The Modeling and Management of Computational Sprinting

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Nathaniel Morris, B.S., M.S.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2021

Dissertation Committee:

Dr. Christopher Stewart, Advisor

Dr. Radu Teodorescu

Dr. Xiaorui Wang

Dr. Xiaodong Zhang

Abstract

Sustainable computing, dark silicon and approximate computing have ushered a new era in which some processing capacity is available only as ephemeral bursts, a technique called computational sprinting. Computational sprinting speeds up query execution by increasing power usage, dropping tasks, precision scaling, and etc. for short bursts. Sprinting policy decides when and how long to sprint. Poor policies inflate response time significantly. However, sprinting alters query executions at runtime, creating a complex dependency between queuing and processing time. Sprinting can speed up query processing and reduce queuing delay, but it is challenging to set efficient policies. As sprinting mechanisms proliferate, system managers will need tools to set policies so that response time goals are met. I provide a method to measure the efficiency of sprinting policies and a framework to create response time models for sprinting mechanisms such as DVFS, CPU throttling, cache allocation, and core scaling. I compared sprinting policies used in competitive solutions with policies found using our models.

To my parents, my fiancée and my mentor.

Acknowledgments

I am truly grateful for all of the support I have received over the years. My parents were always there cheering me on and my fiancée stood by my side and provided emotional support. I owe my PHD advisor many thanks for pushing me to strive higher and for him being patient with my transformation into a researcher. I know the road up until now has been rough, but I am looking forward to the new chapter in my life that follows my graduation. Once again, thank you everyone!

Vita

2003-2007	Trotwood High School - Trotwood,OH
2007-2012	Bachelor of Science, Manufacturing Engineering, Central State University
2012-2014	Masters of Science, Electrical and Computer Engineering, The Ohio State University
2015-2019	Masters of Science, Computer Science and Engineering, The Ohio State University
2015-2021	PhD student in Computer Science and Engineering, The Ohio State University

Publications

Research Publications

Isabelly Roche, **Nathaniel Morris**, Lydia Chen, Pascal Felber, Robert Birke, Valerio Schiavoni, "PipeTune: Pipeline Parallelism of Hyper and System Parameters Tuning for Deep Learning Clusters". *International Middleware Conference*, Delft, Netherlands, December (2020).

Eduardo Romero, Christopher Stewart, **Nathaniel Morris**, "Fast Inference Services for Alternative Deep Learning Structures". *ACM Symposium on Edge Computing*, Washington D.C, November (2019) (Poster).

Jianru Ding, Ruiqi Cao, Indrajeet Saravanan, **Nathaniel Morris**, Christopher Stewart, "Characterizing Service Level Objectives for Cloud Services: Realities and Myths". *International Conference on Autonomic Computing XVI*, Umeå, Sweden, July, (2019).

Jaimie Kelley, **Nathaniel Morris**, "Rapid In-situ Profiling of Colocated Workloads". *International Workshop on Big Data and Cloud Performance*, Paris, France, May (2019).

Nathaniel Morris, Indrajeet Saravanan, Pollyanna Cao, Jerry Ding, Christopher Stewart "SLO Computational Sprinting". *ACM Symposium on Cloud Computing*, Carlsbad, California, October, (2018) (Poster).

Nathaniel Morris, Christopher Stewart, Lydia Chen, Robert Birke, Jaimie Kelley "Model-Driven Computational Sprinting". *European Conference on Computer Systems*, Porto, Portugal, April, (2018).

Nathaniel Morris, Christopher Stewart, Lydia Chen, Robert Birke, Jaimie Kelley "Early Work on Modeling Computational Sprinting". *ACM Symposium on Cloud Computing*, Santa Clara, California, September, (2018).

Nathaniel Morris, Siva Meenakshi Renganathan, Christopher Stewart, Robert Birke, Lydia Chen "Sprint Ability: How Well Does Your Software Exploit Bursts in Processing Capacity?". *International Conference on Autonomic Computing XIII*, Wrzburg, Germany, July, (2016).

Jaimie Kelley, Christopher Stewart, Devesh Tiwari, Yuxiong He, Sameh Elnikety, and **Nathaniel Morris** "Measuring and Managing Answer Quality for Online Data-Intensive Services". *International Conference on Autonomic Computing XII*, Grenoble, France, July, (2015).

