Model-Driven Computational Sprinting

Nathaniel Morris
The Ohio State University
Columbus, Ohio

Christopher Stewart
The Ohio State University
Columbus, Ohio

Lydia Chen
IBM Research
Zürich, Switzerland

Robert Birke
ABB Corporate Research
Baden, Switzerland

Jaimie Kelley
Denison University
Granville, Ohio

ABSTRACT

Computational sprinting speeds up query execution by increasing power usage for short bursts. *Sprinting policy* decides when and how long to sprint. Poor policies inflate response time significantly. We propose a model-driven approach that chooses between sprinting policies based on their expected response time. However, sprinting alters query executions at runtime, creating a complex dependency between queuing and processing time. Our performance modeling approach employs offline profiling, machine learning, and first-principles simulation. Collectively, these modeling techniques capture the effects of sprinting on response time. We validated our modeling approach with 3 sprinting mechanisms across 9 workloads. Our performance modeling approach predicted response time with median error below 4% in most tests and median error of 11% in the worst case. We demonstrated model-driven sprinting for cloud providers seeking to colocate multiple workloads on AWS Burstable Instances while meeting service level objectives. Model-driven sprinting uncovered policies that achieved response time goals, allowing more workloads to colocate on a node. Compared to AWS Burstable policies, our approach increased revenue per node by 1.6X.

KEYWORDS

Computational Sprinting, Resource Management, CPU Throttling, System Models, Simulation, Random Decision Forest, Queuing Models, Accuracy, Prediction, Burstable Instance, Amazon Web Services

ACM Reference Format:
Nathaniel Morris, Christopher Stewart, Lydia Chen, Robert Birke, and Jaimie Kelley. 2018. Model-Driven Computational Sprinting. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3190508.3190543

1 INTRODUCTION

Modern processors are constrained by increasingly tight power caps [32]. Computational sprinting is a resource management approach that speeds up workload execution by using power reserves to boost processing rates above sustained rates for short bursts [31]. Sprinting helps workloads meet response time requirements specified by SLOs [19, 35] and interactive applications [31]. DVFS [19, 44], core scaling [18], and CPU throttling [43] are commonly used to implement sprinting.

This paper examines sprinting for workloads comprising independent query executions (e.g., cloud servers) where all executions share the power budget reserved for sprinting. In this context, sprinting should target queries that most improve whole-system response time [19]. Consider two Spark query executions that compute K-means clustering. The first query arrives when no other queries are in the system. The second arrives during a busy period. On an Intel Xeon 2660, DVFS sprinting can speed up Spark K-means queries by 97%. But what if the sprinting budget afforded only one of the query executions? In this case, the second query execution should sprint; speeding up its own execution and reducing time other queries spend queuing. Generalizing from this example, cloud servers can use sprinting to improve response time by speeding up individual query executions and by reducing queuing delays.

Sprinting policies govern which queries to speed up by setting (1) timer interrupts that trigger sprinting for a query execution, called timeouts [18, 26, 37], (2) processing speed during sprinting, called sprint rate and (3) sprinting budget for a given sprinting mechanism. Sprinting policies have complicated effects on response time. Figure 1 depicts query execution under a sprinting policy where the timeout is 1 minute. In this example, timeouts trigger sprinting for queries 1 and 2, draining the budget. The remaining queries must execute at the sustained processing rate. Here, sprinting is applied too aggressively to early arrivals which causes queuing delays for queries 4, 5 and 6. However, increasing the timeout has mixed effects. A 3-minute timeout counterintuitively degrades response time, because it is too conservative and does not exhaust the sprinting budget. Under a 2-minute timeout, response time improves by 25%. This example shows that subtle changes in sprinting policies can significantly affect response time.

Model-driven computational sprinting uses performance models to set sprinting policies. Performance models map policies and workload conditions to response time. Precisely, sprinting policies include sustained processing rate, sprint rate, timeout, budget, etc. Workload conditions include query semantics, arrival rate, etc. Model-driven sprinting can compare policies under runtime conditions without changing actual policies. System managers can explore a large space and settle on policies that yield low response time. Further, model-driven sprinting can explore what-if questions for past and future workloads. For example, what would response time have been if sprinting budget doubled during last week's spike? Or, how much

EuroSys '18, April 23–26, 2018, Porto, Portugal
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5584-1/18/04...$15.00
https://doi.org/10.1145/3190508.3190543

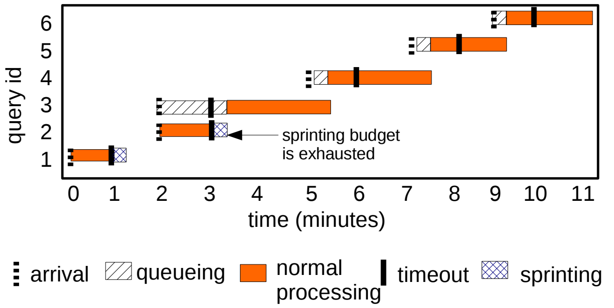

Fig. 1: Query executions under a tight sprinting budget. The first two queries drain the budget. Remaining queries can not sprint despite slow response time.

can be saved by purchasing hardware with the latest sprinting mechanisms?

Computational sprinting makes processing rate and queuing delays interdependent; Long queues trigger sprinting and sprinting reduces queues. Machine learning models can characterize interdependence. A direct approach maps policy and workload inputs to expected response time. While conceptually sufficient, this approach learns complicated semantics of sprinting. As a result, the model is slow to train. We propose a hybrid model that marries machine learning with models based on first principles. Our approach maps inputs to effective sprint rate. Effective sprint rate is the amortized speedup observed at runtime across sprinted executions (i.e., interdependence per sprint). Compared to response time, effective sprint rate is less sensitive to subtle policy changes, making machine learning more efficient. Finally, our hybrid model simulates query arrivals and departures using effective sprint rate. We study whether our hybrid model can produce accurate predictions that can be trained quickly enough for cloud workloads.

We compared modeling approaches using cloud server benchmarks. Our proposed hybrid approach used a random forest to get effective sprint rate and a first-principles queuing simulator to get response time. We compared our approach to an artificial neural network (ANN) that directly mapped inputs to response time. We studied prediction error across a range of (1) sprinting policies, (2) sprinting mechanisms (DVFS, core scaling, and CPU throttling), (3) query semantics (numerical computation, scientific kernels, memory-bound streaming, machine learning, search and mixed workloads) and (4) arrival and service rate distributions. Our hybrid approach achieved median error below 4.5% in most tests. On Spark workloads, its median error was only 3.2%. In contrast, direct-mapping approaches yielded 30% error in most tests and 5% error on Spark workloads. Of course, direct-mapping approaches improved when the training set grew. However, these approaches required 6X–54X larger training set to achieve accuracy comparable to the hybrid approach. We also compared our first-principles simulator without a machine learning model. Median error was 40%.

Our performance models make 900 predictions per minute (throughput scales with processor cores). This enables model-driven sprinting to compare thousands of timeout and budget policies for cloud servers. We used simulated annealing to explore the space. The best policies outperformed the worst policies by 1.65X. Our model-driven policies outperformed policies proposed in Adrenaline [19] and Few-to-Many [18].

Model-driven sprinting also supports service level objectives (SLOs) with response time clauses. In this case, model-driven sprinting can uncover sprint rates and budgets that (1) allow multiple workloads to share a cloud server and (2) respect SLO for hosted workloads. This use-case is inspired by AWS Burstable Instances which use CPU throttling to constrain sustained processing speed, set a fixed 5X sprint rate and budget 720 sprint-seconds per hour [4]. We studied homogeneous and heterogeneous workloads. Excluding model training, model-driven sprinting reduces tail latency by 3.16X and improves profit by up to 1.7X. However, our machine learning models require hundreds of examples for training. Further, the typical virtualized cloud server has a lifetime of 552 hours [9]. When we account for opportunity cost while collecting training data, net profit from model-driven sprinting is 1.6X greater than default AWS settings.

The remainder of this paper is as follows. Section 2 outlines our design for model-driven computational sprinting. Section 3 evaluates our modeling approach across cloud benchmarks and sprinting hardware. Section 4 studies speedup and cost savings from improved sprinting policies. Section 5 frames open challenges to widely deploy and extend model-driven sprinting. Section 6 overviews related work in the area of computational sprinting and modeling for highly dynamic systems. Section 7 draws conclusions.

