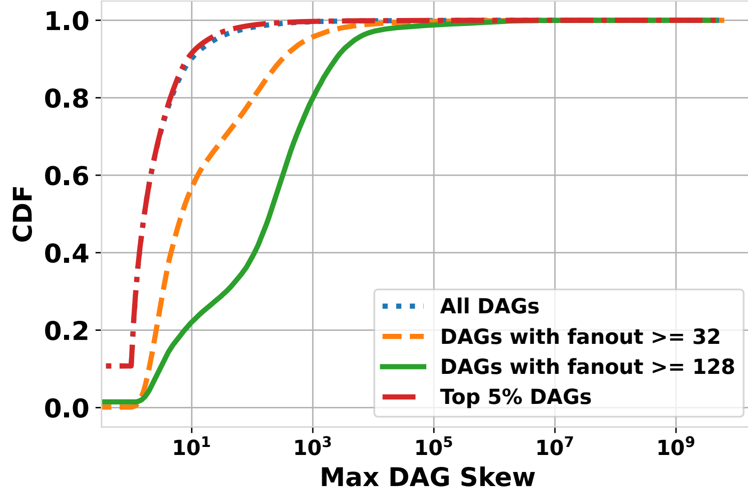


Figure 7.11. CDF of DAG max skew for all DAGs, top 5%, for DAGs with fanout ≥ 32 & ≥ 128.

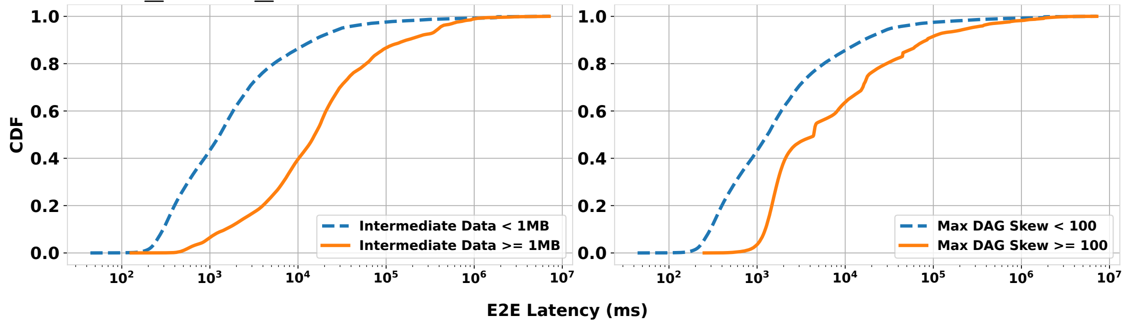


Figure 7.12. CDFs of DAG latency per max skew (right) and intermediate data size (left). Longer E2E latencies are observed for DAGs with higher max skew and intermediate data size.

their average intermediate data size (size < 1MB, and size $\geq$ 1MB). We show the E2E latency distributions for the four groups in Figure 7.12. We notice that both skew and intermediate data size have a significant impact on the DAG E2E latency. Specifically, DAGs with skew $\geq$ 100 have 17× higher median latency than DAGs with skew < 100. Similarly, DAGs with intermediate data size $\geq$ 1MB have 9.5× higher median latency than DAGs with size < 1MB. This motivates our focus on these two factors to optimize through Bundling (reducing execution skew) and Fusion (reducing intermediate data passing latency).

7.3 Design

First, we give an overview of WiseFuse's design and its components. Second, we give the details for building a performance model for each function in the DAG, which we use to estimate the impact of Fusion or Bundling on the E2E latency and cost. Finally, we show how WiseFuse explores the vast search space of actions for Fusion, Bundling, and VM size allocation to find the best execution plan.

7.3.1 Overview

Fusion: Fusion combines two consecutive stages to reduce the data exchange latency between them. For example, performing Fusion of stages S_{i} and $S_{\mathrm{i+1}}$ combines *each* sending function in S_{i} with its receiving functions in $S_{\mathrm{i+1}}$. Accordingly, Fusion needs to take into account the communication primitive between the two stages. For example, if the communication primitive between stages S_{i} and $S_{\mathrm{i+1}}$ is one-to-one (*i.e.* each function in S_{i} is sending data to only one function in $S_{\mathrm{i+1}}$), then the Fusion is performed in one-to-one manner and the resulting stage will have the same degree of parallelism (DOP) as both stages. However, if the communication primitive is one-to-many, many-to-one, or many-to-many, Fusion will generate a single stage with a DOP of $Min(DOP(S_{\mathrm{i}}), DOP(S_{\mathrm{i+1}}))$. Finally, if the communication primitive is all-to-all (*i.e.* shuffling), Fusion combines all functions in stage S_{i} with all functions in stage $S_{\mathrm{i+1}}$, producing a stage with a single function.

Bundling: Bundling is the operation of colocating two parallel workers together to run on the same VM, hence enabling resource sharing between them (Figure 7.13). Higher bundle sizes are achieved by recursively applying the same operation, leading to bundle sizes that are powers of 2. Bundling reduces execution time skew by allowing stragglers to access more resources (*e.g.* CPU capacity and/or Memory) once other colocated workers finish execution. For Bundling to be useful, the function has to be scalable, i.e., given more resources it should be sped up (till a point). Notice that Bundling does *not* require knowledge/prediction of which invocation is the straggler. However, for Bundling to be efficient, stragglers need to be spread out over different bundles as much as possible, which WiseFuse achieves through shuffling (Figure 7.14).

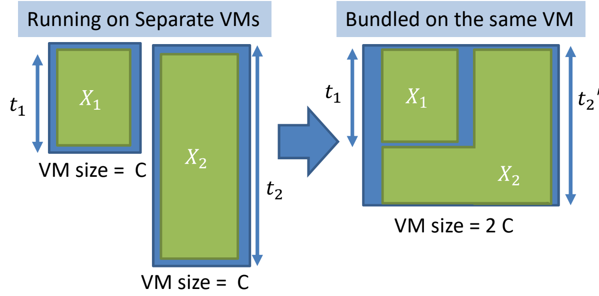


Figure 7.13. Without Bundling, latency is dominated by the straggler (X_2) and is equal to t_2. With Bundling, X_2 gets more resources after X_1 executes, decreasing the stage's latency to t_2'.

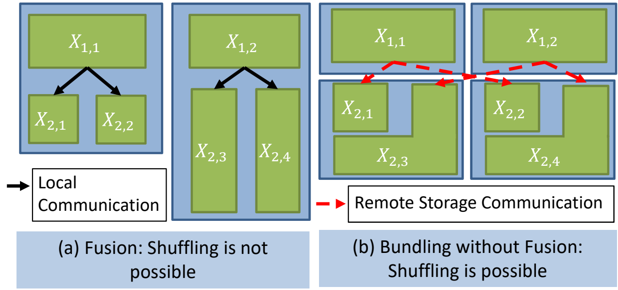


Figure 7.14. Coupling between Fusion and Bundling. Fusion reduces data exchange latency but prevents shuffling, which is essential to break the locality of stragglers and reduce the E2E latency.

Bundling also reduces data exchange latency in two cases. First, for broadcast stages, our bundling technique leverages the fact that all workers within a single bundle can read from the same local copy of the input data, and hence downloads the data once per bundle. Second, in case of a skew in the intermediate data sizes between the two parallel workers, Bundling allows for resource sharing so that the straggler gets a higher network bandwidth, speeding up its upload and download operations.

DAG Transformation: WiseFuse takes as input a user-defined DAG, with each function denoted as a separate node. By applying a sequence of Fusion and Bundling operations, the original DAG is *transformed* to a new DAG with fewer number of nodes. WiseFuse's target is to identify the best combination of Fusion and Bundling operations that transforms the DAG into the best performing DAG in terms of E2E latency and cost. The transformations do not need any additional effort or code changes from the user. Figure 7.2 shows an example of a user-defined serverless DAG (7.1a), WiseFuse's transformed DAG (7.2a), and the gains after the transformation (7.2b).

7.3.2 Workflow and Usage Model

We show the main design components of WiseFuse in Figure 7.3. First, similar to current commercial FaaS platforms, the user provides WiseFuse with the DAG definition, which includes the executable package for each function and the data dependencies between the functions in the DAG.

Second, we execute WiseFuse's `performance modeler`. We represent each function as a composition of three steps: download (input data), process, and upload (output data). We then perform per-function profiling and latency modeling to capture the variability in each of these three steps. This is essential for us to estimate the gains of fusing in-series functions together in a single VM, or Bundling in-parallel functions together in the same VM. For example, functions that have a high latency in the download or upload steps can benefit more from Fusion due to removing the data exchange latency between in-series functions. On the other hand, functions that experience latency skew in the process step can benefit more from Bundling due to resource sharing.

The `modeler` also estimates the degree of correlation between latencies of either in-parallel or in-series functions, which is essential to correctly estimate the impact of performing Fusion or Bundling. The output of the `modeler` is a performance model for each stage in the DAG that can estimate the latency distribution for each of the three steps (download, process, and upload) given a candidate VM size and bundle size.

Third, the `DAG optimizer` uses the generated performance models and explores the vast search space of Fusion, Bundling, and VM size allocations for the entire DAG. Next, it proposes the best set of transformations that leads to a new DAG which is given to the user for approval before deployment.

Usage model: In the typical usage model, the cloud provider deploys WiseFuse and applies it to serverless DAGs submitted by the clients. The cloud provider does the profiling runs needed for the `performance modeler`, without charging the client. In its profiling runs, the vendor cannot use content characteristics to enhance its model, due to data privacy limitations. It can however use metadata like the size of the input. Alternately, WiseFuse can be deployed by users as a user-side optimization tool. In this case, the users can provide

hints of content to WiseFuse such as the input category (*e.g.* for Video-Analytics application, input category can be: Sports, News, etc.). WiseFuse can use these content-aware hints to build a separate performance model for each input category and further improve the estimation accuracy of the model. Further, the user can apply WiseFuse at her end to do the DAG transformation, using Fusion and Bundling, on her original application DAG. She then submits the transformed DAG to the serverless platform of the cloud provider.

7.3.3 Per-function Performance Model

The first step is to build a performance model that maps the amount of resources for a given function to the expected latency distribution. Notice that modeling the latency distribution is important for both latency-sensitive and cost-sensitive applications. This is because of the pay-as-you-go cost model of serverless platforms, which bills the user proportionally to the product of the allocated resources and the function's runtime. We create this latency distribution separately for the three phases of a function's execution — data download, execution latency for the function itself, and data upload.

Profiling: To start off, WiseFuse profiles the latency distributions for 4 VM sizes, including Min and Max VM sizes. *Min implies the lowest size at which the function will execute and Max implies the largest VM size supported by the FaaS platform.* The value of the intermediate VM sizes vary for different FaaS platforms. For AWS Lambda, there are fine-grained VM sizes supported — 128 MB to 10 GB in step sizes of 1 MB. We do not wish to profile for all possible VM sizes and we leverage discontinuities that exist in the vendor offerings. Hence, we pick 1,024 MB as one intermediate profiling point as that is the size at which AWS saturates the network bandwidth (*i.e.* increasing memory beyond it does not provide any more network bandwidth as shown in Figure 7.10). We pick 1,792 MB as another intermediate profiling point as a VM with one full core gets assigned at that point. For GCP, there are only 7 VM sizes to pick from and here we can afford to profile for all sizes.

Interpolation: This initial profiling divides the configuration space into multiple regions. For example, for AWS Lambda, this divides the VM size space into 3 regions: Min-1024,

1024-1792, and 1792-Max. Afterward, WiseFuse performs percentile-wise linear interpolation to infer the CDF for the intermediate memory settings. For example, the P50 latency for 6 GB is estimated as the average between the P50 latency of 1.8 GB and of 10.2 GB. This generates a *prior* distribution for these intermediate memory settings. To verify the prediction accuracy in a region, WiseFuse collects a few test points using the midpoint memory setting in that region to measure its actual CDF (*i.e.* the *posterior* distribution) and compares it with the *prior* distribution. If the error between the *prior* and *posterior* CDFs is high (*i.e.* $\geq 15\%$) in any region, WiseFuse collects more data for the midpoint in that region and adds it to its profiled points, splitting that region further into two smaller regions. This process is repeated till saturation in accuracy is reached for all regions. In practice, we find that dividing the space into 5 regions is sufficient to achieve an error $\leq$ 15% for all latency percentiles.

7.3.4 Estimating the Impact of Fusion

Here we show how WiseFuse estimates the impact of fusing two in-series functions together. We represent the latency of a function as a composition of three components: (1) Download-duration (*Down*): the time to read the input file(s) from remote storage. (2) Process-duration (*Proc*): the computation time. (3) Upload-duration (*Up*): the time to upload the output file(s) to remote storage. We model the function's latency PDF as a *convolution* between the three components:

$$P(latency = t) = P(Down = k, Proc = m, Up = t - k - m) \tag{7.1}$$

Notice that we still represent each component as the distribution of a random variable, rather than a constant, to capture the latency variability. By factorizing the latency into these three components, we can easily estimate the impact of fusing two functions on their combined latency and cost. For example, if we perform Fusion between `Extract` and `Classify` stages in Figure 7.1a, we remove the upload step from `Extract` and the download step from `Classify`. Thus, we estimate the combined latency as a convolution between *Down*(Extract), *Proc*(Extract), *Proc*(Classify), and *Up*(Classify) CDFs. Impor-

Algorithm 1 Get Bundled-Pair Latency

Input: NumIterations=N, PerfModel: Model, WorkerSize: C
Output: Latency CDF for bundled pair: CDF_{bundle}

1: ## Use performance model to get the CDF for a single worker in the bundle with VM of size C
2: CDF_{Single} = Model(C)
3: ## Use performance model to get speedup distribution for a single worker when executed on a VM of size 2C
4: CDF_{Double} = Model(2C)
5: ## divide the percentiles of the CDF with double sized VM over the CDF with single sized VM
6: $CDF_{Speedup} = CDF_{Double}/CDF_{single}$
7: **for** i = $0->N$ **do**
8: ## Sample 2 latency points t1, t2 that where observed in the same run (to sustain the correlation)
9: Sort the points t1<t2
10: Estimate skew = t2-t1
11: Set reduced-skew = $CDF_{Speedup}$(skew) * skew
12: Add (t1 + reduced-skew) to CDF_{bundle}
13: **end for**
14: return CDF_{bundle}

Figure 7.15. Pseudo code for calculating the latency CDF for any sized bundle of functions in one stage

tantly, WiseFuse takes into consideration the correlation between the different components to efficiently estimate the convolution between them. For example, we estimate the Pearson correlation coefficient between all four components and find a strong correlation of 0.73 between *Proc*(Classify) and *Up*(Classify) CDFs. This is because it takes longer to perform object detection for frames with many objects; it also takes longer to crop and upload the detected objects. Hence, in WiseFuse, we use the marginal distributions for the statistically independent components, while we use the joint distribution for highly correlated components (*Proc*(Classify) and *Up*(Classify) in this case). We show the accuracy of WiseFuse's estimates for the impact of Fusion in Section 7.5.4.