Fields of Study

Major Field: Department of Computer Science and Engineering

Table of Contents

Page

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

Currently, gains in processing capacity are a result of scaling out computer hardware. For example, modern CPUs are equipped with multiple cores clocked at lower frequencies compared to their predecessors which had fewer cores clocked at higher frequencies. Early integrated circuit (IC) design relied on the power density staying constant as transistors decreased in size and the switching frequency increased. This is commonly known as Dennard scaling which allowed computer hardware to increase processing capacity without increasing overall power consumption. Dennard scaling started to breakdown after 2006 because current leakage posed greater challenges at smaller transistor sizes. However, the transistor count in circuits are still growing but the performance improvement are more gradual compared to the improvements experienced by frequency increases. The end of Dennard scaling has led to sharp increases in power densities that prevent powering on all transistors at nominal voltage simultaneously, while keeping the circuitry within thermal limits. The portion of IC that cannot be powered on given a thermal design power (TDP) constraint is called Dark Silicon. Consequently, modern CPUs can only activate all of its transistors for short periods of time without exceeding its TDP. This offers a mechanism that provides temporary bursts in processing capacity.

Datacenters are increasingly using renewable energy from wind turbines or solar panels to reduce their dependency on less clean energy. Unlike other managed resources,

renewables are only available when the sun shines or the wind blows. This intermittency is challenging for datacenters to use renewables efficiently. Traditional techniques used to manage energy and other resources cannot easily manage resources with intermittent capacity. Renewables are affected by many environmental factors that change their availability and cause unpredictable supply. For example, renewables may only be produced during low activity periods. When renewables are available, datacenters must treat them as precious resources and maximize their performance impact when used. Renewables are just another mechanism that increase processing capacity for a limited amount of time.

Large scale applications such as data analytics and scientific computing far exceed available resources. In addition, they consume significant amount of time and energy during execution. The amount of data managed by datacenters will outpace the number of processors 5 to 1. Approximation techniques have been gaining popularity as a solution for reducing the amount of resources, time, and energy that is consumed. Techniques such as task dropping, data sampling, and precision scaling are used to speed up execution but not necessarily increase processing capacity. These techniques sacrifice computational accuracy to get large resource savings.

Dark silicon, sustainable computing, and approximating computing use transient resources to reduce execution time for short bursts. This is known as computational sprinting. The processing rate of a query can be increased temporarily when resources are available and certain conditions are met. For example, resources such as thermal headroom for overclocking and accuracy-leniency for approximation are accompanied by conditions that prevent the temperature of a CPU from exceeding the max and the output quality of a query from falling below 90% respectively. Computational sprinting is a resource management approach governed by policies. A sprinting policy specifies when to increase processing

capacity and by how much. When the sprinting policy is triggered, additional resources are allocated to a query (i.e. the query is sprinted) for a specific amount of time. Resources are revoked by the policy or when the budget is exhausted.

Cloud providers strive to maximize the utilization of their hardware. This results in the allocation of resources from a single machine to multiple query requests. Such systems set a service level objective (SLO) for each query type to define the expected performance provided to the client. In many allocation schemes, resources are over-provisioned so that SLOs are easily achieved. Less resources can be provisioned if queries are sprinted during opportune times that significantly reduce response time. Thus, increasing utilization of the system by enabling the colocation of more queries. Each colocated query contributes to the provider's profit. Overly aggressive policies will result in too many SLO violations that cause clients to withdraw from the service. This ultimately hurts the provider's profit. Operating hardware at high utilization benefits providers only if clients continue to use their service, which implies SLOs must be met.

Resources are allocated to avoid SLO violations but at the same time allocations drain the budget. The challenge arises when a sprinting policy must wisely make decisions to minimize the number of SLO violations while conserving the budget for later use. For example, the system receives a sudden surge of requests. The typical decision is to allocate more resources to compensate for excessive queuing delay. Assume the last allocation exhausted the budget. Soon after, the system receives another surge of requests 2X greater than the previous one. At this very moment, the resources used on the first surge would be better spent on the larger one. Effective sprinting policies must be aware of how additional resources impact response time and anticipate changes in system load.

Many of the proposed policies in prior work improve upon naive, greedy policies, showing the importance of good resource management policies. However, it is also important to explore a large space of possible management policies to compare between policies. Resource management software can use only one sprinting policy at a time. The policy used should significantly reduce response time, increase throughput or, more generally, perform well on key performance metrics. Empirical approaches measure performance directly to rank competing policies. Prior research relied on such empirical methods to highlight poor detection and wasteful triggering in simple policies. Empirical approaches are time consuming when there are many competing policies, for example, resource management software can define a new policy by adjusting tunable parameters. As a result, assessing the optimality of sprinting policies becomes even more challenging. In my research, I develop an efficient metric that can help determine whether poor performance is caused by inopportune triggering or insufficient sprinting speedup. Such a metric defines performance under a target policy divided by the best performance under any policy. Performance is a function of the sprinting mechanisms available to competing policies. Each is defined by workload, speedup and budget. The optimal performance is obtained from models or simulation.