2 DESIGN

Figure 2 depicts software, inputs and outputs in our modeling approach. A representative workload includes binaries and query mix. Our profiling software includes a query generator on the front-end and a back-end queue manager. The front-end replays the query mix and the queue manager dispatches queries to server system binaries. Our profiler runs on a server that supports the sprinting mechanism considered. We replay the mix many times, changing query arrival patterns and sprinting policies. The profiler captures response time, service time and queuing delay for each query execution. The profiler outputs the following:

1. **Service rate:** This is the inverse of mean processing time for query executions that do not trigger sprinting. In classic queuing literature, this is μ.
2. **Marginal sprint rate:** This reflects mean processing time when timeouts trigger before the queue manager dispatches queries, i.e., the whole execution is sprinted. We represent this using μ_m.
3. **Observed response times:** Each time the workload is replayed, we observe response times under the tested conditions and policies.

After the profiler finishes, characterizations of the workload and sprinting policy are passed into a random decision forest classifier. The classifier outputs effective sprint rate (μ_e), i.e., amortized speedup of sprinted queries caused by runtime dynamics. This metric captures the effects of having interdependent processing time and queuing delay.

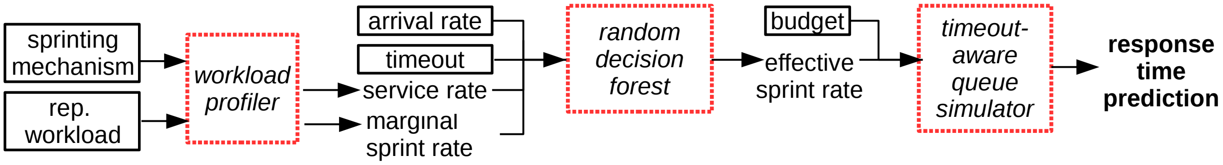

Fig. 2: Our approach combines workload profiling, queue simulation and machine learning (shown in red dotted squares). Black squares show inputs provided by users.

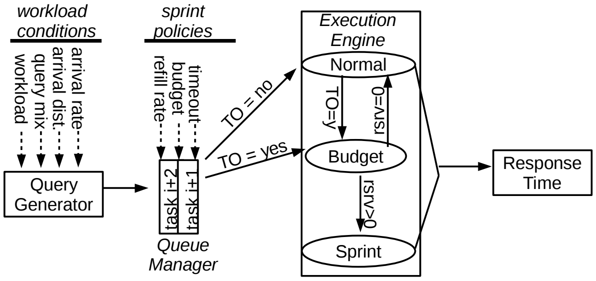

Fig. 3: Key instrumentation points for workload profiling in our approach.

To predict response time, we input sprinting policies, workload characterizations and effective sprint rate into a discrete event queue simulator. The simulator steps through system execution, detecting timeouts and modeling the impact of effective sprint rate. Our simulator can consider a wide range of queuing parameters including exponential, Pareto, and deterministic distributions of arrival, service, and sprint rates. It also executes quickly parallelizing execution across multiple cores and servers easily. This section details each stage in our design.

2.1 Workload Profiling

As shown in Figure 3, our query generator controls (1) the rate at which queries are sent to the queue manager, (2) timeout settings that trigger sprinting, (3) the budget for sprinting (in seconds), and (4) the rate at which the budget is refilled. To be clear, our workload generator is constrained by hard budgets and refill rates. In this paper, we explore soft budgets that provide flexibility. The generator can also switch between workload binaries and different mixes of query types.

The queue manager receives query requests (HTTP) from the query generator. The queue manager timestamps queries upon arrival, adding them to a FIFO queue. Queries wait in the queue manager until queries that arrived earlier complete. The queue manager forwards queries at the head of the queue to the execution engine when a slot opens. The queue manager also timestamps queries when they are sent to the execution engine. Timestamps allow us to compute processing time, queuing delay and response time.

The queue manager detects timeouts and triggers sprinting. For each query, the manager adds the timeout to arrival time and schedules an interrupt with a callback function. If the function executes before the query is dispatched, the queue manager initiates sprinting when the query reaches the head of the queue and a slot opens. If the callback executes after the query is dispatched, the queue manager initiates sprinting right away— provided the sprinting budget is not empty. After each query completes, the queue manager subtracts any time spent sprinting from the budget. In our current implementation, communication between generator, manager, and execution engine is through HTTP.

Our profiling setup covers a wide range of conditions. For example, our profiler can set arrival rates between 0.1%–100% of service rate at step sizes of 0.1%, covering over 1,000 arrival rates. This allows us to test small changes in system utilization. The parameters of sprinting timeouts and budgets also cover a wide range of settings. We use cluster sampling to obtain good coverage for a smaller range of conditions. Specifically, for each workload, we sample 5 settings for arrival rate, 8 timeout settings, and 9 power budgets respectively.

Cluster sampling ensures that random sampling covers key natural clusters within our supported conditions. However, cluster sampling can yield biased and high sampling error. In Section 3, we evaluate our ability to linearly interpolate between clusters to predict response time.

2.2 Timeout-Aware Simulator

Algorithm 1 outlines data structures and pseudo code for the simulator. Here, Vector means a expandable array of query objects. Each query object has the following properties: (1) arrival time, (2) processing time, (3) departure time, (4) time at which processing started, and (5) booleans that signal if queries were sprinted.

Given arrival patterns, we set the time when each query to be processed will arrive. We randomly sample service time data collected during profiling to set $\bar{\mu}$. These properties are set before simulation begins.

Our simulator steps through server execution. We normally set step size to one millionth of a second. To focus on handling timeouts, Algorithm 1 shows a simple setup that does not allow concurrent query executions. Our full simulator is open source and supports concurrent executions [33]. The queue vector holds queries waiting to be processed. When queries arrive, they are appended to the queue vector. If the execution engine has a slot, query processing begins immediately. If not, the query waits until it reaches the head of the queue.

As shown in Equation 1, we model sprinting, i.e., the query *depart* time, as a linear speedup on the query's remaining execution time, using the quotient of service and effective sprint rate as the coefficient. We denote τ as a fraction of completed work, which is defined by the difference of current *clock* time and its *start* time divided by the average processing time. To be clear, all variables in this

```
// Key data structures
GLOBAL Vector queries   // queries to be processed
GLOBAL Vector queue     // capture queueing effects
GLOBAL Vector clock = 0 // fine resolution clock
GLOBAL int    slots = 1 // slots in execution engine

void function qs ( μ, μ_e, timeout, budget ) {
 while (queries.empty() == false) {

   // Add new arrivals to the queue
   arriving_query = queries.elementAt(0)
   if (clock == next_query.arrival) {
     queue.append(arriving_query)
     queries.removeElement(0)
   }

  // Dispatch from queue to execution engine
  if (slots == 1) {
     queue.elementAt(0).start = clock
     queue.elementAt(0).depart = start + μ̄
     slots = 0
  }

  // Check for timeouts
  head_query = queue.elementAt(0)
  if (clock == head_query.arrival + timeout) {
     head_query.TimedOut = true
     if (budget > 0)
       head_query.depart = clock + f(start, μ, μ_e)
  }
  else if (clock == head_query.depart) {
  // Check for query completion
     queue.removeElement(0)
     slots = 1
  }
  clock++
 }
}
```

Alg. 1: G/G/1 timeout-aware queuing simulator.

equation align with Algorithm 1 and clock is captured immediately after timeout.

$$depart = clock + 1 - \tau \cdot \bar{\mu} \cdot \frac{\mu_e}{\mu}, where \tag{1}$$

$$\tau = clock - start\bar{\mu}$$

2.3 Effective Sprint Rate

Workload profiling provides observed response time under tested workload conditions. Furthermore, our queue simulator eschews runtime factors in its model of computational sprinting, see Equation 1. We use workload profiling and queue simulation together to model effective sprint rate. Specifically, our machine classifier targets unaccounted runtime factors, such as: (1) the points in query execution where sprinting begins, (2) queuing delay caused by toggling the sprinting mechanism and (3) queue length when sprinting begins. Our classifier maps workload conditions and sprinting policies to a linear regression that quantifies unaccounted factors. To be precise, we define effective sprint rate as the nearest sprint rate that aligns simulator and observed response time. Equation 2 formalizes our model. RT_{wp} is the response time function for workload profiling. The input is tested workload conditions F and marginal sprint rate μ_m. RT_{qs} is the response time function for the queue simulator. The effective sprint rate makes the smallest absolute change to marginal sprint rate while achieving tolerable error on response time.

$$\mu_e = \mu_m + \min|x|, where \tag{2}$$

$$|x| \in \{RT_{wp}F, \mu_m = t * RT_{qs}F, \mu_m + x, \exists t < T\}$$

We find the expected sprint rate through exhaustive search. We increment and decrement the marginal rate by 1 unit to get candidates for effective sprint rate. We use these candidates as input to the queue simulator and compare observed and simulator response time. If the difference exceeds our tolerance threshold, we repeat.