7.3.5 Estimating the Impact of Bundling

Here we show how our model estimates the impact of bundling parallel invocations together on their combined latency distribution. By bundling parallel invocations, stragglers

can leverage additional resources released by the fast executing workers (Figure 7.13). This allows for local resource sharing between the bundled workers and decreases their combined latency. However, Bundling can cause contention if the number of bundled invocations is high compared to the VM's size. Accordingly, selecting the best bundle size is not trivial and it has to be carefully selected. Therefore, Bundling is beneficial and used when: (1) The DAG has a fanout stage since linear-chains do not benefit from Bundling. (2) Functions are scalable: The function can leverage additional resources when made available. (3) Input content skew: Stragglers experience longer execution times due to their input content, not variability due to infrastructure (*e.g.* poor network bandwidth). (4) Stragglers can be spread out over different bundles, which we achieve through shuffling.

For simplicity of design, we consider that workers can be bundled in powers of two only. Further, all bundle sizes within one stage are equal. We give the steps for estimating the latency distribution of a bundle of workers in Algorithm 1. First, we use the performance model to estimate the latency distribution for a single function when assigned additional resources. For example, if the function is originally allocated a VM size (C) and then is bundled with another function in a VM with double the size ($2C$). The ratio between $\frac{CDF(2C)}{CDF(C)}$ shows the speedup due to additional resources, *i.e.* the function's scalability. We then draw pairs of latency points from our profiles to capture the natural skew observed between the two workers. Then, we estimate the reduction of that skew (due to Bundling) using the speedup CDF. In summary, the algorithm considers two important factors: (1) The speedup distribution due to additional resources (2) The natural degree of skew observed between the two workers.

Notice that we represent the speedup as a distribution rather than a scalar value, which is essential to capture the reduction in latency for each latency percentile. For example, Figure 7.16 shows the two CDFs for the `Classify` function in our Video Analytics application, using 1 core VM vs 6 cores VM. We notice the speedup varies for different percentiles. For example, the speedup on the median is only 11%, whereas it becomes 47% for the P95. This is because invocations at higher percentiles experience higher skew and therefore reap greater benefit from additional resources.

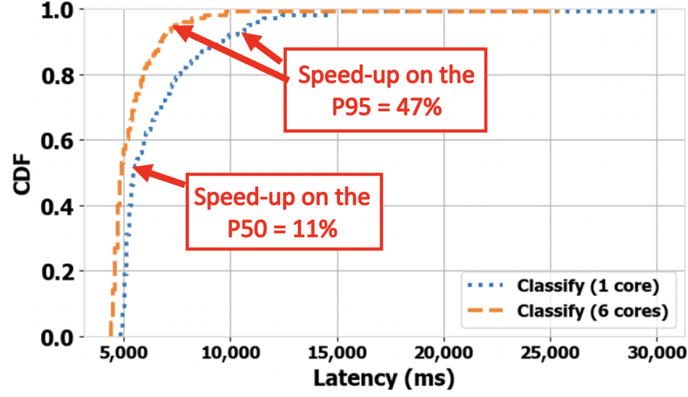


Figure 7.16. Speed-up in `Classify` when executed on a 6-core VM vs a 1-core VM.

A stage's latency is estimated as the "Max" latency among all bundles (stage i with N bundles):

$$P(X_{\text{i}} \leq z) = P(X_{\text{i},1} \leq z, X_{\text{i},2} \leq z ... X_{\text{i},N} \leq z) \tag{7.2}$$

Similar to Eq 7.1, identifying the correlation between the parallel workers is essential to accurately estimate the stage's latency distribution. Specifically, in case of a low correlation between parallel workers, we can simplify Eq 7.2 as the multiplication of the marginal distributions, $P(X_{\text{i}} \leq z) = P(X_{\text{i},1} \leq z)^N$. However, in case of a high correlation, this simplification can cause overestimation of the combined latency. In our applications, we observe a high enough correlation coefficient (0.4-0.6) between parallel workers, hence we use the joint distributions when estimating Eq 7.2.

7.3.6 Execution Plan Optimization

Here we show the steps for finding an optimized execution plan for a user-given latency objective.

Search Space: We calculate the size of DAG transformations, including Fusion, Bundling, and VM size selection. Let D be a DAG of N stages, denoted as $S_1, S_2, ..., S_N$. Since we can perform Fusion of any consecutive pair of stages S_{i} and $S_{\text{i}+1}$, we have $N-1$ consecutive pairs, and for each pair, we have a binary decision (to fuse or not to fuse). By performing this decision recursively, we actually explore *all* possible Fusion configurations between

consecutive stages (not just pairwise Fusion). Hence, we have a total of $2^{(N-1)}$ possible combinations of Fusion actions. Since in practice N is not large (P50 depth is 2 and P99 is 12, Figure 7.6), this exhaustive search of Fusion space is feasible. Now, let the maximum degree of parallelism in any stage be 1K (recall that the actual observed P99 width is 84). Since we consider bundle sizes to be powers of 2 only, we have a maximum of $log_2(1000) \approx 10$ possible bundle sizes for each stage. Notice that after we perform Fusion for any pair of stages, they are represented as a single stage in the new DAG. Therefore, the total number of Fusion and Bundling actions is given by $\sum_{\mathrm{i}=1}^{N} \binom{N-1}{\mathrm{i}-1} \times 10^{\mathrm{i}}$, where i represent number of stages after performing any combination of Fusion actions, and it varies between 1 (all stages are fused into one stage), and N (no Fusion). For a DAG of 8 stages and max degree of parallelism of 1K, we have 215.6M possible Fusion and Bundling actions that we can select from. Notice that this space does not include the possible actions of selecting the VM sizes, which can be as few as 7 different sizes for Google Cloud [216], or as many as any memory size (in steps of 1 MB) between 128 MB and 10.24 GB for AWS Lambda [181]. Thus, an exhaustive search to find the best plan is expensive and impractical.

Importance of E2E Optimization: Here we show the importance of selecting Fusion, Bundling, and VM allocation options for all stages in the DAG *jointly*, rather than optimizing each stage separately. Consider a DAG of 2 stages such that the first stage has only one function denoted as X_1. The second stage has k parallel invocations of a function, denoted as ($X_{2,1}$, $X_{2,2}$, ... $X_{2,k}$) and they receive their input files from X_1. Now, in order to achieve data locality between the two stages and minimize X_1's latency, we perform Fusion. This forces us to execute the sending function and all receiving functions in the same VM. Thus, we force Bundling with a bundle size of k over the second stage. This can be harmful for large values of k where Bundling can cause contention in the VM and increase the E2E latency and cost.

Another cause of local optimization shortcoming is that Fusion removes the possibility to "shuffle" stragglers across bundles, which is needed in many cases to break the locality of stragglers. For example, consider the example shown in Figure 7.14a for a stage of two workers $X_{1,1}$ and $X_{1,2}$ and each of the them triggers two workers in the second stage. In the second stage, workers $X_{2,3}$ and $X_{2,4}$ are stragglers. If we perform Fusion between the two

stages (which is the best local decision for workers $X_{1,1}$ and $X_{1,2}$), we will have to execute the two stragglers together in the same bundle and hence the locality of stragglers will remain. In contrast, if we do not perform Fusion between the two stages, we can spread the two stragglers over two separate bundles, which leads to a lower E2E latency for the bundles (Figure 7.14b). Accordingly, selecting the best Fusion and Bundling actions for each stage separately (*i.e.* local optimization) is sub-optimal. WiseFuse avoids this pitfall by selecting the execution plan (which includes Fusion and Bundling options for all stages) that optimizes the E2E latency (or cost) for the entire DAG.

Search Strategy: As shown in Figure 7.3, the `DAG optimizer` takes as input either a latency objective (for latency-sensitive applications) or a budget (for cost-sensitive ones). The first step is to generate different Fusion variants from the user-defined DAG, and by default, we explore all Fusion variants. However, if exploring all Fusion variants is infeasible in a specific situation, we can select a subset that fuses stages with high data exchange (upload/download) volume. Afterward, a searching algorithm is executed for each variant *in parallel* to identify its best configuration vector $\bar{\mathbf{X}}$, which denotes the best VM sizes and bundle sizes for each stage in that variant. After the configuration vectors are generated for all Fusion variants, we select the vector that meets the latency objective with the lowest cost. Equivalently, for the budget objective, we select the vector that meets the budget objective and provides the lowest latency among the options.

Handling Different Input Sizes: The same serverless DAG can be executed with different input sizes, which can impact the execution time, memory footprint, or the fanout degree of its functions. Accordingly, different input sizes can be optimally executed with different execution plans. WiseFuse leverages polynomial regression to estimate the upload, process, and download CDFs for new (unseen) input sizes. The order of the polynomial is different for different applications and stages. For example, upload and download CDFs of `PCA` stage in our ML pipeline application have a linear relation while its process CDF has a quadratic relation with input size (PCA has quadratic compute complexity [147]). We evaluate WiseFuse's ability to handle different input sizes in Section 7.5.4.

Finding the best configuration vector: We show the pseudocode for WiseFuse's optimizer in Algorithm 2. First, we estimate the latency distribution for the fused stages as

Algorithm 2 Pseudocode for WiseFuse's optimizer

Input: DAG = D with N stages, PerfModel = Model, Latency Target = T
Output: Execution Plan = ExecPlan

1: ## From DAG D, get all $m = 2^{(N-1)}$ Fusion variants DV_1, ... DV_m
2: ## Initialize $Set_{\bar{X}}$ as empty set
3: **for** i = 0 to m (in-parallel) **do**
4:     Set Vector $\bar{X}$ = DP(DV_{i}, Latency Target T)
5:     Add $\bar{X}$ to $Set_{\bar{X}}$[i]
6:     Set Cost[i] = GetCost(DV_{i}, $\bar{X}$, Model)
7:     Set Latency[i] = GetLatency(DV_{i}, $\bar{X}$, Model)
8: **end for**
9: ## Across all variants, find best variant index i_{best}
    such that Latency[i_{best}] $\leq$ T and Cost[i_{best}] is minimum
10: return ExecPlan = ($DV_{\text{i}_{best}}$, $Set_{\bar{X}}$[i_{best}])

shown in Section 7.3.4. Afterward, we apply a Dynamic Programming (DP) search algorithm to select the best VM size and bundle size for each stage. We observe that our optimization problem can be reduced to the well known **Knapsack problem**. We set the knapsack's capacity to be our latency objective for the latency-sensitive application (equivalently, the capacity would be our $ cost objective for a cost-sensitive application). The items in the knapsack represent configurations of each stage (one configuration/stage). The weight of each item is the latency of that configuration and the cost of each item is the inverse of the $ cost of that configuration (since our target is to minimize the cost subject to meeting the latency objective). The algorithm proceeds as follows: For each stage in the DAG, we estimate the latency and cost for each feasible action, which includes all VM sizes (in steps of 128 MB) and all bundle sizes (in powers of 2). We divide the latency objective into equal-sized windows (10 ms) and actions that lead to the same latency window are considered equivalent. Hence only the action that has the least cost is saved in the DP table, while others are pruned. By doing so, we keep the number of solutions that gets transferred from one stage to the next bounded (at most A×T), leading to a much faster searching time. Specifically, the time complexity is O($S \times A \times T$), where S is the number of stages, A is the number of possible actions per stage, and T is the number of latency buckets (Latency Target/10 ms). For a DAG of 8 stages, 46 VM sizes, and 10 bundle sizes for each stage, we have 3.68K actions to explore using DP, compared to 215.6M for the exhaustive case.

7.3.7 Further Design Considerations

Interaction with Cold Starts: WiseFuse's performance modeler profiles the latency distribution for each function in the DAG using different inputs. During profiling, DAG invocations are performed serially and in quick succession to leverage warm VMs and to eliminate cold starts. Accordingly, the latency distribution of our model captures the variability that is due to input content only, not cold starts. One positive side effect of performing Fusion or Bundling is that they cause the DAG execution to use fewer VMs. Hence, they can reduce the incidence of cold starts. For example, assuming a uniform cold-start probability of 1%, the user-defined DAG for our Video Analytics application executes with 65 VMs (1 VM per function). Hence the probability of *any* function in the DAG hitting a cold-start = 1-$(P_{warm})^N$ = 1-$(0.99)^{65}$ = 48%. On the other hand, WiseFuse's execution plan (Figure 7.2a) uses 9 VM's only hence brings this probability down to 1-$(0.99)^9$ = 8.6% only.

Sharing Functions between Multiple DAGs: WiseFuse does not limit sharing of the same function between multiple DAGs. This is because both Fusion or Bundling are implemented during the deployment phase, whereas the functions' user-defined packages and source codes are still separately shared and edited by users to preserve modularity of the functions.