My other research focuses on designing performance models for various sprinting mechanisms such as DVFS, core scale, and CPU throttling. Since computational sprinting makes processing rate and queuing delays interdependent, my work uses hybrid models consisting of a machine learning component and first-principles simulation. Machine learning models can characterize interdependence, but are slow to train when directly used for response time predictions. A hybrid approach that marries machine learning with models based on first principles reduces training time and complexity per prediction unit. This hybrid approach maps policies and workload conditions to response time. The workload conditions

include query semantics from profiling, 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.

Chapter 2: Sprint Ability: How Well Does Your Software Exploit Bursts in Processing Capacity?

Internet services can process queries faster by over clocking processors, exceeding circuit breaker capacity and returning lower quality answers. These solutions can have harmful long-term side effects, but they can be used safely in short bursts. *Sprinting mechanisms* use such techniques to boost processing rates for short periods and then safely disable them before their side effects arise. Recent examples of sprinting mechanisms in research and commercial products include:

- **Adrenaline [51]** uses DVFS to boost processor clock rates, exceeding the system power budget.

- **ApproxHadoop [47]** drops map tasks to speed up map-reduce computations, lowering the quality of final answers.

- **Data center sprinting [119]** temporarily over subscribes data center circuit breakers.

- **Redundant query scheduling [4, 63, 29]** starts redundant query executions, increasing cost per query.

- **Intel TurboBoost [85]** increases per-core clock rates above their rated operating frequency if the whole processor is below power limits.

Sprinting mechanisms can speed up query executions that demand heavy computation. Also, sprinting mechanisms can subdue workload surges that would otherwise cause queuing delays. However, sprinting mechanisms can improve response time only if they can be triggered. Resource management software must preserve precious bursts in processing capacity for moments where they are needed most.

Recent research proposes various policies to manage sprinting mechanisms, termed simply sprinting policies hereon. Across the board, the proposed sprinting policies perform better than policies that do not use sprinting mechanisms. Many of the proposed policies also improve upon naive, greedy policies, showing the importance of good resource management policies. However, it is also important to explore a large space of possible management policies to answer the following research questions:

- How much better are the best sprinting policies compared to proposed sprinting policies?

- Do good sprinting policies share conspicuous features that make them easy to identify?

- Are the best sprinting policies affected by factors that change, e.g., failures, query arrival rates or renewable energy. If so, what autonomics are needed to adapt sprinting policies over time?

This paper discusses *sprint ability*, a metric that assesses sprinting policies, i.e. how well a sprinting mechanism is managed. Sprint ability divides performance achieved under a targeted sprinting policy by the best performance achieved under any sprinting policy. If sprint ability equals 1, the targeted sprinting policy manages its sprinting mechanism better than (or equal to) all other sprinting policies. Sprint ability complements performance debugging tools. It separates the speedup provided by sprinting mechanisms from the

limitations of sprinting policies. To be sure, a sprinting mechanism that supports larger bursts and power budget will improve performance but not necessarily sprint ability. To improve sprint ability, resource management software must (1) use bursts more opportunely or (2) better balance the magnitude and duration of processing bursts.

The baseline for sprint ability is the best performance achieved under any sprinting policy. We argue for new simulation and modeling tools that output expected speedup given workload and sprinting factors. These tools will predict performance in a fraction of the time required to set up the system, configure the target sprinting policy and run tests. However, these tools present new research challenges. First, a core assumption in queuing theory (a widely used approach to model response time) is contrary to sprinting: Service time is supposed to be independent of queuing delay. In contrast, many sprinting policies invoke sprinting mechanisms only when queuing delay is large. For example, Adrenaline triggers DVFS when queries are within 50% of SLO limits [51]. Correlations between service time and queuing delay break queuing theory models, making closed-form results hard to obtain. To obtain response time across a wide range of sprinting policies, we designed a sprinting-aware simulator for simple M/M/k[1] Internet services. Our simulator considers workload factors (e.g., arrival and service rates) and sprinting factors (e.g., speedup from sprinting and sprinting frequency). It uses discrete-event simulation to model queuing and total response time per query, under a given sprinting policy and mechanism. The simulator is an efficient mean to explore large parameter space of sprinting policies and search for the optimal performance that is used to compute the sprint ability.