2.4 Random Decision Forest

Random Decision Forest (RDF) is a combination of decision tree predictors [5]. Each decision tree depends on the values of a random feature vector sampled independently. We use RDFs to infer effective sprint rate. First, we randomly divide profiling runs into training and testing data. We then create random subsamples from the training data. For each subsample, we select a random subset of workload conditions and sprinting policies to build decision trees. Figure 5 depicts the subsampling process. Columns represent workload conditions and sprinting policies (F) used as predictive features for the effective sprinting rate. Offline profiling produces each row, i.e., we observe response time and align simulator results to get effective sprint rate.

For each subsampled training set, we use the ID3 algorithm to build a deep decision tree [30]. A decision tree is an acyclic graph where internal nodes are predictive features, edges are feature settings, and leaf nodes provide regression results for training data that matches feature settings specified in the path from the root. We create binary trees. At each node, we compute variance VS for data that matches feature settings in the path from root (all data for the root node). Then for each feature, we compute variance for data in (1) a proper subsets of the feature settings ($VS_{F_i=k}$ and (2) the complement ($VS_{F_i \neq k}$). As shown in Equation 3, the subset and complement that most reduce variance provide edges to the next node. When all feature settings are exhausted, we create a leaf node by using linear regression on the remaining samples.

$$\max_{F} gain_i = VS - \frac{VS_{F_i=k} + VS_{F_i \neq k}}{2} \tag{3}$$

As shown in Figure 5, each subsample from the training set produces a decision tree. We average the regression parameters from each tree to derive final prediction patterns.

Why Random Decision Forests? Cluster sampling systematically explores a subset of policies but also introduces bias, i.e., it misses policy settings that differ from cluster centroids. Bias causes modeling error for unseen conditions. Random decision forests minimize bias caused by cluster sampling without increasing profiling costs. Without additional data, the key is to understand which data points (i.e., tested runtime conditions) are most similar to unseen conditions. Our key observation is that sprinting policies (1) exhibit low-variance in effective sprinting rate under specific conditions and policy settings and (2) subtle changes on most settings have only small effects on sprint rate. A key insight is that *these observations can be applied to effective sprinting rate, but not necessarily to response time*. Random decision forests minimize bias by creating deep decision trees. We eschew pruning approaches, because shorter

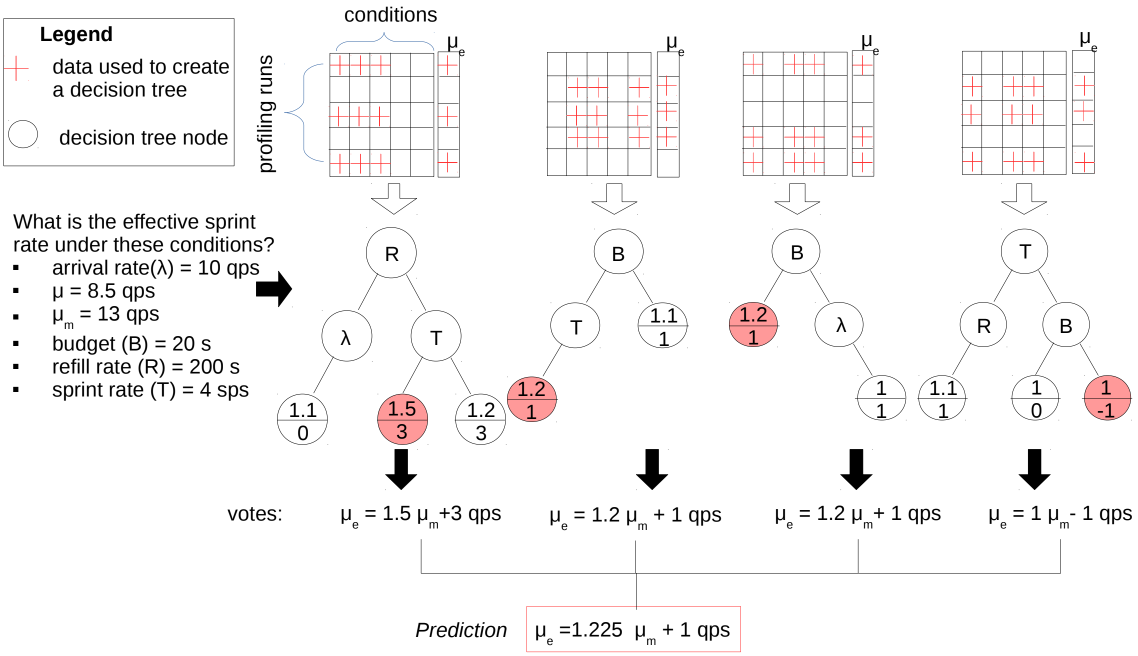

Fig. 5: Creating and using random decision trees to classify effective sprint rate.

trees ignore the complex effects of some workload conditions sprinting policy parameters. However, by creating multiple trees that each use different predictive settings, we can group data points that are likely related on key parameters.

3 PERFORMANCE MODELING RESULTS

In this section, we evaluate our performance modeling approach. Unless otherwise stated, our experiments use the following procedure. For each workload and platform tested, we observe response time for all workload conditions and sprinting policies at cluster sampling centroids. We randomly select a subsample to train our model. The remaining 20% of tested conditions, as well as conditions outside of cluster sampling centroids, are used to compare observed to predicted response time.

We compare performance modeling approaches shown in Table 1(A). **ANN** directly predicts response time from input policies and workload conditions. Recall, response time is sensitive to subtle policy changes. This approach requires machine learning that can characterize discontinuous functions. We used an artificial neural network (ANN), a powerful approach that uses error residuals to find non-linear splits in discontinuous functions. In comparison, bias in the ID3 algorithm limits the effectiveness of random forests on discontinuous functions. **No-ML** uses our timeout-aware simulator with marginal sprint rate. It eschews machine learning. **Hybrid** implements our approach.

Table 1(B) describes sprinting hardware in our tests. **DVFS** and **CoreScale** use a Xeon 2660 processor. **DVFS** uses Pupil [44], state-of-the-art power capping software. Pupil maximizes throughput under a power cap by learning the relationship between DVFS setting and power usage. We sprint by temporarily increasing the power budget, allowing Pupil to adjust the processor to the best DVFS setting for the workload. **CoreScale** increases the number of active cores used during query execution from 8 to 16, using the Linux taskset utility [15, 25]. **EC2DVFS** used an EC2 Extra Large C-class instance (circa 2017). Here, we sprint by changing P-States, i.e., we set DVFS directly.

Table 1(C) compares query execution semantics (workloads) on DVFS hardware. We set up 2 Spark cloud services, running data streaming and K-means benchmarks. These workloads are compute intensive. Their performance scales with Pupil power cap. We also set up 5 HPC kernels that stress specific aspects of the processor architecture. HPC kernels run under MPI 2.1 with 16 software threads. Jacobi and KNN are both computational intensive workloads with good cache locality; sprinting improves throughput significantly (1.2X and 1.7X respectively). Memory bandwidth constrains BFS and Mem, making DVFS sprinting less effective (1.3X speedup). Finally, Leukocyte is limited by synchronization. Sprinting provides speed up of only 1.16X on this workload.

The following list specifies cluster sampling centroids.