Handling Deeper DAGs: The optimizer shown in Algorithm 2 explores *all* Fusion variants for the input DAG, which can be a large pace to explore in case the DAG has many stages. Recall that for a DAG of N stages, we have $m = 2^{(N-1)}$ Fusion variants. Although the Fusion variants are explored in parallel, the computation cost for the optimizer can become problematic if the number of stages grow. To reduce the size of this search space, we can construct an ordered list of pairs of stages combined with the median data communication latency between them: $[(S_{\text{i}}, S_{\text{i+1}}, \mu_{comm})]$. Afterwards, we can limit the optimizer to explore Fusing the subset of pairs of stages that have the highest data communication latency between them, and for which Fusion will be most efficient.

Mitigating Infrastructure Skew WiseFuse 's Bundling objective is to mitigate input-based (demand-side) execution straggler. However, Bundling has a positive side effect of

reducing the overall number of VMs/containers used to execute a stage and therefore can indirectly minimize the infrastructure-based (supply-side) variability as well. For example, with Bundling of broadcast fanout stages, each bundle downloads the input file once and hence the file is made available for all invocations at the same time. This removes any possible variability in download times (e.g. due to network fluctuations) between the invocations in case they were executed on separate VMs.

Using Homogeneous Bundle Sizes: Our solution considers homogeneous bundle sizes only. It might seem more beneficial to bundle parallel workers into heterogeneous bundle sizes according to their execution times. For example, a better solution can be executing straggler workers as standalone functions with high resources, while bundling short running workers together with low resources. Unfortunately, this approach requires identifying based on content, which workers are likely to become stragglers, something that providers are unable to do due to privacy regulations.

7.4 Implementation

We implement WiseFuse in C# with 2.7K LOC. The runtime of our search heuristic is 6 ms for Approx SVD, 535 ms for ML pipeline, and 779 ms for Video analytics. WiseFuse collects execution times for N invocations per function to build its performance model, and the value of N is user-defined. We find empirically that N=300 is sufficient to model P95 latency with error $\leq 15\%$ for our three evaluation applications. For higher percentiles (*e.g.* P98), we would need to profile more points. In general, more points need to be profiled for accurate predictions for higher percentile latencies (e.g. we need at least 1000 points to profile the P999 latency). The profiling cost with N=300 on AWS Lambda is $3.8 for Approx SVD, $0.9 for ML pipeline, and $1.7 for Video analytics, whereas it is $3.4 for Approx SVD on Google. In our evaluation on AWS Lambda, we use StepFunction [188] for function orchestration. However, because of data transfer quota limits in Google Cloud Functions (details in Section 7.5.5), we use remote storage triggers to orchestrate the functions in the DAG.

Fusion: We follow the same programming paradigm as used in current FaaS platforms, where developers identify a `handler` for each function, which serves as the function's entry point. When two functions are to be fused together, their execution packages and dependencies are combined and a single wrapper is added to simply call both function handlers in order. Then, we intercept function calls to remote storage API to redirect the communication to the next fused function through local memory. In case WiseFuse decides *not* to fuse the two stages, the upload and download commands are forwarded to the remote storage API and their latency is recorded. Thus our implementation abstracts away from the user whether Fusion is used, i.e., whether data is passed through remote storage or local memory.

Bundling: To perform Bundling, we rely on parallelism APIs supported by the runtime (*e.g.* Python3's multiprocessing API) to invoke multiple parallel instances of the function's handler. Note that we only bundle invocations of the same function and hence the bundled invocations share the same execution packages and dependencies. Moreover, we bundle invocations that belong to the same DAG execution (and hence belong to the same user) for data privacy considerations. In case the bundled functions are data dependent on the same input file (*i.e.* Broadcast fanout), we download their input file once per bundle. We expect the VM's OS to dynamically assign more resources (*e.g.* more CPU capacity) to stragglers after shorter running workers complete their execution.

7.5 Evaluation

7.5.1 Baselines and Competing Approaches

We evaluate the latency and $ cost for each application using WiseFuse's execution plan versus the following baselines and competing approaches.

1. **User-defined DAG**: This is the state-of-practice in which each function in the DAG runs in a separate VM and VM sizes are either right-sized to the functions' memory footprints (User-Min) or set to max VM size (User-Max).

2. **Faastlane** [213]: Here the entire DAG is executed within a single VM. Faastlane has a fallback strategy of using remote storage when a single VM is not large enough to execute

all the functions within one stage. In this case, the functions are divided into bundles but the bundle size is fixed to # CPU cores of the VM. The VM size is set to fit the most resource-demanding stage.

3. **Sonic** [214]: Reduces communication latency by selecting between three data passing methods: remote storage, direct passing, and local VM-storage passing. However, direct passing cannot be supported in AWS Lambda or GCP and we implement 's remote storage and local VM-storage options only. performs no Bundling or any other skew mitigation technique. Further, it assumes function execution times are deterministic and hence does not use function execution time distributions.

4. **Photons** [202]: Colocates parallel invocations together to improve the memory utilization, not to meet latency or cost objectives. However, this colocation mitigates execution skew to some extent. Moreover, Photons colocates as many parallel invocations as possible as per the function's memory footprint. This can cause excessive bundling and can increase the E2E latency.

5. **Theoretical lowest latency**: This provides the theoretical (oracle), lowest latency for any serverless DAG. It runs all the functions in the serverless application within one VM, which can be arbitrarily large to accommodate the largest number of parallel workers in any stage.

7.5.2 Applications

The three applications[2] that we use in our evaluation are adopted from prior works and they show a diversity in structure (*i.e.* depth and width), intermediate data volumes, and execution time skews.

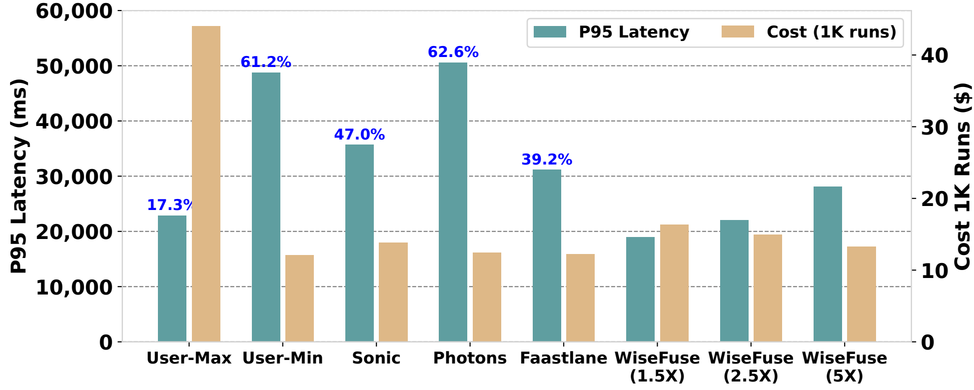
(1) **Video Analytics**: Adopted from Pocket [114] and [214]. This application performs object detection and classification for frames in a video (Figure 7.1a). The first stage in the DAG, called `Split`, downloads the input video from remote storage (S3) and splits it into equal chunks of 2 seconds. Each chunk is then sent to `Extract` stage to extract a representative frame from the chunk. Each frame is then sent to `Classify` stage to detect

[2]↑The code of the three applications is available from

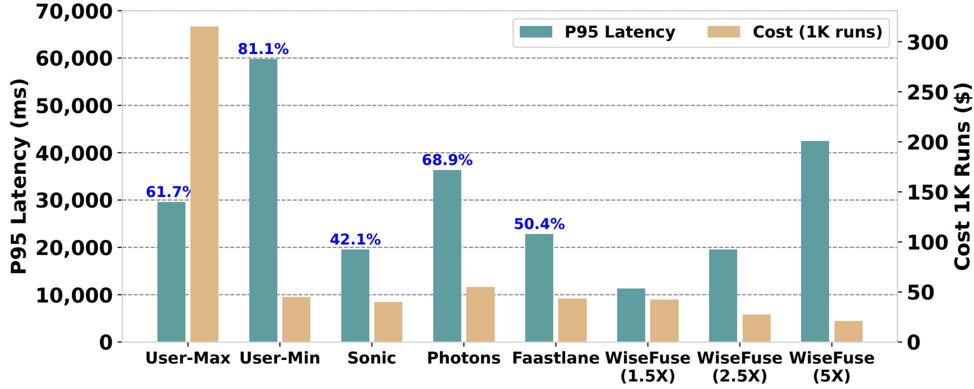
and classify all objects in the frame. Finally, the objects are either sent to `Face-Detect` if the frame contains a person, or sent to `OCR` if the frame contains a car. We use 600 YouTube videos of lengths 1 minute, 5 minutes and 10 minutes. Of them, 50% are used for profiling, and the other 50%, for evaluation.

(2) **Approximate Singular Value Decomposition (SVD)**: This application estimates an approximate Singular Value Decomposition (SVD) for a large-scale input matrix, which is widely used in recommender systems [217], [218]. SVD has a time complexity of $O(n^3)$ and a space complexity of $O(n^2)$. Accordingly, several approximation techniques have been proposed to parallelize SVD computations and significantly reduce the runtime and memory footprints [219]–[221]. We perform SVD for the Netflix prize dataset [222] and use the Split-Merge approach [223]. The dataset has movie ratings for 17,770 movies, submitted by 480,189 users, and each rating is on a 5-star integer scale, from 1 to 5. The application has two stages. The first stage performs a matrix split operation to a user-defined number of sub-matrices. Each sub-matrix is then sent to an instance of RunSVD function to estimate its decompositions. Finally, all matrix decompositions are saved to remote storage. First, as a preprocessing step, we save the Netflix data in 128 equal blocks, divided by rows. This is the minimum degree of parallelism that we can run at without hitting the maximum memory limit in AWS Lambda (10 GB). The 128 blocks are then saved to remote storage. We set the number of split functions in the first stage to 128 (one invocation per block) and each invocation splits the block further into k sub-matrices ($k = 2$ in our case). Thus, we run 256 instances of the RunSVD function, in parallel, on the sub-matrices. With this degree of decomposition, we calculate the reconstruction error and it is small ($< 0.5\%$), which is considered acceptable for most SVD deployments.

(3) **ML Pipeline**: Adopted from Cirrus [115], this application trains a random forest (RF) prediction model for the MNIST handwritten characters [144], containing 810K images. The first function downloads the dataset from remote storage and performs Principal Component Analysis (PCA). In the second function, a hyper-parameter tuning stage is invoked, which includes 64 parallel lambdas, each executes with a unique set of hyper-parameters (# features, # trees, and tree max depth). In the final function, the accuracy of each model is



(a) Video Analytics



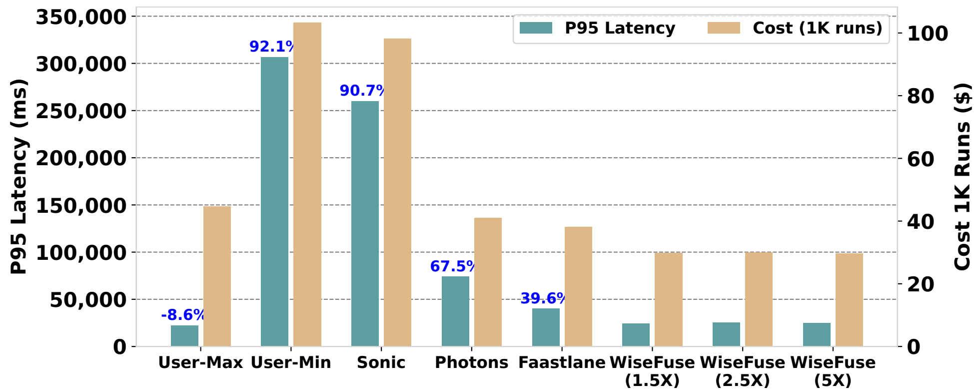
(b) Approximate SVD

Figure 7.17. WiseFuse's comparison in latency and $ cost. % over the bars show WiseFuse's gains in E2E latency (1.5× configuration) over this particular scheme. We set the latency target to (1.5×, 2.5×, and 5×) of the best theoretical latency (total processing time using an arbitrarily large VM and zero communication latency).

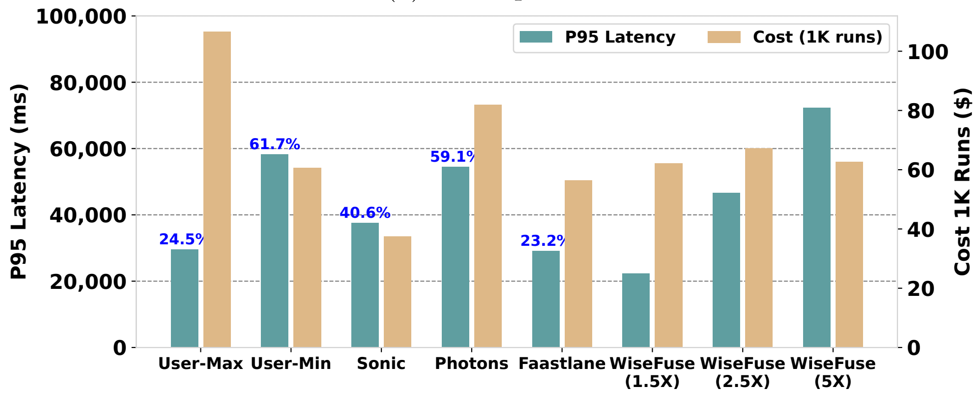
estimated on a held-out dataset and the top 10 models are uploaded to remote storage. The LightGBM boosting framework was used for training the RF [191].