We used our simulator to evaluate the sprint ability of internet services under two sprinting mechanisms. The first mechanism speeds up query executions by 1500X but only 25%

[1]M/M/k is a system with k servers in which the arrival and service rates are exponentially distributed.

of query executions can trigger the mechanism in a 5-minute interval. This mechanism was inspired by ApproxHadoop [47]. Query executions are akin to map tasks that are dropped to speed up execution. For a given query, the sprinting policy in ApproxHadoop drops a fixed number of maps (i.e., 25%) as set by the user. We explored alternative sprinting policies where the drop rate changes at the end of each 5-minute interval. Our results showed that the sprinting policy that drops the same number of maps in each interval has sprint ability of 71%, and sprinting policies that drop 49% of maps once a day maximize performance.

The second sprinting mechanism speeds up query execution by up to 1.57X, but it has a limited energy budget. Sprinting policies choose how often to trigger the mechanism. This mechanism was inspired by Adrenaline [51]. We studied the proposed policy that triggers sprinting when (1) the query execution time neared SLO limits or (2) the query execution time fell above the 85^{th} percentile. In our tests, this policy achieved 75% sprint ability. With more degrees of freedom, these tests highlighted the wide ranging effects of sprinting policies.

This paper generalizes sprinting mechanisms as ephemeral processing bursts triggered by software. The hardware and software techniques used to burst processing capacity are orthogonal to the sprinting policies that determine when and how to sprint. It is our position that *there are rich research problems in setting and adapting sprinting policies.*

The remainder of this paper is as follows. Section 2.1 compares three simple policies. The discussion reveals challenges in setting sprinting policies. Section 2.2 defines sprint ability formally and makes the case that sprint ability is hard to measure without new research on performance models and system profiling. Section 2.3 presents a simulator that provides a first step toward new models on the effects of sprinting policies. Section 2.3.1 uses the simulator to consider sprinting mechanisms similar to ApproxHadoop

and Adrenaline. Our results suggest that there is room to improve recently proposed policies. Section 2.4 discusses related work.

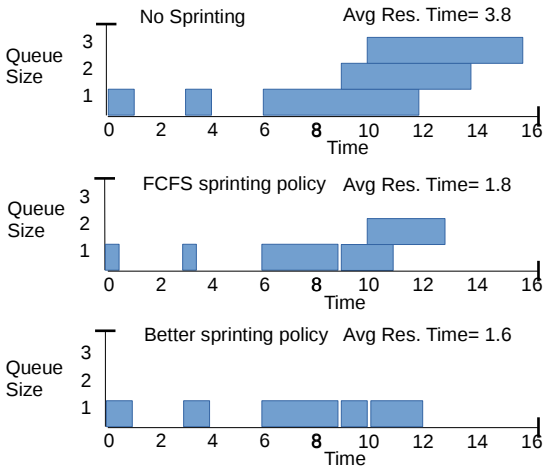

Fig. 2.1: Depiction of a sprinting mechanism that speeds up indivudal query executions. We compare response time under policies that (top) never trigger the sprinting mechanism, (center) greedily trigger via first come first serve and (bottom) trigger more opportunely.

2.1 Motivating Example

In this paper, a sprinting mechanism is hardware or software that allocates additional resources for processing queries. Dynamic voltage frequency scaling (DVFS) under an energy budget is a concrete example. It allocates additional voltage for short periods. Sprinting policies control how often and when to trigger these mechanisms. The terminology borrows from earlier operating systems concepts [64].

Figure 2.1 depicts query execution times, queue lengths and response time for an Internet service with access to a DVFS sprinting mechanism. Here and throughout the paper, we characterize sprinting mechanisms as a triplet: target workload, speedup per sprint and budget. In Figure 2.1, the DVFS mechanism targets individual query executions, speeds up their processing rate by 2X and has a budget of 4 sprinting seconds.

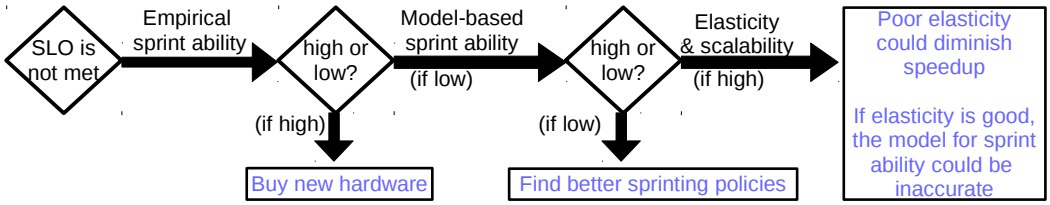

Fig. 2.2: Using sprint ability, empirical and model-based, to debug performance problems.