Query Arrival Rate: 30%, 50%, 75%, 95%

Workload Mix: Uniform, Weighted

Arrival Distribution: Exponential, Pareto

Timeout: 50, 60, 70, 80, 120, 130, 160 (seconds)

Refill Time: 50, 200, 500, 800, 1000 (seconds)

Sprint Budget: 14%, 16%, 18%, 20%, 40%, 60%, 80%

Query arrivals are shown as percentages of service rate (i.e., system utilization in queuing literature). We have studied mixes of uniform

(A) Performance Modeling Approaches	
Approach ID	Description
ANN	Multi-layer (10 layers and 100 neurons) artificial network maps policies and workload conditions directly to response time
No-ML	timeout-aware queue simulation uses marginal sprint rate (no machine learning)
Hybrid	our hybrid approach → random forest (10 trees) + timeout-aware simulation

(B) Sprinting Hardware	
Mechanism ID	Processor Specs
DVFS	16 Cores, 62 GB RAM, 20 M Cache 1.2 –- 2.40 GHz processing speed Sustained power cap: 44--70 watts Burst power cap: 90--190 watts
CoreScale	16 Cores, 62 GB RAM, 20 M Cache 2.1 GHz (active cores) Sustained speed: 8 active cores Burst speed: 16 active cores
EC2DVFS	36 virtual CPU, 60 GB RAM Sustained speed: 1.4 Ghz Burst speed: 2.0 Ghz

(c) Cloud Server Workloads		
Wrkld ID	Description	Sustained/Burst Tput (on DVFS)
Spark Stream	continuously process data from source	87 qph / 224 qph
Spark Kmeans	cluster analysis in data mining	73 qph / 144 qph
Jacobi	solve Helmholtz equation	51 qph / 74 qph
KNN	k-nearest neighbors	40 qph / 71 qph
BFS	breadth-first-search	28 qph / 41 qph
Mem	stress memory bandwidth	28 qph / 37 qph
Leuk	track leukocytes in medical images	25 qph / 29qph

Tab. 1: Identifiers (IDs) for models, hardware and workload in our experiments.

query workloads with exponential and heavy-tail arrival distributions. The queue manager enforces a global timeout on each query execution. After *refill time* elapses without sprinting, the budget for computational sprinting reaches full capacity. We set the sprinting budget as the percentage of maximum query throughput during the refill time. To be clear, both sprinting budget and query arrivals depend on service rate— this is why we normalize.

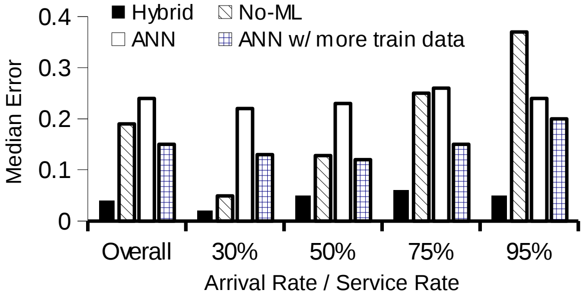

Fig. 7: Absolute relative error produced by competing performance modeling approaches.

3.1 Impact of Modeling Approach

Figure 7 shows error rate as system utilization increases. The hardware is DVFS. We report averages across all tested workloads. Hybrid and ANN approaches were trained for 7.2 hours (80% of sampling centroids). We also show results where ANN was trained for 8.6 hours.

Our hybrid approach achieves median error of 4%. It outperforms ANN and No-ML approaches by 4X-6X. ANN performs better with more training data. Median error drops to 15% after enlarging training data by 20%. We adjusted training data for Hybrid and ANN until they performed similarly. ANN required 6X–54X more training data to match our Hybrid approach depending on the workload.

Interdependent processing rate and queueing delay hurts No-ML under high system utilization. At low arrival rates, No-ML performs nearly as well as our hybrid approach, but under heavy arrivals, it performs worse than all others. No-ML uses only our timeout-aware simulator. We validated our simulator using classic MMK workloads, where it achieved median error of 5%.

3.2 Impact of Query Execution Semantics

Figures 8(A) and 8(B) evaluate Hybrid and ANN models for each workload. Both models were trained with 80% of cluster sampling centroids. These tests use DVFS.

Hybrid achieves lower median error than ANN for all workloads. Its median error is below 5% for Spark K-means, Spark Stream, Jacobi and Leuk. Median error is below 10% for all workloads.

Hybrid errors by more than 35% for 20% of Leuk and 17% of Jacobi tests. In contrast, ANN achieves median error below 10% for Jacobi and Leuk. These workloads have low variance in service time distribution which reduces the learning burden for ANN. Hybrid struggled to capture late timeouts for Leuk, a workload with strong execution phases. Late timeouts trigger while execution is in flight. Our random forest did not detect discontinuous shifts in response time where long timeouts trigger sprinting after sprinting-friendly phases passed. Nonetheless, the hybrid approach translates between marginal and effective sprint rate well. Its predictions align with observed response time across a wide range of workloads and policies.

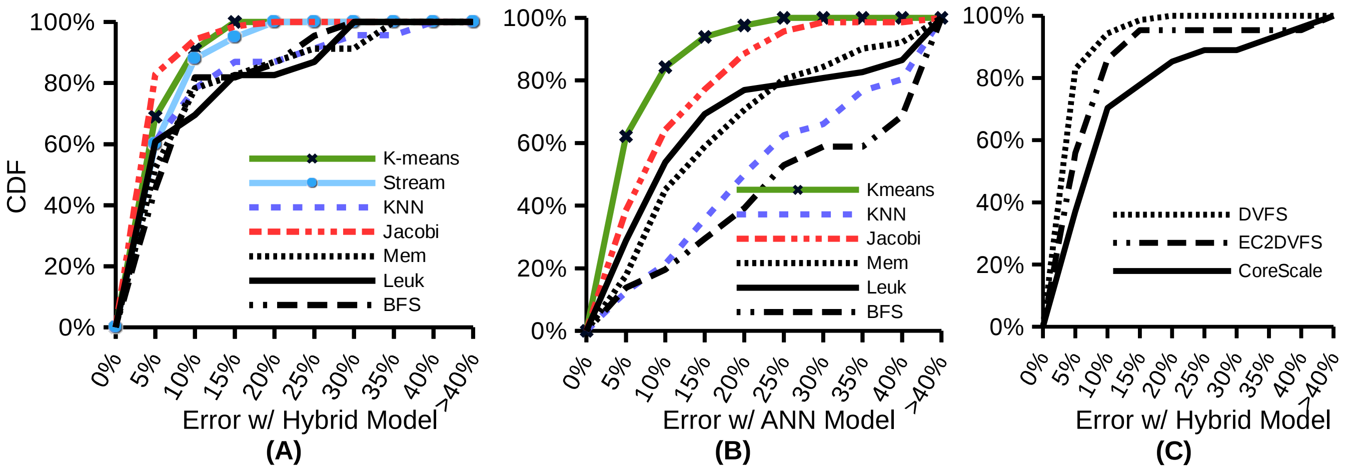

Fig. 8: CDFs of prediction error across workloads with hybrid approach.

3.3 Impact of Sprinting Hardware

Figure 8(C) plots error for Hybrid across sprinting hardware. This plot shows only Jacobi. Median error with DVFS and EC2DVFS was below 4%. On these platforms, our approach yielded error below 10% on over 80% of the tested sprinting policies. With core scaling as the sprinting mechanism, median error was 8%, and over 60% of tested policies yielded error below 10%.

Core scaling is limited by Amdahl's Law; as execution progresses there are fewer active software threads and the potential speedup from increased parallelism diminishes. In Jacobi, marginal sprint rate with core scaling is 1.87X faster than service rate, i.e., with sustainable processing mode, the execution time was 202 seconds but, if the whole kernel was sprinted, the execution time was 108 seconds. However, if only the last 22 seconds are sprinted, the speedup drops to 1.5X. Bias caused by cluster sampling and ID3 algorithm makes it hard to model such phase behavior. Adding data can reduce bias. In particular, the following techniques dropped median error below 5%:

· Cluster sampling at 60% and 85% query arrival rates,

· Using 90%–10% training-to-test data split.