7.5.3 E2E Evaluation

Video Analytics: We compare the latency and cost of WiseFuse to the baselines using 300 test videos, which were not used in the profiling. We show the P95 latency and cost for 1K executions for each baseline in Figure 7.17a. WiseFuse and all baselines are executed in AWS Lambda, using Step Functions for DAG orchestration and Amazon S3 as the remote storage. We report the cumulative cost of 1K executions. We show the performance of three



(a) ML Pipeline



(b) Approximate SVD on Google

Figure 7.18. WISEFUSE's comparison in latency and \$ cost. % over the bars show WISEFUSE's gains in E2E latency (1.5× configuration) over this particular scheme. We set the latency target to (1.5×, 2.5×, and 5×) of the best theoretical latency (total processing time using an arbitrarily large VM and zero communication latency).

different settings of WiseFuse, corresponding to different latency objectives relative to the Theoretical Lowest Latency (1.5×, 2.5×, and 5×). This evaluates the ability of WiseFuse to be configured to meet different price-latency points. For the rest of this E2E evaluation, we refer to the performance of WiseFuse 1.5×.

Compared to the user-defined DAG baseline, WiseFuse achieves either 63% lower cost than User-Max or 61% lower P95 latency than User-Min (although it increases the cost somewhat (11%) over User-Min). This is because of WiseFuse's execution plan, which fuses the `Split` stage with the `Extract` stage to reduce the communication latency, also bundles parallel invocations of `Classify` together to mitigate computation skew. Moreover, by assigning the right resources to each set of bundled or fused functions, WiseFuse meets the latency objective with significantly lower cost.

Compared to , WiseFuse achieves 47% lower P95 latency. Recall that only considers fusing in-series functions to leverage data locality, and hence fuses the `Split` with the `Extract` stage together. However, it neither performs any Bundling nor considers the latency distribution and hence assigns resources that minimize the *average* latency, in contrast to WiseFuse that assigns the right resources to meet the required latency percentile objective.

Considering Photons, we notice its latency is as high as User-Min and WiseFuse achieves a lower latency by 62%. Recall that Photons performs Bundling mainly to improve memory utilization. Therefore, it bundles as many parallel invocations as possible based on the functions' memory footprint. This causes Photons to bundle 16 parallel invocations of the `Classify` stage together in one VM. Although this maximizes the memory utilization, it assigns less than optimal resources to each worker and causes a significant increase in latency. WiseFuse uses its performance model to identify the best bundle size, which is 4 for this application.

We notice Faastlane is the closest baseline to WiseFuse, yet WiseFuse achieves 39% lower P95 latency. Faastlane (similar to WiseFuse) fuses the `Split` and the `Extract` stages together, but falls back to remote storage (S3) when executing the `Classify` stage. Moreover, it uses a fixed bundle size of 6 workers to match the 6 vCPUs that are provided by AWS Lambda's Max VM size. Additionally, Faastlane cannot mitigate the significant execution skew that exists among the parallel workers.

Approximate SVD: We show the latency and cost of WiseFuse versus baselines in Figure 7.17b. Compared to user-defined DAG, WiseFuse achieves lower latency than User-Min and User-Max by 61% and 81% respectively. Although User-Max assigns the maximum resources to each worker, it still suffers from the increased communication latency between the split and SVD stages. WiseFuse performs Fusion of these stages and reduces the communication latency significantly. WiseFuse achieves 50% lower latency than Faastlane, 68% lower latency than Photons, and 42% lower latency than . This again shows the importance of *jointly* performing both types of optimizations while assigning the cost-optimized VM sizes. In terms of cost, only (marginally) reduces the cost over WiseFuse (7%), while it (significantly) increases the latency (42%).

ML Pipeline: We show the evaluation results in Figure 7.18a. We notice two main differences in this application. First, using Min VM sizes (as done by User-Min) does not provide the lowest cost. In fact, it increases the cost by almost 2× compared to using the Max VM sizes. The reason is that the `ParamTune` function shows a superlinear improvement in the runtime when allocated additional resources. For example, the function's runtime reaches 306 sec when assigned a VM of size 1,240 MB (VM min size) and it becomes only 22.2 sec when the VM size is increased to 10,240 MB—an 8× increase in the allocated resources leads to a 14× decrease in the latency, and therefore WiseFuse provides lower cost than User-Min. Second, WiseFuse produces similar execution plans for latency targets of 1.5×, 2.5×, and 5× the best theoretical latency. This is again because exploring lower VM sizes or bundle-sizes for this application does not reduce the cost compared to the 1.5× case, and hence the optimizer converges to this particular plan as it meets all latency targets with reduced cost. Finally, WiseFuse meets the latency targets of 1.5×, 2.5×, and 5× the best theoretical latency for all three applications with a small error in the range of [0.3%,12%].

Comparison to exhaustive search: As a proxy for exhaustive search, we compare WiseFuse with grid search where we set grid search to start from the best Fusion strategy given by WiseFuse and then explore all bundle sizes between 1 and 16 (increments of 1) and all VM sizes between 1 GB and 10.24 GB (AWS Lambda's Max) (increments of 500 MB). This results in a total of 32K profiling runs for grid search (recollect we need at least 100 samples for each data point for creating a distribution) compared to 300 points for

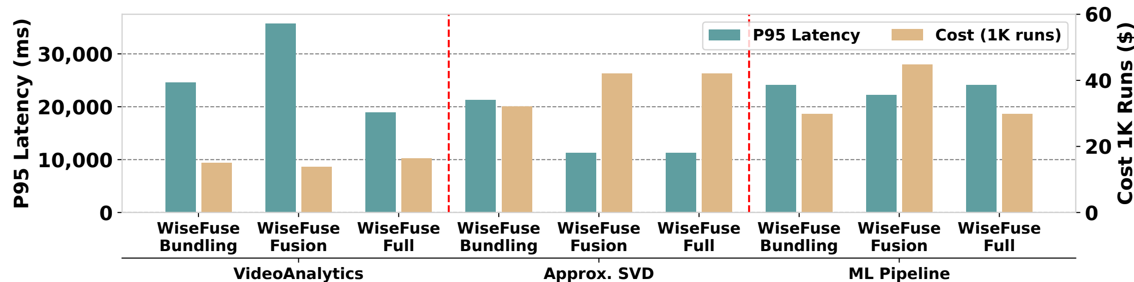


Figure 7.19. Comparison between WiseFuse (-Full) and its two variants that do Bundling only, or Fusion only.

WiseFuse (which simply uses its performance model trained with these 300 points). We show the result in Figure 7.20. The execution plan found by grid search gives 1.1% lower latency for Video Analytics, 15% for Approx SVD, and 2.7% for ML Pipeline. However, grid search's plans use a smaller bundle sizes compared to WiseFuse and hence increase the cost by 28% for video analytics and 5% for Approx. SVD. Recall that the cost shown is only the cost of executing the DAG, and does not include the profiling cost, which as discussed above is much higher for grid search. Considering the lower profiling cost, and correspondingly lower incurred cost, of WiseFuse, this latency performance would be considered acceptable in many situations.

Compute and Communication Latency Scaling in GCP We look at the change in both compute and communication latency with varying memory sizes on Google Cloud Functions for the Approx SVD application in Figure 7.24. The general serverless notion of scaling all resources with respect to the memory allocated is generally true, except when going from 4GB to 8GB. Both these memory sizes have the same compute power (2.4 GHz) and thus the average compute latency doesn't change between these two. As for communication latency, we see a consistent decrease as the memory size increases. This points to the network bandwidth continuing to scale unlike in AWS Lambda where it saturates at 1792MB (1 core).

Ablation Study: To analyze the gains of Fusion and Bundling separately, we limit WiseFuse to perform Bundling only (called WiseFuse-Bundling) or Fusion only (called WiseFuse-Fusion). Both variants still select the best VM sizes for nodes in the DAG after transformation. We compare the performance of these two variants to our full solution (WiseFuse-Full)

in Figure 7.19 for our three applications. For Video Analytics, we notice that WiseFuse-Full achieves lower latency than both variants. We also notice that WiseFuse-Bundling achieves lower latency than WiseFuse-Fusion, indicating that execution skew contributes more latency than data exchange for this application. For Approx SVD, we notice that WiseFuse-Full achieves a similar performance to WiseFuse-Fusion, whereas WiseFuse-Bundling has a higher latency but lower cost. The reason is that in this DAG, Fusion forces bundling of size 2, which WiseFuse-Full also found to be the best transformation. Finally, for ML Pipeline, WiseFuse-Full achieves similar performance to WiseFuse-Bundling as WiseFuse-Full found that all Fusion decisions were harmful to performance and hence performs Bundling only for this DAG.

In summary, WiseFuse outperforms all baselines and competing approaches and is able to provide an execution plan that reduces E2E latency and cost. Further, both Fusion and Bundling are required for achieving the gains, albeit these two contribute to different extents for different applications.

7.5.4 Microbenchmarks

We run microbenchmarks to evaluate the accuracy of WiseFuse's performance model, impact of Fusion, impact of Bundling, and impact of varying the input size on the E2E latency and cost.

Per-function Modeling Accuracy

Here we show accuracy of our per-function performance model, described in Section 7.3.3. First, we profile each function in the DAG using the initial VM size set: {Min, 1 GB, 1.8 GB, and 10.24 GB(Max)}. The Min VM size is different for each function and it is selected for each function based on its memory footprint. For each VM size, we collect 300 latency points for each function. We also collect the download and upload, while we estimate the processing time by subtracting the download and upload latencies from the function's overall latency. We then use these latencies to construct the CDF for each of the three operations (Download, Upload, and Process). For each VM size $\in$ {Min, 1 GB, 1.8

Table 7.1. RMSE in latency estimates for each function in our evaluation applications.

Application	Function	P95 RMSE (Actual vs Estimated)			
		Download	*Process*	*Upload*	*Overall*
Video Analytics	Split	2.44%	2.89%	7.6%	3.61%
	Extract	0.44%	1.34%	1.8%	0.25%
	Classify	5.2%	1.6%	11.8%	2.3%
3*ML Pipeline	PCA	0.8%	1.7%	12%	4.4%
	ParamTune	16%	3%	18%	13%
	Combine	0.24%	0.33%	0.9%	0.4%
2*Approx SVD	SplitRows	3.8%	0.1%	15%	12%
	RunSVD	2.5%	7.7%	6%	2.7%
Average		**3.9%**	**2.3%**	**9.1%**	**4.8%**

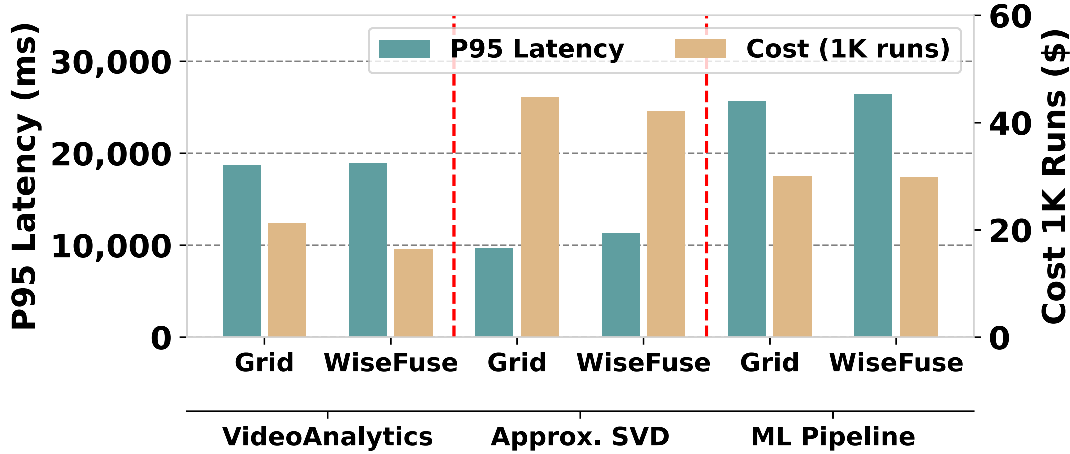


Figure 7.20. Comparison between WiseFuse and Grid Search.

GB, and 10.24 GB(Max)}, we collect profiling information for each function. For evaluation, we use 300 points for 3 test VM sizes. We use our performance model to estimate the CDFs for each function in our evaluation applications and compare the estimated P95 latency to the actual P95 latency. We show the results in Table 7.1. We notice that our performance model has very low error, in the range [0.25%, 13%] across all functions and all applications. Moreover, the upload and download operations have higher errors on average than the process operation. This is because of the high variability in data exchange latency that we empirically observe in production environments due to network bandwidth fluctuations. This stresses the benefit of our Fusion mechanism to reduce communication latency.

Estimating Latency After Fusion and Bundling

In this section, we evaluate the accuracy of our performance model in estimating the latency after we perform either Fusion or Bundling to our video analytics application. For Fusion, we use a single linear chain of ”Split→Extract→Classify” and we want to estimate the chain's E2E CDF when all three functions are fused together, thus evaluating our algorithm described in Section 7.3.3. As mentioned in Section 7.3.4, considering the correlation between the operations is essential to accurately estimate their combined CDF. Here we notice that ”Classify-Process” and ”Classify-Upload” have a high correlation of 0.7. This high correlation is because frames with a large number of objects take longer for identification, cropping, and uploading. We compare in Figure 7.21 the case where we neglect this correlation with the case where we consider it (as in WiseFuse). As in the figure, ignoring this correlation causes underestimation of the combined latency and increases the error to a maximum of 12.5%. However, WiseFuse, using the joint distributions to take this correlation into consideration, reduces the error to $\leq$ 3.4%. This shows the importance of our correlation-aware performance model to accurately estimate the impact of fusing functions together.

To evaluate the benefit of Bundling and our ability to predict the latency CDF, we consider the Bundling strategy WiseFuse chooses for the `Classify` stage. The `Classify` stage is a ripe target as it experiences the highest computation skew in the DAG. WiseFuse selects a Bundle size of 4. We compare WiseFuse's estimated CDF using the approach in Section 7.3.5 to the actual CDF of the Bundle in Figure 7.22. We also show the standalone (*i.e.* no bundling) CDF before Bundling using a VM size of 1,792 MB, which comes with a full vCPU core. We notice that with Bundling, the combined CDF improves the tail latency significantly compared to standalone CDF as P95 latency decreases by 27%. This highlights the gain due to skew mitigation through Bundling. Moreover, we notice that our estimated CDF for the bundle is very close to the actual, with errors in the range of ± 7% across all percentiles. In summary, WiseFuse's performance model shows high accuracy in estimating the impact of Fusion and Bundling on the DAG's latency, which is essential in selecting the best execution plan for the DAG.