Figure 2.1 compares 3 policies that govern sprinting for the same query arrival workload. The top chart depicts query executions under a policy that never triggers DVFS sprinting. Queuing delay increases average response time to 3.8 seconds per query. The center chart depicts a policy that triggers sprinting for queries as they arrive until budget is exhausted. This fist-come-first-serve approach wastes resources on queries that execute quickly without sprinting. Finally, the bottom chart shows a policy that opportunely sprints on the third and fourth arriving queries. By triggering DVFS at the right time, this approach reduces query processing and eliminates queuing delay. Response time is 2.3X and 1.12X faster.

Figure 2.1 shows two key challenges in setting sprinting policies: (1) detecting which moments provide the greatest utility for sprinting and (2) carefully preserving budgeted resources for only those moments. Poor detection arises from poor scheduling or uncertainty regarding future demands. Resources are wasted when sprinting is unavailable due to triggering too often or inefficiently using resources.

2.2 Sprint Ability: Vision and Research Challenges

Resource management software can use only one sprinting policy at a time. The policy used should significantly reduce response time, increase throughput or, more generally, perform well on key performance metrics. Empirical approaches measure performance directly to rank competing policies. Prior research relied on such empirical methods to highlight poor detection and wasteful triggering in simple policies [51, 119, 63]. Empirical approaches are time consuming when there are many competing policies, for example, resource management software can define a new policy by adjusting tunable parameters. As a result, assessing the optimality of sprinting policies becomes even more challenging. We advocate that sprint ability offers an efficient metric to evaluate and improve sprinting policies across a wide range of sprinting mechanisms.

Particularly, sprint ability narrows competing policies by answering the following questions for a targeted policy: (1) Does the targeted policy exploit bursts in processing capacity well? (2) Are there better policies available? Equation 2.1 formally defines sprint ability as performance under a target policy (P_{trg}) divided by the best performance under any policy (P_{opt}). $\vec{c}$ is a collection of sprinting mechanisms available to competing policies. Each is defined by workload, speedup and budget.

$$SA = \frac{Perf(P_{trg}, \vec{c})}{Perf(P_{opt}, \vec{c})} \tag{2.1}$$

In our vision, P_{opt} is obtained from models or simulation. (If P_{opt} can be obtained from direct measurements, then empirical ranking remove the need for sprint ability.) Models and simulators can compute expected performance across a wide range of sprinting policies much faster than empirical approaches. There are two ways to compute P_{trg}. Empirical sprint ability (ESA) uses direct measurements. Model-based sprint ability (MSA)

use models or simulations. ESA and MSA are both useful to debug performance problems with sprinting policies. *Sprint ability does not replace response time, throughput or other first-class metrics.* We plan to use sprint ability to complement performance debugging in the presence of a sprinting mechanism. Specifically, sprint ability determines whether poor performance is caused by inopportune triggering or insufficient sprinting speedup.

Figure 2.2 illustrates performance debugging with sprint ability. Assume an SLO violation has occurred. If ESA is high (i.e., greater than 95%) then replacing the sprinting policy will provide only small performance gains. More powerful sprinting mechanisms are needed. If ESA and MSA are both low, the sprinting policy is causing unneeded slow down. Some modelling approaches may suggest competing policies to explore for ESA. If ESA and MSA disagree, other metrics can uncover the difference between model expectations and actual performance. For example, systems with poor elasticity may over state speedup by discounting adaptation time. A wide range of tools exist to align model expectations to actual performance [28, 27, 96]. The remainder of the section outlines research in managing and measuring sprint ability.

Systems and Adaptive Resource Management: It is important to show that sprint ability integrates nicely with existing systems. We would like to characterize exactly which pieces of code in today's widely used platforms yield high sprint ability. However, modern systems are complex, comprising multiple layers (e.g., platform, OS and networking stacks). Each layer may comprise thousands of lines of code. The relevance of systems code to sprint ability depends on the sprinting mechanism. For example, code about data sampling has more relevance to software-driven approximation bound by quality limits than to dark silicon sprinting limited by power.