3.4 Impact of Query Mix

We also studied our approach under a mix of workloads. In queuing theory, a query mix alters the probability distribution governing processing time. Prior studies have shown that query mix is best modeled with M/**G**/K models, where G stands for general processing time distribution. We tested two query mixes. In the first mix, 50% of query executions ran Jacobi and 50% ran Stream. The second mix evenly split query executions between Jacobi, Stream, NN, and BFS. We also changed arrival distribution to Pareto ($\alpha = 0.5$). In classic queuing models, this setup is called G/G/K. There is not a closed-form analytic model for this setup. However, our approach, which uses simulation, can predict expected response time even with computational sprinting enabled.

We ran tests on DVFS. Our workload profiler measured sustained service rates of 35 and 30 for query Mix I and II, respectively. Note, sustained service rate for each mix fell below the average of the kernels in isolation due to interference between the workloads. Figure 9 plots the CDF of error for Mix I and Mix II. The median error was 7% for Mix I and 10% for Mix II. 75% of the predictions for Mix I achieved error below 15%. For Mix II, 60% achieved error below 15%.

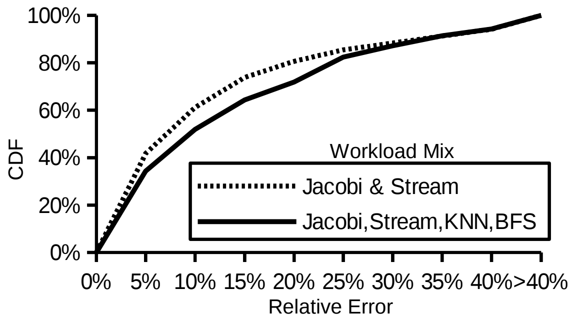

Fig. 9: The cumulative distributions of prediction error for two distinct mixed workloads.

3.5 Impact of Other Design Factors

In our approach, service rate, arrival rate, timeout setting, and sprint budget are first-class parameters. We studied the impact of these parameters on prediction accuracy. Here, we used experiments from all platforms and workloads and grouped them according to their setting on these parameters. We used binary groups (hi & low). For service rate, we split at 40 qph. For arrival rate (utilization), we split at 60%. For timeout setting, we split at 100 seconds. And for sprint budget, we split at 40%.

Figure 10 shows average prediction error for each grouping. 75th and 25th percentiles are shown as bars. The largest error across all groups was 4%. Our approach predicts response time well regardless of throughput of target workload, system utilization, and sprinting policy setting.

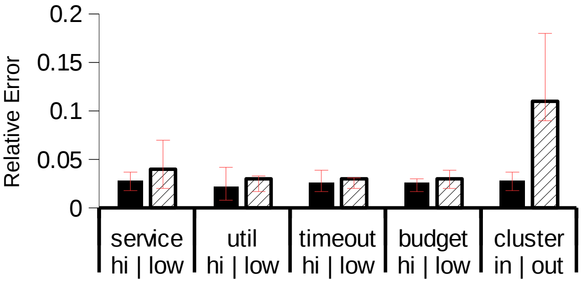

Fig. 10: Impact of service rate, arrival rate, timeout, budget and cluster sampling.

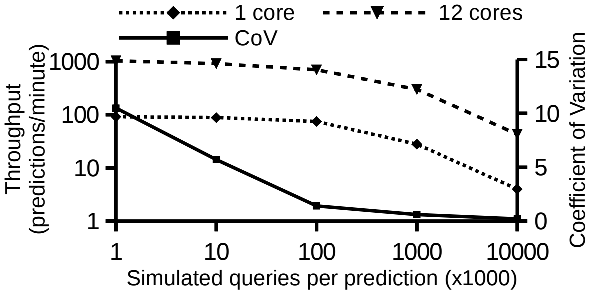

Fig. 11: Throughput and variance for response time predictions using our timeout-aware simulator.

Cluster sampling reduces profiling time, but also makes training data less representative (i.e., more biased). We studied the accuracy of our model for response time prediction under conditions *not* in cluster sampling centroids. We studied linear interpolation in arrival rate and timeouts by removing cluster centroids from training data. Specifically, we remove arrival rate of 75% and timeouts of 60, 70 and 120 seconds. We used observations under these settings as test conditions and evaluated the accuracy of our predictions. To traverse the decision tree, we snap parameter settings to the nearest centroid. Figure 10 shows median error when these centroids are *in* the training data and *out* of the training data. As expected, our predictions yield 2.5X greater error when test conditions are not cluster sampling centroids. Median error was 10% and workload variance was larger than other design factors. Still, 10% error is sufficient to help system managers choose between competing sprinting policies.

3.6 Predictions Per Minute (Overhead)

Figure 11 plots the number of predictions our approach can produce per minute. These tests were run on an Intel Xeon 12-core processor at 2.6 GHz. Figure 11 shows that our approach achieves speedup of 11.4X when scaled from 1 to 12 cores. Throughput also depends on the number of queries simulated. Fewer queries increase throughput, but lead to less accurate predictions. We observed a knee-point in the variance of our predictions at 100K queries simulated. At that point, we can compute roughly 100 predictions per minute.

4 MODEL-DRIVEN COMPUTATIONAL SPRINTING

Model-driven sprinting helps cloud workloads achieve low response time *and* cloud providers share hardware among workloads. This section studies the impact of model-driven sprinting in both contexts. First, we describe computational sprinting for this context.

4.1 Computational Sprinting

Computational sprinting uses a resource budget to speedup a workload. Traditional sprinting overclocks hardware for short burst followed by a cooldown period. The additional heat generated from overclocking restricts the duration. A resource budget assigned on a CPU-cycle granularity allows policies to precisely control sprinting. This finer control prevents thermal runaway when overclocking hardware. A coarser granularity is sufficient when speedups are gained through allocating more hardware or increasing utilization. For example, cloud providers throttle-down the CPU to conserve power during low traffic periods. High traffic periods trigger the CPU to throttle-up. This dynamic scaling maintains the quality of service regardless of load variation. The use-case in the following section defines the budget in seconds and uses CPU throttling for sprinting. CPU throttling operates within normal thermal limits. Therefore, defining the budget in CPU-cycles is unnecessary.

4.2 Timeout Policy Exploration

Our exploration algorithm iteratively selects policies from a multi-dimensional space. It computes the response time of each policy and finds the one with the lowest result. This technique is probabilistic. It randomly makes large leaps to other policies of the search space. This helps it avoid stopping at local minima or maxima.

Our objective is to find a timeout policy which leads to the minimum response time. That is, we iteratively adjust the timeout setting to our model-driven approach for a maximum number of iterations. Then, we choose the setting associated with the lowest expected response time. Equation 4 formalizes the problem.

$$MINRT : \exists t \geq 0, RT = P_M S_{F \neq t}, t \tag{4}$$

RT is expected response time produced by our model-driven approach P_M. $S_{F \neq t}$ is a subset of workload conditions and sprinting policies— excluding timeout. This algorithm finds timeout t as follows:

(1) Generate a random timeout t_o and predict RT_o.
(2) Generate a neighboring timeout t_n and predict RT_n.
(3) If $RT_n < RT_o$ move to the new solution; Else, move to the new solution based on the acceptance probability.
(4) Repeat steps 2-3 until maximum number of iterations is reached.

Neighbor timeouts t_n are drawn randomly from a narrow range of timeouts. Specifically, we use $t_o - 100, t_o + 100$. Step 3 compares RT_o and RT_n and makes a decision to accept the tuple $< RT_o, t_o >$ or $< RT_n, t_n >$. This algorithm avoids local minimums by using an acceptance probability a when RT_n performs worse than RT_o. Acceptance probability is defined in Equation 5.

$$a = \begin{cases} 1 & \text{if } RT_o - RT_n > 0 \\ e^{\frac{RT_o - RT_n}{Z}} & \text{otherwise} \end{cases} \tag{5}$$

Z starts at 1 and decreases by 10% per 100 timeout settings explored. This reduces the probability of searching new gradients as the algorithm progresses.