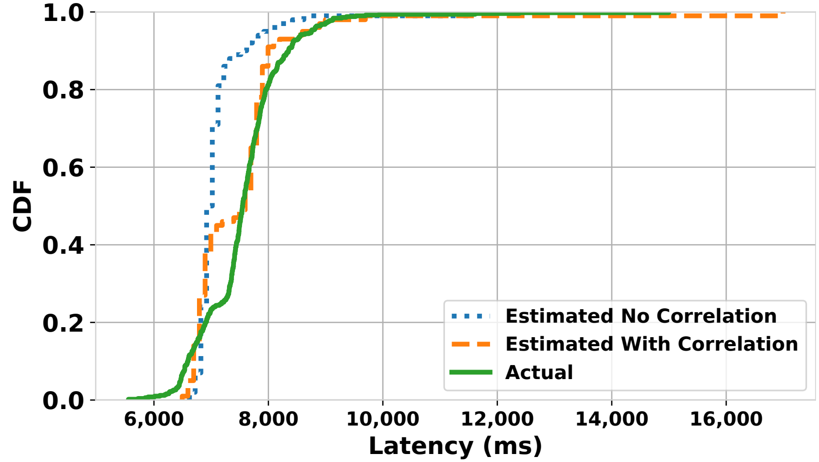


Figure 7.21. Accuracy of WiseFuse's CDF estimation for a fused chain for our Video Analytics application. Without considering correlation, errors can be up to 12.5%. With correlation, errors come down to $\leq$ 3.4%.

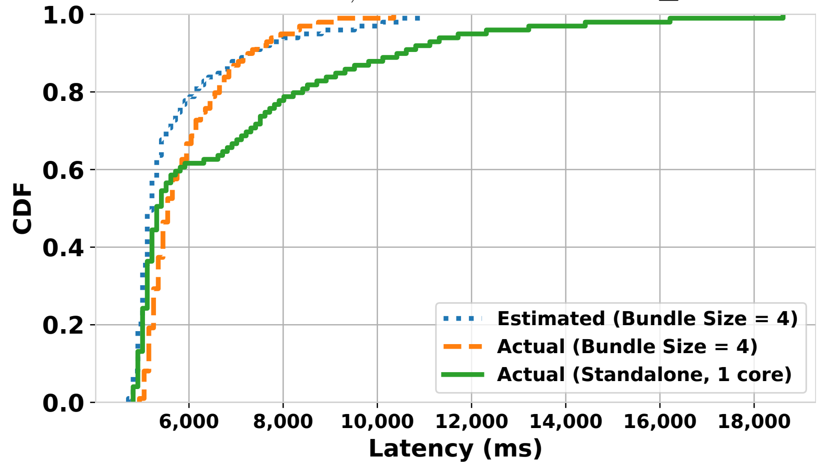


Figure 7.22. Accuracy of WiseFuse's CDF estimation for a bundle of 4 `Classify` workers. WiseFuse achieves errors in the range of [-6.4%,7%] compared to the actual CDF.

Varying Input Data Size

Here we evaluate WiseFuse's performance and cost with different input sizes. We execute the Video Analytics application with varying input video lengths of 1, 5, and 10 minutes, using 300 videos for each length. We notice that changing the input size impacts the following parameters: (1) Upload, process, and download CDFs for `Split` stage. (2) Fanout degree for `Extract` and `Classify` stages. To evaluate the accuracy of our polynomial regression, we use two sizes as inputs and estimate for the third size. Our estimated CDFs

show an error of $\leq 9.4\%$ across all percentiles. Afterward, we compare WiseFuse's execution plan to the user-defined DAG with both min- and max-VM sizes in Figure 7.23. We notice the following: first, with 1-min video inputs, WiseFuse and User-Max achieve similar latency but with 68% lower cost for WiseFuse. The reason is that WiseFuse mitigates execution skew through Bundling. However, user-Max, by allocating the max VM size to all workers, also mitigates execution skew, albeit, with higher cost. In contrast, with 5-min and 10-min video clips, WiseFuse achieves lower latency than user-Max by 61% and 69% respectively as WiseFuse performs Fusion between `Split` and `Extract`. Hence, WiseFuse reduces the data exchange time, which increases as we increase the input video length. In summary, by estimating the impact of varying the input size on the upload, process, and download CDFs, WiseFuse finds optimized execution plans for different input sizes.

7.5.5 WiseFuse on Google Cloud

Here we show the results of WiseFuse and baselines on Google Cloud using the Approx SVD application (Figure 7.18b). This is to verify if the benefits of WiseFuse are tied to any esoteric FaaS platform features (AWS Lambda) or if they generalize. We use the same implementation of WiseFuse as on AWS Lambda, with S3 being substituted by Google Cloud Storage. Significantly, the GCP equivalent for AWS Step Functions, Google Workflows, has very limited support for invoking serverless functions directly and this feature is still in beta [224]. The recommended approach is to invoke functions using HTTP calls but this is infeasible for applications with large fanouts (*e.g.* Approx SVD) since we hit the memory limit of 64KB [225] easily when counting the sizes of all the HTTP headers for all responses. Thus, as a workaround, we use Google Cloud Pub/Sub [226] topics where one stage publishes to a given topic upon completion, and the subsequent stage, is triggered by an event from Pub/Sub. We use the Approx SVD application in preference to the other two since this has a much higher fanout and thus stresses our workaround for GCP Workflows.

Our benchmarks show that in Google Cloud Functions, network bandwidth continues to scale with VM sizes (unlike in AWS Lambda) as shown by the decrease in communication latency with memory sizes (Figure 7.10). However, we still see a high data communication

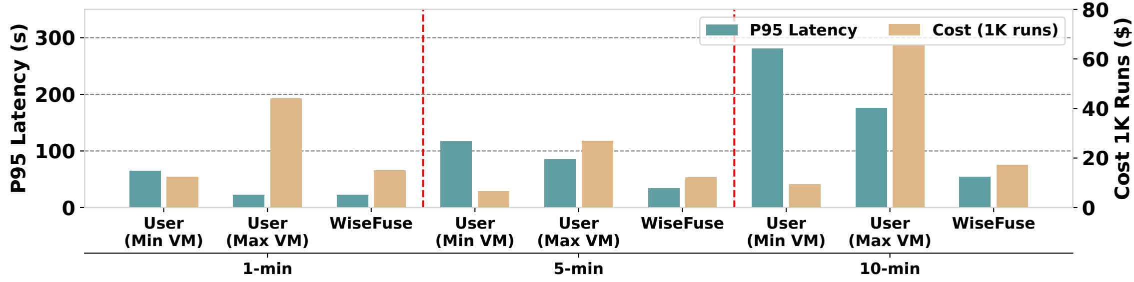


Figure 7.23. Video Analytics: Comparison between WISEFUSE's optimized plan and user-defined DAG using min and max VM sizes. We show how the latency and cost are impacted by varying input sizes.

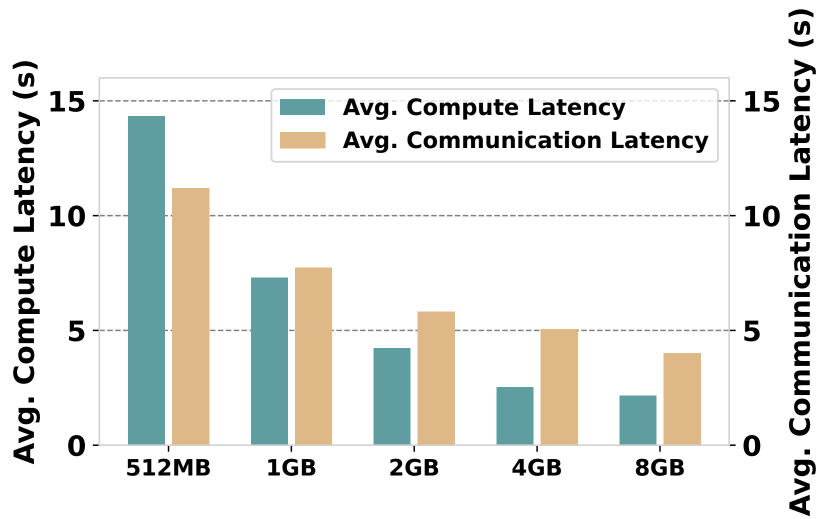


Figure 7.24. Average latency for compute and communication on Google Cloud Functions with varying memory size. The network bandwidth continues to scale unlike in AWS Lambda.

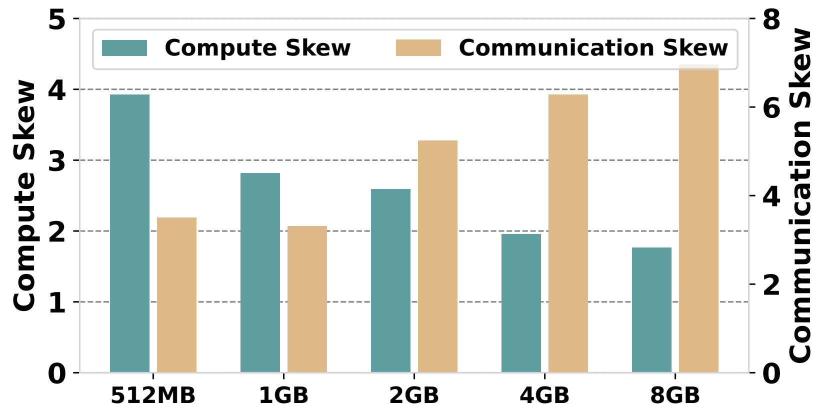


Figure 7.25. Skew for compute and communication on Google Cloud Functions. While network bandwidth scales with memory size, the communication skew grows.

skew — up to 6.8× between parallel workers as seen in Figure 7.25 — which causes significant increase in the E2E latency and cost. This skew is mitigated by WiseFuse's Fusion since that eliminates data communication between stages. Figure 7.18b shows that WiseFuse achieves lower latency than all other baselines similar to the evaluation on AWS Lambda. The main difference between the two platforms is the amount of compute power available. In GCP, the maximum memory size of 8 GB still only provides the same compute power (2 cores) as a 4 GB VM [227]. In contrast, in AWS Lambda, we get up to 6 cores with the 10.24 GB VM. This means that Bundling is not as effective on Google Cloud. Consequently, baselines that perform only Fusion (and Faastlane) are much closer in performance to WiseFuse.

7.6 Related Work

Characterization and modeling of serverless workloads. A related study [179] characterizes FaaS workload of Azure Functions, focusing on individual functions rather than on DAGs. We draw a number of comparisons between our work and this study, and highlight their implications on performance and cost (Section 7.2). Atoll [203] also characterizes popular apps from AWS Serverless Application Repository (SAR) but does not shed light on the DAG structures. Microservice architecture is a more general computing model than serverless computing. A microservice is similar to a serverless function, but being stateful, can be long running. The workload characterization of microservices at Alibaba [228] shows that the distribution of microservices execution time is heavy-tailed. In contrast, serverless functions are shorter and providers impose timeouts on function execution times. Less directly related but an inspiration for our work is analysis of workloads on the cloud that was used to drive capacity planning and task scheduling decisions [229], [230]. A few prior studies have targeted predicting the execution time of serverless functions. For example, Sizeless [201] predicts (a point estimate) and optimizes resources for a single serverless function by building regression models from a host of synthetic functions. Another study [180] observes a variance in execution time in serverless environments, and hence, applies mixture density networks to predict the distribution of the function cost. However, its Monte-Carlo simulation mechanism is sample inefficient. WiseFuse uses a more direct method by ap-

plying statistical operations to combine the distributions of individual functions, and thus, to infer the E2E latency distribution.

Optimizing performance of serverless DAGs. Several recent proposals aim at making serverless executions more efficient, in terms of both latency and cost. [214] proposed a hybrid and dynamic approach to pick between three different intermediate data passing methods in serverless DAGs. The solution focuses only on reducing data exchange latency, and hence cannot mitigate execution skew or right size the resources to meet latency targets. Moreover, makes a major simplification, assuming that function execution times are deterministic and thus ignores runtime variances. The idea of bundling multiple parallel invocations to mitigate execution skew was proposed in ORION [231]. However, the solution does not consider Fusion or any similar technique to reduce communication latency between consecutive stages in the DAG. Moreover, ORION finds the best bundle size through trial and error rather than leveraging the performance model as done by WISEFUSE. Llama [232] considers optimizing the latency and cost for serverless video analytics pipelines. Llama is application specific and is designed for tuning video analytics parameters (including content-dependent parameters) such as sampling rate or batch size. Finally, Caerus [198] targets optimizing the latency and cost of two-stage Map-Reduce jobs by deciding when to start the execution of each mapper/reducer function. The solution applies to functions that can start execution when their input data is partially available, and hence, solves an orthogonal problem. Compared to these solutions, WISEFUSE performs both Fusion and Bundling and allocates resources to generate a cost optimized plan that meets a user-given latency target. A complementary line of work provides efficient scheduling for serverless DAGs. WUKONG [233] provides decentralized and parallel scheduling distributed across Lambda executors. It leads to serverless DAGs using the network I/O highly efficiently. Xanadu [199] and Kraken [200] tackle the problem of cascading cold starts in a dynamic DAG. Overall, no prior work in this category considers execution time distributions or combines such distributions for optimizing E2E latency or cost. Schedulers on the cloud like Apache Spark [234] and Dask [235], which deal with stateful services, have an orthogonal set of concerns, such as, state migration and cluster utilization [236], [237].