Mechanism	Systems layer	Task	Challenge	Description
software-driven approximate computing	runtime platform	data sampling	detect when sprinting will have impact	sampling at a fixed rate for all queries does not minimize queue delay
	OS & distributed file system	key-value lookup	preserve resources	sampling data hosted on remote servers may limit speedup
	coordination service	resource containers	preserve resources	changing a query's sampling rate on the fly demands mechanisms to track execution over distributed resources
dark silicon	OS & runtime platform	thread manager	preserve resources	OS and runtime platforms are slow to detect new cores, limiting speedup
	runtime platform	detect slow queries	detect when sprinting will have impact	heavy query arrival rates should encourage frequent sprinting
cloud burst	distributed stores	spot market provisioning	detect when sprinting will have impact	when spot prices decrease, sprinting frequency can safely increase

Table 2.1: Open research problems for high sprint ability in adaptive resource management.

Table 2.1 provides examples of systems issues that affect sprint ability under different categories of sprinting mechanisms. While the taxonomy is not complete, it outlines interesting challenges for the autonomic computing field. For example, in software-driven approximation, fixed data sampling across all queries is similar to FCFS policy described in Section 2.1. Policies that use variable, time changing rates could use approximation opportunely. In Section 2.3.1, we provide early results that confirm this opportunity. This problem along with tail detection are concerned with improving choices about when to sprint. Research on these topics directly improve sprint ability.

Sprint ability also improves when sprinting mechanisms are preserved for longer periods. In Table 2.1, we highlight slow core detection from Apache YARN. The problem is that core scaling spends a portion of its energy budget when systems software can't use it. Both policy and mechanism can attack this problem. Policies that target workloads in early stages, before the resource manager provisions nodes, can improve detection

speeds—increasing sprint ability. In contrast, YARN could better support core scaling as a sprinting mechanism. This research will improve speedup from sprinting but could improve or degrade sprint ability. Looking closely at Equation 2.1, we note that such research extends the vector of candidate constraints, i.e., a larger collection of sprinting mechanisms in $\vec{c}$. If this increases P_{opt} by more than P_{trg}, sprint ability will decrease—even though speedup improves. Research that improves the speedup of sprinting mechanisms is best evaluated using performance improvement (not sprint ability). However, it is important to measure sprint ability whenever there are major speedups as the decision making process may need to change fundamentally.

Models for Optimal Performance: Sprint ability needs to model the best performance across a wide range of policies. Building accurate models is challenging. However, our charactization of sprinting mechanisms (workload, speedup, budget) reduces complexity. For this paper, we discuss two possible research solutions. The first and ideal solution would be closed-form queuing models that capture the effects of sprinting. These models would consider the speedup gained from sprinting and the decision making process for sprinting. However, effective sprinting that targets reduced queuing delay breaks the following fundamental assumption underlying most queuing theory models: "the presence of a long queue is supposed to have no effect on the speed of service. [60]" Queue-dependent service times complicate Markovian analysis. Multi-level queues could yield closed form for certain sprinting conditions, e.g., coarse grained server setup times [43]. Further, replication for predictability and redundancy also offer some sprinting benefits, albeit with limited choice in query targeting. We call for continued research on such models. The second solution is to build accurate simulators. To this end, Section 3.1.2 presents a first-cut and open-source simulator for sprinting targeted at query executions.

Profiling Speedup Due to Sprinting: Speedup due to sprinting is hard to measure. As profiling software must know when sprinting is on. This requires advanced context tracking. Some sprinting mechanisms, such as approximation, the performance with sprinting off simply requires extending query execution with sprinting on [59] However, other mechanisms require two totally separate query executions. Another approach uses statistical models to understand average speedup.

2.3 First Steps: A Simulator

We built an M/M/k like simulator that supports a wide range of sprinting mechanisms and policies. In addition to arrival and departure rate parameters, the simulator accepts speedup and budget parameters that describe a sprinting mechanism. In its current version, our simulator only supports mechanisms that accelerate specific query executions. The simulator accepts parameters on the frequency with which sprinting policy triggers sprinting mechanisms, including support to target specific types of queries (e.g., tail response time).

Figure 2.3 illustrates our proposed discrete-event simulator. Prior to the first discrete time step, the simulator creates an array of query objects. A query is defined by its arrival and service time. Arrival time is relative to the first tick (0 time). Service time is absolute. In M/M/k mode, the simulator increments the clock by one, checks for an arrival, then checks the queue for a departure. In sprinting mode, the simulator executes sprinting policies at each time step. It marks whether the query execution triggered a sprinting mechanism and when its execution ended. After the simulation, it checks to see if the policy would have violated budgeted power, quality loss or cost by accelerating too many queries for too long.