4.3 Model-Driven Sprinting for Cloud Workloads

Model-driven sprinting helps workloads decide (1) *when* to sprint by finding good timeout policies and (2) *how* much budget to request. We answered these questions for Jacobi. The sprinting mechanism studied was CPU throttling. In CPU throttling, resource managers enforce a sustained processing rate by limiting access to CPU. During a sprint, managers remove limitations until a workload exhausts its budget. Jacobi's throughput was throttled to 20% of its sprint throughput on DVFS. Sustained processing rate was 14.8 queries per hour (qph). Sprint rate was 74 qph. Budget allowed 5 query executions to sprint fully. Queries arrived at 11.8 qph (80% utilization).

Our service level objective (SLO) was to throttle CPU but keep response time nearly the same. To be precise, our SLO allows response time to increase by 15% relative to throttling turned off. We compared these approaches:

+ **big-burst**: Timeout is 0. Each arriving query sprints until budget is drained.

+ **small-burst**: Timeout is 0. Each arrival sprints but with lower sprint rate (44 qph) and larger budget to 10 queries.

+ **few-to-many**: Adapts Few-to-Many to our context [18]. Profiles marginal sprint rate for query executions offline. Then, finds the largest timeout setting that exhausts budget (speeding up the slowest queries). Throughput improved 1.9X.

+ **adrenaline**: Adapts a key policy in Adrenaline to our context [19]. Sets timeout to the $85^{th}\%$ percentile of non-sprinting response time.

Results: Figure 12(A) shows the response time across a range of timeout settings. Poor performing settings exceed SLO response time by 1.4X. In contrast, timeouts set well meet SLO and approach no-throttling performance.

When sprint rate improved throughput by 5X (big-burst), model-driven sprinting found settings that improved response time by 1.44X compared to Adrenaline and by 1.3X for Few-to-Many. Our approach explores all timeout policies, including policies that target short executions. For Jacobi with fast sprinting, fast timeouts kept queue length small which improved response time. In contrast, under 3X sprint rate (small-burst), Few-to-Many matched our approach. With larger sprint budget, Few-to-Many's approach sprinted for enough fast queries to perform well.

Figure 12(B) compares timeout settings on Mix I (Jacobi and Mem). Small-burst never meets SLO, because this CPU throttling offers low speedup for Mem. Few-to-Many finds good timeout settings for both small-burst and big-burst setups. Model-driven policies still outperform Adrenaline by 1.17X and 1.24X in small-burst and big-burst respectively.

For the tests in Figure 12(C), we fixed timeout setting and explored the impact of sprinting budget. The best timeout setting depended on sprinting budget. Under tight budgets, loose timeouts that target very slow queries led to lowest response time. Under loose budgets, strict timeouts that aggressively sprint led to lowest response time. This finding parallels a key intuition in Few-to-Many [18]: Under low utilization, all query executions should sprint aggressively but, under heavy utilization, resource managers must sprint for the most needy queries.

4.4 Model-Driven Sprinting for Cloud Providers

Amazon EC2 T-class Burstable Instances allow multiple workloads to share a server with CPU throttling. Burstable instances can compute at a sustained rate and sprint at a faster rate. On EC2, burstable instances of the same class have the same sprinting policy, regardless of workload. For example, T2.small Instances use 20% of a single core, sprint 5X faster and have a budget of 720 sprint-seconds per hour [4]. In this case, the budget specifies how long a workload has access to 100% of the CPU before returning to baseline performance. Amazon sells T2.small Instances at $0.026 per hour per workload. However, the number of workloads that can share a server is limited by SLO. Workloads that incur SLO violations will not use T2.small Instances.

In this section, we use our model-driven approach to colocate workloads on burstable instances. We compute expected response time under a policy's sustained processing rate, sprint rate and budget. If the policy meets SLO (i.e., 1.15X of response time under no throttling), then workload is permitted to colocate. We add workloads until we have committed 100% of CPU resources (i.e., the sum of sustained rate and sprinting). Colocation is not allowed to over subscribe.

Figure 13 compares revenue per node of the following approaches to set sprinting policy.

+ **AWS:** Sets a fixed sprint rate and budet for each workload. To be precise, each workload receives 20% of a single core and sprints 5X faster for 12 minutes per hour.

+ **Model-Driven Budgeting:** Enlarges sprint rate by shrinking budget. Searches for combination that meets SLO.

+ **Model-Driven Sprinting:** After setting budget, this approach also explores timeout settings. Workloads allow cloud providers to change their timeouts.

Results: Figure 13 shows revenue per node across competing sprinting policies, i.e., $\$0.03 \times n$ where n is number of colocated workloads. We studied three workload combinations. The first workload combo has 4 copies of a Jacobi service running at 70% utilization. Our model-driven approach finds good sprinting policies for this workload. The budget approach can host 2 workloads under SLO. Budget+timeout can host 3 workloads. AWS policy hosts 1 workload per server, essentially making the server a dedicated host. The second combo hosts 2 Jacobi (70% util) and 2 Stream (80% util). Again, adjusting budget and timeout allows workloads to meet SLO without overbooking. The third combo hosts diverse workloads with utilization ranging from 50% to 80%. We find unique budgets and timeouts for each. In this case, we can host all workloads under SLO.

We also examined 99^{th} percentile tail latency for Jacobi (i.e., response time >335 seconds). Compared to our model-driven policy, the AWS policy produced 3.16X more query executions in the tail. Model-driven policy reduced 99.9^{th} percentile (i.e., >521 sec.) by

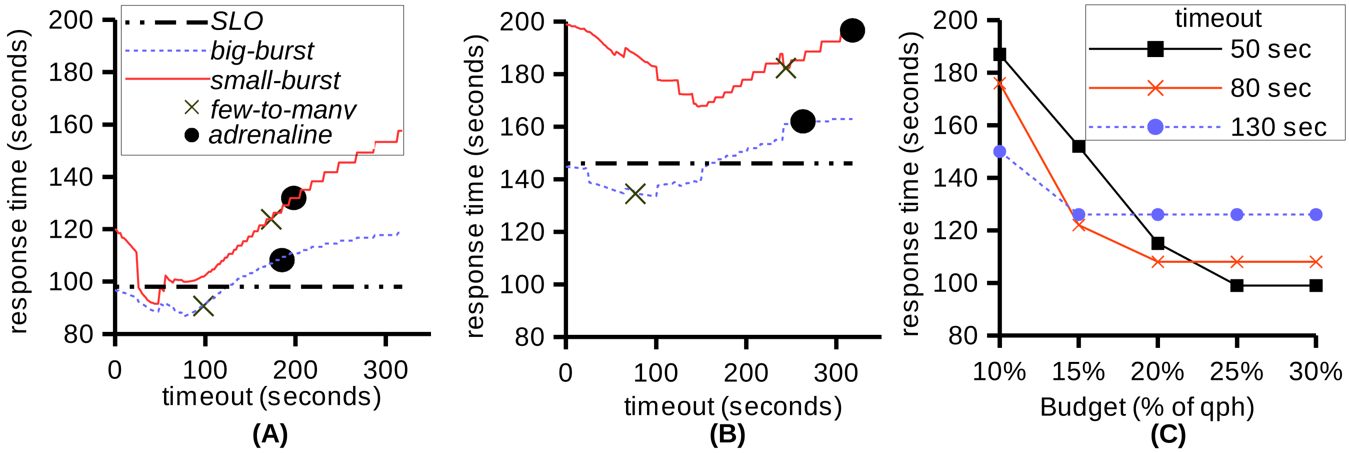

Fig. 12: Our model-driven approach was used for space exploration on DVFS. The vertical axes are the expected response times for a workload. Exploring timeout policies with (A) Jacobi kernel and (B) Mix I (Jacobi & Stream). (C) Response time as sprinting budget and timeout vary. We report budget as percent of sustained processing rate.

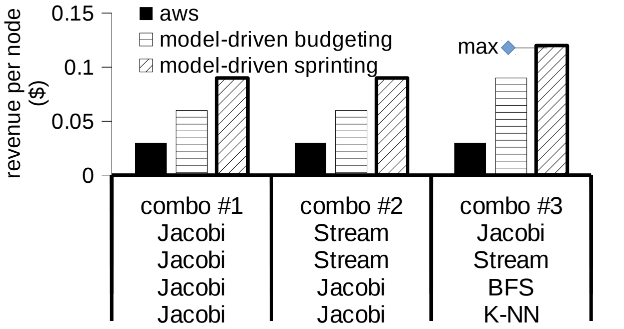

Fig. 13: Dollars earned for a burstable instances which optimizes the sprinting policy based on budget and timeout.