Optimizing communication latency in serverless workflows. Besides , recent work

also identifies communication latency as a major performance degrading factor in serverless workflows. Pocket [114] and Locus [99] show that current options for remote storage are either slow disk-based (*e.g.* S3) or expensive memory-based (*e.g.* ElastiCache Redis). Therefore, Pocket combines different storage media (*e.g.* DRAM, SSD, NVMe) that users can choose to conform to their application needs. SAND [118] (similar to Faastlane [213]) avoids network communication altogether by forcing all functions that belong to the same DAG to execute on the same container, thus leveraging data locality between them. While all these systems try to reduce communication latency between serverless functions, they either take an extreme approach of forcing *all* functions in a DAG to execute in the same container (sacrificing scheduling flexibility and parallelism), or they rely on users to select storage media for their applications.

Optimizing resources in serverless workflows. Photons [202] uses a technique similar to bundling restricted to DAGs with one fanout stage while focusing on security issues and reduction in the memory footprint. In contrast, WiseFuse optimizes more general DAGs with several fanout stages and cascade functions, rightsizing resources to functions and function-bundles. The concept of optimizing for a target latency in a serverless DAG is seen in Atoll [203], proactively initializing VMs to schedule function requests onto them. Proactive approaches run into the challenge of predicting when to start new containers with specific dependencies, exacerbated by unpredictable request arrival rates. Moreover, this approach is complementary to WiseFuse's DAG restructuring.

There is significant work in improving serverless runtimes and these can benefit us by reducing execution times of individual functions. Such approaches include reducing latencies by low-level system optimizations [141], [238], [239], keeping containers alive [179], [205], or quick checkpoint save and restore [240].

7.7 Discussion

Updating the profiled information: In order to keep WiseFuse's performance model current, we need to monitor the workload to detect changes, and update the profiled characteristics. WiseFuse achieves this by applying the following steps. WiseFuse models

the latency of a function as the contribution of three components: download, process, and upload, and maintains a distribution for each component. The serverless provider logs the initialization time and total execution time for each function invocation, for billing, and for continuous monitoring (*e.g.* AWS CloudWatch and Azure Cloud Monitor). Since WiseFuse already captures per-function distributions and estimates the DAG E2E latency distribution, it can detect changes to the workload characteristics or to warm-up latencies by comparing its profiled distributions to the online monitored distributions. Reprofiling is triggered when comparing statistics of the captured distribution to those of the monitored distribution indicates a change as follows: (1) when a change in a per-function statistic exceeds a threshold (10% error in P50, or 15% in P95), (2) when a change in per-DAG latency statistic or the user-specified target latency percentile exceeds similar thresholds, and (3) upon changes to the function binary or the DAG structure. When such a change is detected, WiseFuse creates a new distribution using N data samples. In many cases, the N data samples would be already available from the monitored distribution. By default $N = 300$, which we find empirically sufficient to model P95 latency with low errors. This kind of profiling for tail latencies is conceptually similar to that in OptimusCloud [138].

Larger serverless functions: WiseFuse's objective is to transform the user-defined DAG to an optimized DAG. This transformation includes colocating multiple parallel invocations together (*i.e.* Bundling), and/or fusing multiple stages together (*i.e.* Fusion). Accordingly, each node in the optimized DAG serves as a new function and hence becomes the new basic unit of scaling and scheduling. Thus, the new nodes in the transformed DAG will, expectedly, have higher memory footprints (*e.g.* due to Bundling) and/or longer execution times (*e.g.* due to Fusion) compared to nodes in the user-defined DAG. However, this is not a concern as cloud providers are increasingly supporting larger VMs and relatively long execution times (*e.g.* AWS Lambda supports up to 10 GB of memory and 15 min of execution time [241]).

Security Concerns for Fusion and Bundling: WiseFuse only bundles or fuses functions of the same DAG that belong to the same user. However, there could be security sensitive functions that should not be bundled or fused. For example, assume function A has sensitive data and function B should not have access to that data even when both functions belong

to the same DAG of the same user. In such a case, the user provides this constraint to WiseFuse so that the security-sensitive function is excluded from WiseFuse's search space.

7.8 Conclusion

We characterize production workloads of serverless DAGs at a major FaaS provider and see that serverless DAGs are increasing in popularity. We analyze the execution parameters (function execution time, skew among the parallel invocations of a function, and data transfer between functions). From our analysis, we identify two major performance bottlenecks in serverless DAGs: (1) communication latency between in-series functions (due to infrastructure reasons) (2) computation skew among in-parallel function invocations (due to data skews). We present WiseFuse that addresses these challenges through two operations — Fusion (of in-series stages) and Bundling (of in-parallel function invocations), addressing communication and computation skew, respectively. Concretely, WiseFuse uses Fusion and Bundling operations to derive an optimized execution plan that meets a user-defined latency SLA with low cost. Through experimental evaluation and comparisons with baselines and competing approaches (Faastlane, , and Photons), and using three applications, we show that WiseFuse is superior. Specifically, for an ML pipeline, WiseFuse achieves a P95 latency that is 67% lower than Photons, 39% lower than Faastlane, and 90% lower than , without increasing the $ cost. Our evaluation on AWS Lambda and GCP Functions also highlights some hitherto unknown platform-specific nuances of serverless execution.

8. Conclusion

We target the problem optimizing performance and cost for two types of cloud-native applications: NoSQL datasores and Serverless computing.

First, we highlighted the problem of tuning NoSQL datastores for dynamic workloads, as is typical for most real-world deployments such as MG-RAST. The application performance is particularly sensitive to changes to the workloads and datastore configuration parameters. We design and develop three frameworks: The first, called RAFIKI, targets the problem of *quickly* searching and identifying the best configuration parameters when the workload changes. The seconds, called SOPHIA, targets the problem of modeling the downtime observed by the servers during reconfiguration and optimizes the frequency of performing reconfiguration actions. The third, called OPTIMUSCLOUD, targets the problem of *jointly* optimizing the selection of database configurations as well as the cloud VM types and sizes. Using a novel concept of Complete Sets, OPTIMUSCLOUD provides a technique to search through the large search space brought out by heterogeneity. Configurations found by OPTIMUSCLOUD outperform those by prior works, CherryPick, Selecta, and SOPHIA, in both Perf/$ and tail latency, across two NoSQL DBMSs, Cassandra and Redis, and all experimental conditions.

Second, we highlighted the problem of optimizing the performance and cost for Serverless DAG applications. We characterize production workloads of serverless DAGs at a major cloud provider. Our analysis highlights two major factors that limit performance: (a) lack of efficient communication methods between the serverless functions in the DAG, and (b) stragglers when a DAG stage invokes a set of parallel functions that must complete before starting the next DAG stage. We proposed three frameworks to address these challenges. The first, called SONIC, selects the best data passing method between each pair of in-series stages in the DAG to minimize the DAGs latency and cost. The seconds, called ORION, models the performance variability for each function in the DAG as well as the entire DAG by representing latency as a distribution. Hence, using statistical distribution combination operations (e.g. Conv and Max), it models the latency distribution for entire DAG. This allows for providing service level performance objectives with reduced cost. Finally, we

proposed WiseFuse, which uses Fusion and Bundling operations to derive an optimized execution plan that meets a user-defined latency SLA with low cost. Through experimental evaluation and comparisons with baselines and competing approaches (Faastlane, Sonic, and Photons), and using three applications, we show that WiseFuse is superior. Specifically, for an ML pipeline, WiseFuse achieves a P95 latency that is 67% lower than Photons, 39% lower than Faastlane, and 90% lower than Sonic, without increasing the $ cost. Our evaluation on AWS Lambda and GCP Functions also highlights some hitherto unknown platform-specific nuances of serverless execution.

REFERENCES

[1] C. E. Cook, M. T. Bergman, R. D. Finn, G. Cochrane, E. Birney, and R. Apweiler, "The european bioinformatics institute in 2016: Data growth and integration," *Nucleic acids research*, vol. 44, no. D1, pp. D20–D26, 2016.

[2] D. Shasha and P. Bonnet, *Database tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann, 2002.

[3] S. Chaudhuri and V. Narasayya, "Self-tuning database systems: A decade of progress," in *Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment, 2007, pp. 3–14.

[4] Y. Zhang, T. Cao, S. Li, X. Tian, L. Yuan, H. Jia, and A. V. Vasilakos, "Parallel processing systems for big data: A survey," *Proceedings of the IEEE*, vol. 104, no. 11, pp. 2114–2136, 2016.

[5] Y. Guo, P. Lama, C. Jiang, and X. Zhou, "Automated and agile server parametertuning by coordinated learning and control," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 4, pp. 876–886, 2014.

[6] Z. Bei, Z. Yu, Q. Liu, C. Xu, S. Feng, and S. Song, "Mest: A model-driven efficient searching approach for mapreduce self-tuning," *IEEE Access*, vol. 5, pp. 3580–3593, 2017.

[7] Z. Bei, Z. Yu, H. Zhang, W. Xiong, C. Xu, L. Eeckhout, and S. Feng, "Rfhoc: A random-forest approach to auto-tuning hadoop's configuration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1470–1483, 2016.

[8] G. R. Iversen and H. Norpoth, *Analysis of variance*, 1. Sage, 1987.

[9] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, ACM, 2014, pp. 303–316.

[10] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ACM, 2017, pp. 1009–1024.

[11] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 1246–1257, 2009.

[12] A. Wilke, J. Bischof, W. Gerlach, E. Glass, T. Harrison, K. P. Keegan, T. Paczian, W. L. Trimble, S. Bagchi, A. Grama, *et al.*, "The mg-rast metagenomics database and portal in 2015," *Nucleic acids research*, vol. 44, no. D1, pp. D590–D594, 2016.

[13] L. Shi, Z. Wang, W. Yu, and X. Meng, "A case study of tuning mapreduce for efficient bioinformatics in the cloud," *Parallel Computing*, vol. 61, pp. 83–95, 2017.

[14] R. Cattell, "Scalable sql and nosql data stores," *ACM Sigmod Record*, vol. 39, no. 4, pp. 12–27, 2011.

[15] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[17] E. Point, *The Write Path to Compaction*, http://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml_write_path_c.html#concept_ds_wt3_32w_zj__dml-compaction, 2017.

[18] I. Csiszar and J. Körner, *Information theory: coding theorems for discrete memoryless systems*. Cambridge University Press, 2011.

[19] D. Enterprise, *The cassandra.yaml configuration file*, http://docs.datastax.com/en/cassandra/3.0/cassandra/configuration/configCassandra_yaml.html, [Online; accessed 1-May-2018].

[20] D. Enterprise, *Apache Cassandra: Configuring compaction*, http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsConfigureCompaction.html, [Online; accessed 1-May-2018].

[21] Alteroot, *How to change Cassandra compaction strategy on a production cluster*, https://blog.alteroot.org/articles/2015-04-20/change-cassandra-compaction-strategy-on-production-cluster.html, [Online; accessed 1-May-2018].

[22] S. Overflow, *Can Cassandra compaction strategy be changed dynamically?* http://stackoverflow.com/questions/26640385/can-cassandra-compaction-strategy-be-changed-dynamically, [Online; accessed 1-May-2018].

[23] K. Deb, "An efficient constraint handling method for genetic algorithms," *Computer methods in applied mechanics and engineering*, vol. 186, no. 2, pp. 311–338, 2000.

[24] K. Deep, K. P. Singh, M. L. Kansal, and C. Mohan, “A real coded genetic algorithm for solving integer and mixed integer optimization problems,” *Applied Mathematics and Computation*, vol. 212, no. 2, pp. 505–518, 2009.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proceedings of the 1st ACM symposium on Cloud computing*, ACM, 2010, pp. 143–154.

[26] *Matlab neural network toolbox*, http://http://metagenomics.anl.gov//, [Online; accessed 27-April-2018].

[27] ScyllaDB, *Scylla release: version 1.6*, http://www.scylladb.com/2017/02/06/scylla-release-version-1-6/, Feb. 2017.

[28] A. Mahgoub, P. Wood, S. Ganesh, S. Mitra, W. Gerlach, T. Harrison, F. Meyer, A. Grama, S. Bagchi, and S. Chaterji, “Rafiki: A middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads,” in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, ACM, 2017, pp. 28–40.

[29] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, “Bestconfig: Tapping the performance potential of systems via automatic configuration tuning,” in *Symposium on Cloud Computing (SoCC)*, 2017.

[30] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. Tung, “A new approach to dynamic self-tuning of database buffers,” *ACM Transactions on Storage (TOS)*, 2008.

[31] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer, *Using probabilistic reasoning to automate software tuning*, 1. ACM, 2004, vol. 32.

[32] F. Meyer, S. Bagchi, S. Chaterji, W. Gerlach, A. Grama, T. Harrison, W. Trimble, and A. Wilke, “Mg-rast version 4–lessons learned from a decade of low-budget ultra-high-throughput metagenome analysis,” *Briefings in Bioinformatics*, vol. 105, 2017.

[33] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, “Query-based workload forecasting for self-driving database management systems,” in *Proceedings of the 2018 International Conference on Management of Data*, ACM, 2018, pp. 631–645.

[34] C. Faloutsos, J. Gasthaus, T. Januschowski, and Y. Wang, “Forecasting big time series: Old and new,” *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 2102–2105, 2018.

[35] B. Mozafari, C. Curino, A. Jindal, and S. Madden, “Performance and resource modeling in highly-concurrent oltp workloads,” in *Proceedings of the 2013 acm sigmod international conference on management of data*, ACM, 2013, pp. 301–312.

[36] DCSL, *Rafiki configuration tuner middleawre*, https://engineering.purdue.edu/dcsl/software/, Feb. 2018.

[37] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[38] J. L. Carlson, *Redis in action*. Manning Publications Co., 2013.