Limitations: As a first step, our simulator allows us to explore the effects of a wide range of policies. However, simulation is much slower than analytic models. Closed-form models for response time with sprinting should eventually replace this simulator. Our team has already begun working on such models. However, queuing theory models are fundamentally harder when the service rate depends on the queue size. Ghandhi et al. needed multi-level hidden Markov models to build queuing models for elastic services where the queue

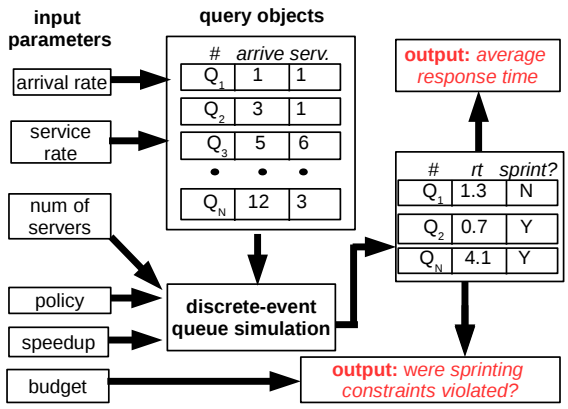

Fig. 2.3: Architecture of the simulator used to explore performance under a wide range of policies.

size adjusts the number of servers slowly over time [43]. However, sprinting is harder to model because the increase in processing capacity is ephemeral (by definition, not lasting to steady state) and applied to individual executions (rather than all queries).

The implementation of our simulator could simplify the specification of policies and mechanisms. Currently, the package is written in GoLang. New policies, excluding minor parameter tweaks, require new source files. Also, power budgets are checked after the simulation. This makes it hard to simulate approaches like data center sprinting that rely on the mechanisms to prevent sprinting too often.

2.3.1 Early Results

We used our simulator to compute model-based sprint ability for policies proposed in Adrenaline and ApproxHadoop [47, 51]. We extracted key features regarding the sprinting mechanism (speedup and budget) and sprinting policy in each paper. Our simulator computed the expected response time under a wide range of policies and searched for the best performance defined in the denominator of the sprint ability.

ApproxHadoop: ApproxHadoop subsamples data and drops map tasks to reduce running time for Hadoop jobs. We focused on the configuration where all data was used and map tasks were dropped. In this configuration, the price for dropping tasks is lower answer quality. Specifically, ApproxHadoop targets Hadoop jobs that output statistical answers. Answer quality is defined using the 95th percentile bound on the absolute error of the output (i.e., 1% - error). Users specify a target for answer quality and ApproxHadoop then completes each job quickly (dropping map tasks) while respecting the target quality. By default, ApproxHadoop drops map tasks randomly. We consider this policy and an alternative where ApproxHadoop drops map tasks that run slowly. We term the latter as ApproxHaddop with straggler prediction. Finally, we test the hypothesis of not using the same target answer quality for each query. Instead, we allow ApproxHadoop to sprint at a higher rates (up to 5% more dropped map tasks) as long as the average quality after 1 day is within budget. Specifically, our metric of merit is average response time across 96 jobs issued throughout the day. Table 2.2 shows the settings associated with our tests inspired by ApproxHadoop.

ApproxHadoop Simulation Settings		
mechanism	speedup	budget
drop maps	1,500X	avg. quality = 97.5%
policy name	query selection	worst target quality
default	random	97.5%
rand	random	95 – 97.5%
strag	stragglers	95 – 97.5%

Adrenaline Simulation Settings		
mechanism	speedup	budget
DVFS	1.57X	energy = 9.6
policy name	query selection	frequency
default	tail	15%
rand	tail	10 – 50%

Table 2.2: Sprinting mechanisms and policies simulated [47, 51].

Figure 2.4 plots CDF of sprint ability under two different ApproxHadoop sprinting mechanisms, namely random and straggler dropping. Each point in CDF represents a particular sprinting policy setting, i.e., frequency, measured at the simulator. Particularly, the sprint ability achieved by ApproxHadoop's default strategy is at 71%. We tested random policies that allowed various worst target quality at intervals .1%. Thanks to our simulator, we are able to efficiently explore 250 settings. And, only 74 out of 250 met the quality budget. ApproxHadoop's default strategy out performed only 3. The best policy allowed sprinting with quality loss of 4.5% for one job. As shown in the ApproxHadoop paper, dropping map requests can cause significant reduction in response time even while degrading answer quality only slightly. Comparing the random and straggler dropping policies reveals the importance of well targeted workloads on sprint ability. Straggler reduces the variance of sprint ability substantially. The median under straggler dropping policy that does not violate budget achieves over 97% sprint ability.