3.76X. By its nature, sprinting shrinks the tail [19]. Model-driven policies amplify gains.

Does model-driven sprinting save cloud providers more than it costs? Our model-driven approaches require offline workload profiling. The previous section shows that our sprinting policies can improve revenue per server *but* does not consider revenue lost during profiling. Figure 14 explores the following approach. When a user starts a burstable instance, the server owner runs it on a dedicated node *and* starts workload profiling on another node. During profiling, the server owner (Amazon) does not profit. After profiling, however, the server owner benefits from increased revenue per server. Our approach needs roughly 7.2 hours per workload (e.g., on the DVFS platform) for profiling. For the four workloads in Combo III, total profiling time would be 28.8 hours. After 2.5 days, model-driven sprinting with our hybrid model is cost effective. Recall, the ANN model trains longer, but this approach is eventually cost effective too. Over the lifetime of a virtualized server [9]. Model-driven sprinting with our hybrid model increases revenue by 1.6X.

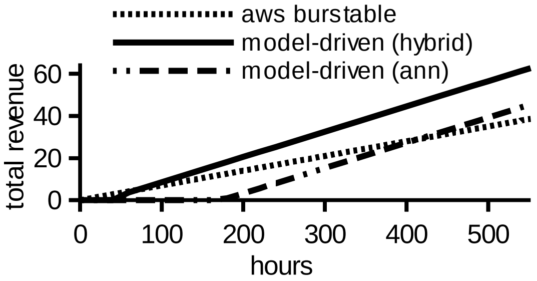

Fig. 14: The number of hours required to offset the profiling cost using our model.

5 DISCUSSION

Model-driven sprinting requires that sprints target specific query executions. This prohibits sprinting mechanisms that affect all concurrent queries. Fortunately, most container platforms already track and manage resources usage of query executions. Early research prototypes aimed to provide such first-class support to query executions [2, 3]. Intellectual descendants of these systems include Omega [42], Apache Hadoop, Apache Spark, and C-groups Docker.

Cloud services can also use model-driven sprinting by managing sprint resources directly. With careful thread management, OS tools can be used to isolate sprinting to query executions. For example, *taskset* can pin a query's threads to specific cores, enabling core scaling. Similarly, P-states can be used to control per-core DVFS for each query execution. Such explicit application control can actually enable improved performance [7].

Model-driven sprinting provides greater speedup and cost savings by setting timeout policies. However, applications normally implement, set and manage timeouts without involving platforms. Few platforms manage timeout policies directly. Recent research platforms have shown that it is possible to provide rich timeout support [16, 22, 23]. AWS Lambda is a commercial platform that supports application timeouts. Its adoption promises exciting use cases for model-driven sprinting.

In this work, we evaluated and tested our performance models under known runtime conditions (e.g., arrival rate). A key open challenge is to estimate runtime conditions online and apply our model on noisy predictions. Sliding window approaches can be used to estimate runtime conditions. Building upon these approaches in the context of sprinting is critical. A related challenge is updating machine-learned models when runtime conditions shift. This can be especially challenging when there are multiple sprinting mechanisms available.

The generality of model-driven sprinting depends on the data used for training and the simulator's parameters. Data representative of the policy-space enables accurate predictions for unseen sprinting policies. However, this approach cannot extrapolate its predictions for more sprint rates than allowed by the simulator. This is also true for different timeouts assigned across workloads. Only small modifications to the simulator are needed to support multiple sprint rates and timeouts.

Finally, cloud applications are widespread and have significant economic impact. Results like our 3.16X improvement to tail response time could save a large services millions annually. But model-driven sprinting has applications beyond cloud servers. Mobile software, IoT/edge systems and even client-side browsers increasingly struggle under tight (often battery limited) energy budgets. Sprinting mechanisms under these contexts will also benefit from model-driven space exploration.

6 RELATED WORK

Systems limited by *dark silicon* [10, 40] cannot sustain peak processing rates. Sprinting mechanisms enable peak processing in bursts and have been implemented across the whole system stack from transistors [12] to processors [36] to racks [32] to data centers [19, 45] and cooling systems [17]. Sprinting policies [11, 31] address the management challenge: Can short bursts in processing rate boost response time for the whole system? If so, how large of a budget is needed? And which runtime factors matter? Our contribution is a model-driven approach to explore these problems for cloud systems. The remainder of this section divides prior work by the workload types, i.e., single v.s., multiple jobs, and model-driven approaches.

6.1 Sprinting policies for a single job

The dominant heuristic for computational sprinting within a single long-running job is to use the sprinting budget on different workload phases so as to minimize the execution time. Profiling the workload phases is the very first step to develop phase-specific sprinting policies. PUPIL [44] runs a wide range of offline experiments to build a model of phases' execution to performance improvement. Coz [8] uses static program analysis to identify phases and reduce training time to characterize per-phase benefits. Bailey et. al [1, 18] also consider parallelism between phases and interactions when their executions overlap. While the aforementioned studies achieve significant speedups for a given job compared to phase-agnostic policies, they neglect the effects on a stream of arriving jobs, especially on queuing delay.

The sprinting game [11] expands beyond a greedy policy by exploring the impact of a sprint on the executing code and the future possibilities to sprint. A key observation is that greedy approaches largely under-explore the capabilities of sprinting mechanisms, highlighting the need to better consider the workload patterns and configure the sprinting policies. Indeed, the sprinting game and our model-driven approaches explore the whole-space of policies. The sprinting game presumes cost models per sprint. Our approach enables such models based on response time.

6.2 Sprinting policies for server systems

When queries arrive independently, sprinting policies decide which jobs to accelerate to meet service level objectives. Prior works have explored which jobs to accelerate. Jeon et. al [20, 21] use a model-driven approach to detect which web search queries will have long response times. Queries expected to have slow response time are allowed to execute on more cores, i.e., core scaling. We extend this work with robust models that can characterize response time (1) in advance for planning, (2) under unseen conditions and (3) for a range of sprinting mechanisms. The key intellectual difference is that our modeling techniques do not use runtime data on queue length. However, as noted in [42], our dependence on testing and varying runtime conditions can make our approach hard to scale for warehouse workloads. In future work, we hope to explore passive approaches to calibrate our model [34].

Verma et al. [41] compare techniques to evaluate workload packing in the presence of bursts that presumably cause SLO violations. They found packing techniques that explore resource lean environments, e.g., by reducing sustained and sprint rate or sprinting budget, avoid over valuing high utilization and reduce the number of resources stranded by not fitting into fixed hardware. Our model-driven approach provides first steps toward realizing such resource compaction techniques in the presence of computational sprinting. Currently, our approaches are likely too heavyweight for cluster-scale sprinting. We target computational sprinting on single machines. We also do not consider heterogeneous memory or processor cache speeds yet, but plan to do so in future work.

Wang et. al [43] use CPU throttling to sprint virtual machines from different tenants, factoring in their sensitiveness to the price performance ratio. They showed that using effective VM capacity modulation as a control knob in a leader-follower game can fulfill the performance requirement of tenants and increase the profit of cloud providers. Such CPU throttling is widely used by cloud providers. Section 4 uses our model-driven approach to explore sprinting policies in this scenario.

6.3 Model-driven approach

Queuing models can predict response time for server systems [24, 35]. However, these models assume queuing delay and service time are independent. Interdependent queuing and service times lead to complicated models [27]. Recent research have created models for specific conditions and sprinting mechanisms, e.g., query replication [14, 29], admission control and dynamic voltage scaling [6]. Determining the optimal service rate for servers [28] is a long standing difficult problem even for simple queuing systems because of the interdependency between service rate and waiting time. Markov decision processes offer an efficient mean to compute the optimal rules for systems whose arrival and service processes have Markovian properties. Chen et. al [6] model the dynamic voltage scaling problem of a single server as a discrete time Markov decision process by leveraging fluid approximation. Gardner et. al [14] develop models to answer how to sprint a certain set of jobs by replicating them and giving them more opportunities to access resources. While their focus is on the average latency, Qiu et. al [29] use matrix analytics methods to model the entire distribution of latency under different degrees of replication factors.