[39] S. Chaterji, J. Koo, N. Li, F. Meyer, A. Grama, and S. Bagchi, "Federation in genomics pipelines: Techniques and challenges," *Briefings in bioinformatics*, vol. 20, no. 1, pp. 235–244, 2017.

[40] F. Meyer, S. Bagchi, S. Chaterji, W. Gerlach, A. Grama, T. Harrison, T. Paczian, W. L. Trimble, and A. Wilke, "MG-RAST version 4—lessons learned from a decade of low-budget ultra-high-throughput metagenome analysis," *Briefings in bioinformatics*, 2017.

[41] S. G. Kim, M. Harwani, A. Grama, and S. Chaterji, "EP-DNN: A deep neural network-based global enhancer prediction algorithm," *Scientific reports*, vol. 6, p. 38 433, 2016.

[42] J. Koo, J. Zhang, and S. Chaterji, "Tiresias: Context-sensitive approach to decipher the presence and strength of MicroRNA regulatory interactions," *Theranostics*, vol. 8, no. 1, p. 277, 2018.

[43] S. G. Kim, N. Theera-Ampornpunt, C.-H. Fang, M. Harwani, A. Grama, and S. Chaterji, "Opening up the blackbox: An interpretable deep neural network-based classifier for cell-type specific enhancer predictions," *BMC systems biology*, vol. 10, no. 2, p. 54, 2016.

[44] H. Scheffe, *The analysis of variance*. John Wiley & Sons, 1999, vol. 72.

[45] K. Mahadik, C. Wright, J. Zhang, M. Kulkarni, S. Bagchi, and S. Chaterji, "Sarvavid: A domain specific language for developing scalable computational genomics applications," in *International Conference on Supercomputing*, ACM, 2016, p. 34.

[46] D. Featherston, "Cassandra: Principles and application," *Department of Computer Science University of Illinois at Urbana-Champaign*, 2010.

[47] D. K. Gifford, "Weighted voting for replicated data," in *SOSP*, 1979.

[48] L. Ma, *Tiramisu: Dataset for bus tracking applications*, http://www.cs.cmu.edu/~malin199/data/tiramisu-sample/, [Online; accessed 1-September-2018], Jun. 2018.

[49] R. L. Henderson, "Job scheduling under the portable batch system," in *Workshop on Job Scheduling Strategies for Parallel Processing*, Springer, 1995, pp. 279–294.

[50] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *Proceedings of the 7th ACM european conference on Computer Systems*, ACM, 2012, pp. 99–112.

[51] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, 2012, pp. 21–21.

[52] Medium, *Cassandra: Distributed key-value store optimized for write-heavy workloads*, https://medium.com/coinmonks/cassandra-distributed-key-value-store-optimized-for-write-heavy-workloads-77f69c01388c, [Online; accessed 1-May-2019].

[53] ScyllaDB, *ScyllaDB*, http://www.scylladb.com/, Sep. 2017.

[54] ScyllaDB, *Scylla vs. Cassandra benchmark*, http://www.scylladb.com/technology/cassandra-vs-scylla-benchmark-2/, Oct. 2015.

[55] A. Khandelwal, R. Agarwal, and I. Stoica, "Blowfish: Dynamic storage-performance tradeoff in data stores.," in *14th USENIX Symposium on Networked Systems Design and Implementation NSDI'16*, 2016, pp. 485–500.

[56] J. Leipzig, "A review of bioinformatic pipeline frameworks," *Briefings in bioinformatics*, vol. 18, no. 3, pp. 530–536, 2017.

[57] A. N. Lab and U. of Chicago, *MG-RAST's m5nr-table schema*, [Online; accessed 29-August-2018], 2019.

[58] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10, Boston, MA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[59] R. Labs, *Spark-Redis: Analytics Made Lightning Fast*, https://redislabs.com/solutions/use-cases/spark-and-redis/, [Online; accessed 1-May-2019], 2019.

[60] R. Labs, *Redis Virtual Memory*, https://redis.io/topics/virtual-memory, [Online; accessed 1-September-2018], 2018.

[61] Serverfault, *Should I use vm or set the maxmemory*, https://serverfault.com/questions/432810/should-i-use-vm-or-set-the-maxmemory-with-redis-2-4, Sep. 2012.

[62] Groups.google, *Problem with restart redis with vm feature on*, https://groups.google.com/forum/#!topic/redis-db/EQA0WdvwghI, Mar. 2011.

[63] Stackoverflow, *Redis Virtual Memory in 2.6*, https://stackoverflow.com/questions/9205597/redis-virtual-memory-in-2-6, Feb. 2012.

[64] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, ACM, 2010, pp. 143–154.

[65] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2013, pp. 66–76.

[66] H. Greenberg, J. Bent, and G. Grider, "{mdhim}: A parallel key/value framework for {hpc}," in *7th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.

[67] R. Labs, *Redis Cluster*, https://redis.io/topics/cluster-tutorial, [Online; accessed 1-May-2019], 2019.

[68] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach," in *Proceedings of the 2017 Symposium on Cloud Computing*, ACM, 2017, pp. 452–465.

[69] Argonne National Laboratory, *MG-RAST: Metagenomics Analysis Server*, urlhttps://www.mg-rast.org/, 2019.

[70] S. Chaterji, J. Koo, N. Li, F. Meyer, A. Grama, and S. Bagchi, "Federation in genomics pipelines: Techniques and challenges," *Briefings in bioinformatics*, vol. 20, no. 1, pp. 235–244, 2019.

[71] A. Mahgoub, P. Wood, A. Medoff, S. Mitra, F. Meyer, S. Chaterji, and S. Bagchi, "SOPHIA: Online reconfiguration of Clustered NoSQL Databases for Time-Varying Workloads," in *2019 USENIX Annual Technical Conference ATC'19*, Usenix, 2019, pp. 223–240.

[72] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation NSDI'17*, 2017, pp. 469–482.

[73] A. Klimovic, H. Litz, and C. Kozyrakis, "Selecta: Heterogeneous cloud storage configuration for data analytics," in *2018 USENIX Annual Technical Conference ATC'18*, 2018, pp. 759–773.

[74] B. Kemme, A. Bartoli, and O. Babaoglu, "Online reconfiguration in replicated databases based on group communication," in *Dependable Systems and Network (DSN)*, IEEE, 2001, pp. 117–126.

[75] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SIGOPS operating systems review*, ACM, vol. 41, 2007, pp. 205–220.

[76] R. Klophaus, "Riak core: Building distributed applications without shared state," in *ACM SIGPLAN Commercial Users of Functional Programming*, ACM, 2010, p. 14.

[77] Clutch, *Data replication in cassandra*, https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeReplication.html, 2019.

[78] Redis, *Using redis as an lru cache*, https://redis.io/topics/lru-cache, 2019.

[79] Redis, *Redis cluster tutorial*, https://redis.io/topics/cluster-tutoriallpp, Feb. 2015.

[80] AWS, *Amazon EC2*, https://aws.amazon.com/ec2/pricing/on-demand/, May 2018.

[81] Datastax, *Cassandra dynamic snitching*, https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archSnitchDynamic.html, 2019.

[82] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik, "Learning-based query performance modeling and prediction," in *2012 IEEE 28th International Conference on Data Engineering*, IEEE, 2012, pp. 390–401.

[83] G. Kousiouris, T. Cucinotta, and T. Varvarigou, "The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks," *Journal of Systems and Software*, vol. 84, no. 8, pp. 1270–1291, 2011.

[84] N. I. of Standards and T. (NIST), *D-optimal designs*, https://itl.nist.gov/div898/handbook/pri/section5/pri521.htm, 2019.

[85] A. Palczewska, J. Palczewski, R. M. Robinson, and D. Neagu, "Interpreting random forest classification models using a feature contribution method," in *Integration of reusable systems*, Springer, 2014, pp. 193–218.

[86] I. Konstantinou, D. Tsoumakos, I. Mytilinis, and N. Koziris, "Dbalancer: Distributed load balancing for nosql data-stores," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ACM, 2013, pp. 1037–1040.

[87] J. Koo, J. Zhang, and S. Chaterji, "Tiresias: Context-sensitive approach to decipher the presence and strength of microrna regulatory interactions," *Theranostics*, vol. 8, no. 1, p. 277, 2018.

[88] A. Ghoshal, A. Grama, S. Bagchi, and S. Chaterji, "An ensemble svm model for the accurate prediction of non-canonical microrna targets," in *Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics*, 2015, pp. 403–412.

[89] S. Koziel and X.-S. Yang, *Computational optimization, methods and algorithms*. Springer, 2011, vol. 356.

[90] G. F. W and K. G. A, *Handbook of metaheuristics*. Springer Science & Business Media, 2006, vol. 57.

[91] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *computer*, vol. 27, no. 6, pp. 17–26, 1994.

[92] Solid, *Solid:a comprehensive gradient-free optimization framework written in python*, https://github.com/100/Solid, 2019.

[93] scikitlearn, *Scikit-learn: Machine learning in python*, https://scikit-learn.org/stable/, 2019.

[94] *Best practices for running apache cassandra on amazon ec2*, https://aws.amazon.com/blogs/big-data/best-practices-for-running-apache-cassandra\-on-amazon-ec2/, [Online; accessed 17-September-2019].

[95] *5 tips for running redis over aws*, https://redislabs.com/blog/5-tips-for-running-redis-over-aws/, [Online; accessed 17-September-2019].

[96] N. Hug, *Surprise, a python library for recommender systems*, http://surpriselib.com, 2017.

[97] K. Beyls and E. D'Hollander, "Reuse distance as a metric for cache behavior," in *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, vol. 14, 2001, pp. 350–360.

[98] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *ACM Sigplan Notices*, ACM, vol. 38, 2003, pp. 245–257.

[99] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 193–206.

[100] Y. Kim and J. Lin, "Serverless data analytics with flint," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 451–455.

[101] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "A case for serverless machine learning," in *Workshop on Systems for ML and Open Source Software at NeurIPS*, vol. 2018, 2018.

[102] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, "Towards a serverless platform for edge {ai}," in *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[103] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 363–376.

[104] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 263–274.

[105] R. Xu, C.-l. Zhang, P. Wang, J. Lee, S. Mitra, S. Chaterji, Y. Li, and S. Bagchi, "Approxdet: Content and contention-aware approximate object detection for mobiles," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020, pp. 449–462.

[106] Amazon, *Aws step functions: Assemble functions into business-critical applications*, https://aws.amazon.com/step-functions/, Last retrieved: Jan, 2021.

[107] Microsoft, *Azure durable functions overview*, https://docs.microsoft.com/en-us/azure/azure-functions/durable/, Last retrieved: Jan, 2021.

[108] Google, *Cloud composer: A fully managed workflow orchestration service built on apache airflow*, https://cloud.google.com/composer, Last retrieved: Jan, 2021.

[109] T. Elgamal, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, IEEE, 2018, pp. 300–312.

[110] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, "Understanding ephemeral storage for serverless analytics," in *2018 USENIX Annual Technical Conference (USENIXATC 18)*, 2018, pp. 789–794.

[111] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 115–130.

[112] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[113] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden, "Starling: A scalable query engine on cloud functions," in *SIGMOD*, 2020.

[114] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 427–444.

[115] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 13–24.

[116] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, *et al.*, "Protean:{vm} allocation service at scale," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 845–861.

[117] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[118] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018, pp. 923–935.

[119] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pp. 205–218, ISBN: 978-1-939133-14-4. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad.

[120] B. Raupach, *Choosing the right amount of memory for your aws lambda function*, https://medium.com/@raupach/choosing-the-right-amount-of-memory-for-your-aws-lambda-function-99615ddf75dd, 2018.

[121] R. RIBENZAFT, *How to make aws lambda faster: Memory performance*, https://epsagon.com/observability/how-to-make-aws-lambda-faster-memory-performance/, 2018.

[122] A. Forums, *How to set memory size of azure function?* https://social.msdn.microsoft.com/Forums/en-US/9a6e4728-d54a-488d-9007-5fdb80fc105e/how-to-set-memory-size-of-azure-function?forum=AzureFunctions, 2018.

[123] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "{graphene}: Packing and dependency-aware scheduling for data-parallel clusters," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 81–97.

[124] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu, "Allox: Compute allocation in hybrid clusters," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[125] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types.," in *Nsdi*, vol. 11, 2011, pp. 24–24.

[126] D. L. Banks and S. E. Fienberg, "Multivariate statistics," 2003.

[127] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 153–167.

[128] G. D. Forney, "The viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[129] G. D. Forney Jr, "The viterbi algorithm: A personal history," *arXiv preprint cs/0504020*, 2005.

[130] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Scalable atomic visibility with ramp transactions," *ACM Transactions on Database Systems (TODS)*, vol. 41, no. 3, pp. 1–45, 2016.

[131] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A fault-tolerance shim for serverless computing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–15.

[132] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, IEEE, 2011, pp. 454–465.

[133] OpenLambda, *An open source serverless computing platform*, https://github.com/open-lambda/open-lambda, 2021.

[134] AWS, *Aws lambda faqs*, https://aws.amazon.com/lambda/faqs/, 2021.

[135] A. CloudWatch, *Monitoring ec2 network utilization*, https://cloudonaut.io/monitoring-ec2-network-utilization/, 2020.

[136] V. J. Ribeiro, R. H. Riedi, R. G. Baraniuk, J. Navratil, and L. Cottrell, "Pathchirp: Efficient available bandwidth estimation for network paths," in *Passive and active measurement workshop*, 2003.

[137] H. Wang, K. S. Lee, E. Li, C. L. Lim, A. Tang, and H. Weatherspoon, "Timing is everything: Accurate, minimum overhead, available bandwidth estimation in high-speed wired networks," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, 2014, pp. 407–420.

[138] A. Mahgoub, A. M. Medoff, R. Kumar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "{optimuscloud}: Heterogeneous configuration optimization for distributed databases in the cloud," in *USENIX Annual Technical Conference (USENIX ATC)*, 2020, pp. 189–203.