The immediate challenges to apply existing queuing models are twofold. First, how do we map abstract queuing models to complex systems? Second, how do we parameterize the model inputs? Motivated by the difficulty of using queuing models to predict actual systems performance, IRONModel [38] argues the effectiveness of combining first-order queuing models with statistical learning to capture the dynamics of non-fidelity regions where the queuing models' assumptions are violated.

Fisher et. al [13] present a solution to reduce peak CPU temperature for real-time systems. They model peak temperature for a set CPU frequency based on schedulability conditions. Thiele et. al [39] automate the calibration of these models to reduce simulation time. In some cases, their model explores policies that exceed the budget. Our approach never explores policies that exceed the budget. Both [13] and [39] capture the system behavior for fixed frequencies. Our model-driven approach models a system with dynamic or static frequencies.

Our modeling techniques can predict the job response times for actual systems that host complex workloads while being subject to given sprinting budgets. Combining the merits of queuing simulator and random forest, we are able to accurately explore the large space of system and workload configurations and identify near-optimal sprinting policies.

7 CONCLUSION

Computational sprinting problems use short, targeted bursts in processing speed to reduce whole system response time. Sprinting policies control (1) which query executions sprint, (2) how long they sprint, and (3) how much speedup they receive. However, subtle changes to sprinting policies have complex, unexpected effects. This paper showed that subtle changes in timeout settings (in either direction) can increase response time significantly. This paper proposes a model-driven approach, where sprinting policies are compared based on their expected response time. We show that model-driven approaches are plausible by creating an accurate performance model for computational sprinting policies. The key aspects to our model are (1) profiling workload characteristics, (2) accounting for dynamic runtime factors via machine learning, and (3) using effective sprint rate instead of marginal sprint rate as input into first-principles queuing simulation. We validated our model across multiple sprinting hardware, query semantics, and workload conditions. Our model-driven approach outperforms state-of-the-art results and widely used ad-hoc approaches.

Acknowledgments: We received tremendous feedback from our shepherd Alexander Matveev, Siva R. Meenakshi, Thomas Wenisch and Benjamin Lee. Our work was funded by NSF grants #1749501, #1350941, and SNSF NRP75 project Dapprox 407540_167266.

REFERENCES

[1] P. E. Bailey, A. Marathe, D. K. Lowenthal, B. Rountree, and M. Schulz. Finding the limits of power-constrained application performance. In *SC*, pages 79:1–79:12, 2015.

[2] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, 1999.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[4] J. Barr. https://aws.amazon.com/blogs/aws/low-cost-burstable-ec2-instances/, 2014.

[5] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL https://doi.org/10.1023/A:1010933404324.

[6] L. Y. Chen and N. Gautam. Server frequency control using markov decision processes. In *INFOCOM*, pages 2951–2955, 2009.

[7] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[8] C. Curtsinger and E. D. Berger. Coz: finding code that counts with causal profiling. In *SOSP*, pages 184–197, 2015.

[9] Datadog. 8 surprising facts about real docker adoption, Oct 2015. URL https://www.datadoghq.com/docker-adoption/.

[10] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, 2012.

[11] S. Fan, S. M. Zahedi, and B. C. Lee. The computational sprinting game. In *ASPLOS*, pages 561–575, 2016.

[12] G. Fiori, F. Bonaccorso, G. Iannaccone, T. Palacios, D. Neumaier, A. Seabaugh, S. K. Banerjee, and L. Colombo. Electronics based on two-dimensional materials. *Nature nanotechnology*, 9(10):768–779, 2014.

[13] N. Fisher, J. J. Chen, S. Wang, and L. Thiele. Thermal-aware global real-time scheduling on multicore systems. In *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 131–140, April 2009. doi: 10.1109/RTAS.2009.34.

[14] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytiä. Reducing latency via redundant requests: Exact analysis. In *Sigmetrics*, pages 347–360, 2015.

[15] H. R. Ghasemi and N. S. Kim. Rcs: runtime resource and core scaling for power-constrained multi-core processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 251–262. ACM, 2014.

[16] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ASPLOS*, 2015.

[17] I. Goiri, T. D. Nguyen, and R. Bianchini. Coolair: Temperature- and variation-aware management for free-cooled datacenters. In *ASPLOS*, pages 253–265, 2015.

[18] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. *SIGPLAN Not.*, 50(4):161–175, Mar. 2015. ISSN 0362-1340.

[19] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. F. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *HPCA*, 2015.

[20] M. Jeon, S. Kim, S. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: taming tail latencies in web search. In *SIGIR*, 2014.

[21] M. Jeon, Y. He, H. Kim, S. Elnikety, S. Rixner, and A. L. Cox. TPC: target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *ASPLOS*, 2016.

[22] J. Kelley, C. Stewart, N. Morris, D. Tiwari, Y. He, and S. Elnikety. Measuring and managing answer quality for online data-intensive services. In *IEEE ICAC*, 2015.

[23] J. Kelley, C. Stewart, N. Morris, D. Tiwari, Y. He, and S. Elnikety. Obtaining and managing answer quality for online data-intensive services. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 2(2), 2017.

[24] D. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, 1953.

[25] Y. Liu, S. C. Draper, and N. S. Kim. Sleepscale: runtime joint speed scaling and sleep states management for power efficient data centers. In *ACM SIGARCH Computer Architecture News*, 2014.

[26] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 11(2):146–178, May 1993. ISSN 0734-2071. doi: 10.1145/151244.151246. URL http://doi.acm.org/10.1145/151244.151246.

[27] N. Morris, S. M. Renganathan, C. Stewart, R. Birke, and L. Chen. Sprint ability: How well does your software exploit bursts in processing capacity? In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 173–178, July 2016. doi: 10.1109/ICAC.2016.61.

[28] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.

[29] Z. Qiu, J. F. Pérez, and P. G. Harrison. Variability-aware request replication for latency curtailment. In *INFOCOM*, pages 1–9, 2016.

[30] J. R. Quinlan. Induction of decision trees. *MACH. LEARN*, 1:81–106, 1986.

[31] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Computational sprinting. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-0827-4. doi: 10.1109/HPCA.2012.6169031. URL http://dx.doi.org/10.1109/HPCA.2012.6169031.

[32] D. A. Reed and J. Dongarra. Exascale computing and big data. *Commun. ACM*, 58(7):56–68, 2015.

[33] Rerout Lab Git Repo. http://reroutlab.org/projects.html, 2016.

[34] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *EuroSys Conf.*, Mar. 2007.

[35] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *IEEE ICAC*, 2013.

[36] S. Swanson and M. B. Taylor. Greendroid: Exploring the next evolution in smartphone application processors. *IEEE Communications Magazine*, 49(4): 112–119, 2011.

[37] A. Tchernykh, D. Trystram, C. Brizuela, and I. Scherson. Idle regulation in non-clairvoyant scheduling of parallel jobs. *Discrete Applied Mathematics*, 157(2):364 – 376, 2009. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/j.dam.2008.03.005. URL http://www.sciencedirect.com/science/article/pii/S0166218X08001285.

[38] E. Thereska and G. R. Ganger. Ironmodel: Robust performance models in the wild. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '08, pages 253–264, 2008. ISBN 978-1-60558-005-0.

[39] L. Thiele, L. Schor, H. Yang, and I. Bacivarov. Thermal-aware system analysis and software synthesis for embedded multi-processors. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 268–273, June 2011.

[40] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS*, pages 205–218, 2010.

[41] A. Verma, M. Korupolu, and J. Wilkes. Evaluating job packing in warehouse-scale computing. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 48–56. IEEE, 2014.

[42] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[43] C. Wang, B. Urgaonkar, A. Gupta, L. Y. Chen, R. Birke, and G. Kesidis. Effective capacity modulation as an explicit control knob for public cloud profitability. In *ICAC*, pages 95–104, 2016.

[44] H. Zhang and H. Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *ASPLOS*, pages 545–559, 2016. ISBN 978-1-4503-4091-5.

[45] W. Zheng and X. Wang. Data center sprinting: Enabling computational sprinting at the data center level. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 175–184. IEEE, 2015.