[139] C.-J. Hsu, V. Nair, T. Menzies, and V. W. Freeh, "Scout: An experienced guide to find the best cloud configuration," *arXiv preprint arXiv:1803.01296*, 2018.

[140] D. Z. Tootaghaj, F. Farhat, M. Arjomand, P. Faraboschi, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Evaluating the combined impact of node architecture and cloud workload characteristics on network traffic and performance/cost," in *2015 IEEE International Symposium on Workload Characterization*, IEEE, 2015, pp. 203–212.

[141] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "{sock}: Rapid task provisioning with serverless-optimized containers," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 57–70.

[142] MXNet, *Using pre-trained deep learning models in mxnet*, https://mxnet.apache.org/api/python/docs/tutorials/packages/gluon/image/pretrained_models.html, 2021.

[143] LightGBM, *Lightgbm's documentation!* https://lightgbm.readthedocs.io/en/latest/index.html, 2021.

[144] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[145] P. J. Grother, "Nist special database 19 handprinted forms and characters database," *National Institute of Standards and Technology*, 1995.

[146] Amazon, *Aws lambda features*, https://aws.amazon.com/lambda/features/, 2021.

[147] X. Yi, D. Park, Y. Chen, and C. Caramanis, "Fast algorithms for robust pca via gradient descent," in *Advances in neural information processing systems*, 2016, pp. 4152–4160.

[148] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the gap between serverless and its state with storage functions," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 1–12.

[149] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[150] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *2019 USENIX Annual Technical Conference*, 2019, pp. 475–488.

[151] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, *et al.*, "Ray: A distributed framework for emerging {ai} applications," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 561–577.

[152] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the faas track: Building stateful distributed applications with serverless architectures," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 41–54.

[153] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "Numpywren: Serverless linear algebra," in *ACM Symposium on Cloud Computing (SoCC)*, 2020, pp. 1–14.

[154] A. Mahgoub, P. Wood, A. Medoff, S. Mitra, F. Meyer, S. Chaterji, and S. Bagchi, "{sophia}: Online reconfiguration of clustered nosql databases for time-varying workloads," in *2019 USENIX Annual Technical Conference (USENIXATC 19)*, 2019, pp. 223–240.

[155] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, *Cloudburst: Stateful functions-as-a-service*, https://arxiv.org/pdf/2001.04592.pdf, 2020.

[156] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: Skip redundant paths to make serverless fast," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–15.

[157] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 419–434.

[158] J. Donkervliet, A. Trivedi, and A. Iosup, “Towards supporting millions of users in modifiable virtual environments by redesigning minecraft-like games as serverless systems,” in *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[159] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, “Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud,” in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, IEEE, 2019, pp. 199–208.

[160] J. R. Gunasekaran, P. Thinakaran, N. Chidambaram, M. T. Kandemir, and C. R. Das, “Fifer: Tackling underutilization in the serverless era,” *arXiv preprint arXiv:2008.12819*, 2020.

[161] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, *et al.*, “Spark: Cluster computing with working sets.,” *HotCloud*, vol. 10, no. 10-10, 2010.

[162] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” in *Proceedings of the 2007 Eurosys Conference*, Association for Computing Machinery, Inc., Mar. 2007.

[163] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, “Ciel: A universal execution engine for distributed data-flow computing,” in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’11, 2011, pp. 113–126.

[164] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, “Learning scheduling algorithms for data processing clusters,” in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 270–288.

[165] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, “Lavea: Latency-aware video analytics on edge computing platform,” in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–13.

[166] H. Wang, D. Niu, and B. Li, “Distributed machine learning with a serverless architecture,” in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, IEEE, 2019, pp. 1288–1296.

[167] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, “Serverless linear algebra,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 281–295.

[168] V. Gupta, S. Kadhe, T. Courtade, M. W. Mahoney, and K. Ramchandran, "Oversketched newton: Fast convex optimization for serverless systems," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 288–297. DOI: 10.1109/BigData50022.2020.9378289.

[169] Q. Zhang, H. Sun, X. Wu, and H. Zhong, "Edge video analytics for public safety: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1675–1696, 2019.

[170] L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring serverless computing for neural network training," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, 2018, pp. 334–341.

[171] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 133–146.

[172] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 285–301, ISBN: 978-1-939133-23-6. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/mahgoub.

[173] Azure, *Durable functions overview*, https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp, Last retrieved: May, 2022.

[174] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, IEEE, 2020, pp. 129–138.

[175] M. Bilal, M. Serafini, M. Canini, and R. Rodrigues, "Do the best cloud configurations grow on trees? an experimental evaluation of black box algorithms for optimizing cloud workloads," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2563–2575, 2020.

[176] python, *Chatbots: Intent recognition dataset*, https://www.kaggle.com/elvinagammed/chatbots-intent-recognition-dataset, Last retrieved: May, 2022.

[177] F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li, "Quantifying cloud elasticity with container-based autoscaling," *Future Generation Computer Systems*, vol. 98, pp. 672–681, 2019.

[178] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera, "Burst-aware predictive autoscaling for containerized microservices," *IEEE Transactions on Services Computing*, 2020.

[179] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC)*, 2020, pp. 205–218.

[180] S. Eismann, J. Grohmann, E. van Eyk, N. Herbst, and S. Kounev, "Predicting the costs of serverless workflows," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 265–276.

[181] Amazon, *Configuring functions in the aws lambda console*, https://docs.aws.amazon.com/lambda/latest/dg/configuration-console.html, 2021.

[182] Wikipedia, *Best-first search*, https://en.wikipedia.org/wiki/Best-first_search, Last retrieved: May, 2022.

[183] Y. Kwon, K. Ren, M. Balazinska, B. Howe, and J. Rolia, "Managing skew in hadoop.," *IEEE Data Eng. Bull.*, vol. 36, no. 1, pp. 24–33, 2013.

[184] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "GRASS: Trimming stragglers in approximation analytics," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA: USENIX Association, Apr. 2014, pp. 289–302, ISBN: 978-1-931971-09-6. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/ananthanarayanan.

[185] L. Bindschaedler, J. Malicevic, N. Schiper, A. Goel, and W. Zwaenepoel, "Rock you like a hurricane: Taming skew in large scale analytics," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–15.

[186] J. Teoh, M. A. Gulzar, G. H. Xu, and M. Kim, "Perfdebug: Performance debugging of computation skew in dataflow systems," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 465–476.

[187] Amazon Web Services, *Parallel Processing in Python with AWS Lambda*, https://aws.amazon.com/blogs/compute/parallel-processing-in-python-with-aws-lambda/, Last retrieved: May, 2022.

[188] Amazon, *Aws step functions documentation*, https://docs.aws.amazon.com/step-functions/index.html, Last retrieved: May, 2022.

[189] Amazon, *Parallel processing in python with aws lambda*, https://aws.amazon.com/blogs/compute/parallel-processing-in-python-with-aws-lambda/, Last retrieved: May, 2022.

[190] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[191] LightGBM, *Lightgbm python-package*, https://lightgbm.readthedocs.io/en/latest/Python-Intro.html, Last retrieved: May, 2022.

[192] DeepQuest-AI, *Imageai : Video object detection, tracking and analysis*, https://github.com/OlafenwaMoses/ImageAI/blob/master/imageai/Detection/VIDEO.md, Last retrieved: May, 2022.

[193] Q. Chen, J. Yao, and Z. Xiao, "Libra: Lightweight data skew mitigation in mapreduce," *IEEE Transactions on parallel and distributed systems*, vol. 26, no. 9, pp. 2520–2533, 2014.

[194] Spark, *Spark speculation*, https://spark.apache.org/docs/latest/configuration.html, Last retrieved: May, 2022.

[195] Y. Guo, J. Rao, C. Jiang, and X. Zhou, "Moving hadoop into the cloud with flexible slot management and speculative execution," *IEEE Transactions on Parallel and Distributed systems*, vol. 28, no. 3, pp. 798–812, 2016.

[196] *Pre-warming Policy Simulator*, https://github.com/icanforce/Orion-OSDI22, Last retrieved: May, 2022.

[197] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 311–327.

[198] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: Nimble task scheduling for serverless analytics.," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021, pp. 653–669.

[199] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 356–370.

[200] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 153–167.

[201] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 248–259.

[202] V. Dukic, R. Bruno, A. Singla, and G. Alonso, "Photons: Lambdas on a diet," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 45–59.

[203] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A scalable low-latency serverless platform," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 138–152.

[204] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *13th USENIX Symposium on Networked Systems Design and Implementation NSDI'16*, 2016, pp. 363–378.

[205] A. Fuerst and P. Sharma, "Faascache: Keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 386–400.

[206] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 1–13.

[207] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[208] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. Greenberg, "Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services," *Performance Evaluation*, vol. 68, no. 11, pp. 1056–1071, 2011, Special Issue: Performance 2011, ISSN: 0166-5316. DOI: https://doi.org/10.1016/j.peva.2011.07.015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166531611001064.

[209] Amazon, *Lambda quotas*, https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, Last retrieved: May, 2022.

[210] B. Schroeder and M. Harchol-Balter, "Web servers under overload: How scheduling can help," *ACM Trans. Internet Technol.*, vol. 6, no. 1, pp. 20–52, Feb. 2006, ISSN: 1533-5399. DOI: 10.1145/1125274.1125276. [Online]. Available: https://doi.org/10.1145/1125274.1125276.

[211] M. E. Crovella, R. Frangioso, and M. Harchol-Balter, "Connection scheduling in web servers," Boston University Computer Science Department, Tech. Rep., 1999.

[212] M. Harchol-Balter, M. E. Crovella, and C. D. Murta, "On choosing a task assignment policy for a distributed server system," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 204–228, 1999, ISSN: 0743-7315. DOI: https://doi.org/10.1006/jpdc.1999.1577. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731599915770.

[213] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating Function-as-a-Service workflows," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX Association, Jul. 2021, pp. 805–820, ISBN: 978-1-939133-23-6. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/kotni.

[214] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX Association, Jul. 2021, pp. 285–301, ISBN: 978-1-939133-23-6. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/mahgoub.

[215] *Azure Serverless DAG traces (public dataset)*, https://github.com/Azure/AzurePublicDataset, 2022.

[216] Google, *Google cloud: Configuring memory limits*, https://cloud.google.com/run/docs/configuring/memory-limits, Last retrieved: May, 2022.

[217] V. Kalantzis, G. Kollias, S. Ubaru, A. N. Nikolakopoulos, L. Horesh, and K. Clarkson, "Projection techniques to update the truncated SVD of evolving matrices with applications," in *Proceedings of the 38th International Conference on Machine Learning*, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, 18–24 Jul 2021, pp. 5236–5246. [Online]. Available: https://proceedings.mlr.press/v139/kalantzis21a.html.

[218] YongchangWang and L. Zhu, "Research and implementation of SVD in machine learning," in *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*, 2017, pp. 471–475. DOI: 10.1109/ICIS.2017.7960038.

[219] Y. Song, N. Sebe, and W. Wang, "Why approximate matrix square root outperforms accurate svd in global covariance pooling?" In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 1115–1123.

[220] T. Hastie, R. Mazumder, J. D. Lee, and R. Zadeh, "Matrix completion and low-rank svd via fast alternating least squares," *The Journal of Machine Learning Research*, vol. 16, no. 1, pp. 3367–3402, 2015.

[221] X. Ren and D. Ramanan, "Histograms of sparse codes for object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2013, pp. 3246–3253.

[222] Netflix, *Netflix prize data*, https://www.kaggle.com/netflix-inc/netflix-prize-data, 2021.

[223] F. Liang, R. Shi, and Q. Mo, "A split-and-merge approach for singular value decomposition of large-scale matrices," *Statistics and its interface*, vol. 9, no. 4, p. 453, 2016.

[224] Google, *Google Cloud Function Quotas*, https://cloud.google.com/functions/quotas#footnote, 2022.

[225] Google, *Quota Limits for Google Cloud Functions*, https://cloud.google.com/workflows/quotas, 2022.

[226] Google, *Cloud Pub/Sub*, https://cloud.google.com/pubsub, 2022.

[227] Google, *Pricing in google cloud function*, https://cloud.google.com/functions/pricing, Last retrieved: May, 2022.

[228] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 412–426.

[229] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: Insights from google compute clusters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4, pp. 34–41, 2010.

[230] I. S. Moreno, P. Garraghan, P. Townend, and J. Xu, "An approach for characterizing workloads in google cloud to derive realistic resource utilization models," in *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, IEEE, 2013, pp. 49–60.

[231] A. Mahgoub, E. B. Yi, K. Shankar, S. Chaterji, S. Elnikety, and S. Bagchi, "ORION and the Three Rights: Sizing, co-location, and prewarming for serverless dags," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 1–14.

[232] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," in *Proceedings of the ACM Symposium on Cloud Computing*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1–17, ISBN: 9781450386388. [Online]. Available: https://doi.org/10.1145/3472883.3486972.

[233] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 1–15.

[234] A. Foundation, *Job scheduling - spark 3.2.1 documentation*, https://spark.apache.org/docs/latest/job-scheduling.html, Last retrieved: Jan, 2022.

[235] A. Inc., *Scheduling - dask documentation*, https://docs.dask.org/en/stable/scheduling.html, Last retrieved: Jan, 2022.

[236] C. Fuerst, S. Schmid, L. Suresh, and P. Costa, "Kraken: Towards elastic performance guarantees in multi-tenant data centers," in *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2015, pp. 433–434.

[237] L. Zheng, C. Joe-Wong, C. G. Brinton, C. W. Tan, S. Ha, and M. Chiang, "On the viability of a cloud virtual service provider," *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1, pp. 235–248, 2016.

[238] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 419–433.

[239] Z. Jia and E. Witchel, "Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 152–166.

[240] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 467–481.

[241] Amazon, *Lambda quotas*, https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, 2022.