Adrenaline: Adrenaline exploits a DVFS mechanism capable of voltage transitions in nanoseconds. It significantly reduces energy wasted transitioning to higher frequencies.

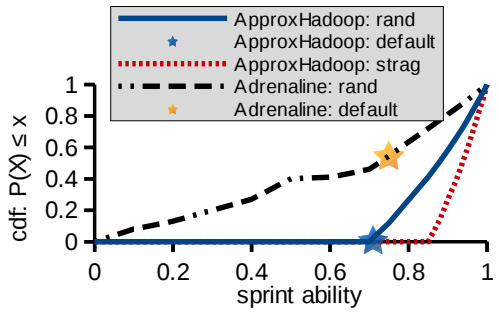

Fig. 2.4: Sprint ability for policies proposed in ApproxHadoop and Adrenaline.

With this technology, Adrenaline sets out to speed up query executions that are likely to violate SLO. It targets (i) query executions that exceed the 85th percentile in service time and (ii) queries with total execution time within 50% of the SLO target, i.e queries whose response time exceed 500 ms. In the paper, a DVFS increase from 1.4 GHz to 2.1 GHz provide 1.57X speedup. There is a global energy budget of 30% of the baseline.

For this paper, we changed the percentile that constitutes the tail. To stay within energy budget, the change in tail was matched by a proportional change in DVFS max speed. More aggressive tail settings provided less speed up per sprint. Using data from the paper, we set the service rate to 5 ms. The utilization was 90% (labeled high in the paper). Figure 2.4 shows the CDF of sprint ability under any sprinting frequency explored by the simulator. One can see that sprint ability in Adrenaline had higher variance than ApproxHadoop. There are two reasons. First, our policies prohibit invalid settings. Second, Adrenaline boosts performance significantly under high utilization. Poor policies that essentially disable sprinting mechanisms cause significant harm. Figure 2.5 shows the speedup of the Adrenaline sprinting mechanism under a fixed speedup as the tail threshold increases. This figure highlights the outsized effects under 90% utilization.

2.4 Related Work

Sprint ability complements metrics used to guide resource management, e.g., response time, scalability, and elasticity. This section overviews existing metrics and draws key contrasts, making the case for the research community to use sprint ability more often.

Elasticity measures how quickly a system can adapt to workload changes by provisioning and deprovisioning resouces [50]. Elasticity subsumes scale-out scalability which measures how efficiently a system can continuously add resources. These metrics concern the detection of resources and distribution of work. Systems with poor elasticity and scalability are hard to model, making sprint ability hard to compute. However, elasticity presumes changes are caused by workload rather than ephemeral resources. Systems that require workload surges to scale, e.g., workloads that load balance at coarse granularities, can achieve high elasticity but poor spint ability. We expect future processors will support fine-grained sprinting within the context of a single query execution. Support for ephemeral resources that are available for very short bursts distinguishes sprint ability. Auto scaling techniques can not support short bursts yet [43, 44].

Research on powering systems with intermittent renewable energy shares a key feature with sprinting: Variability is supply side not demand side. Many recent papers propose policies to quickly adapt resources in response to solar outages [110, 89]. These systems coupled with recent papers on sprinting mechanisms have focused on proposing good policies. In contrast, sprint ability answers (1) does a sprinting mechanism permit policies that provide sufficient SLO, etc. and (2) are there policies better than the current policy.

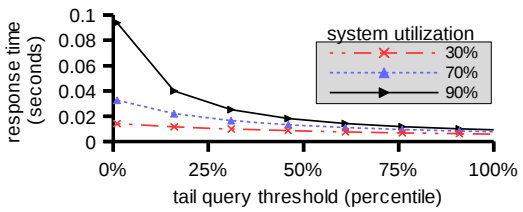

Fig. 2.5: Speedup from sprinting as a function of system utilization.

2.5 Conclusion

Sustainable computing, dark silicon and approximate computing have ushered a new era in which some processing capacity is available only as ephemeral bursts, a technique called sprinting. New techniques and metrics are needed to achieve lower response time and high throughput with sprinting mechanisms. Sprint ability complements existing metrics by distinguishing the role of software policies from sprinting mechanisms. This paper argues that sprint ability, or similar metrics, should become a part of resource management vernacular. We built a simulator to start evaluating sprint ability. Early results revealed that proposed, good policies are not always best. We have open sourced our simulator to encourage others to begin studying sprint ability [83].

Chapter 3: Model-Driven Computational Sprinting

Modern processors are constrained by increasingly tight power caps [81]. 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 [79]. Sprinting helps workloads meet response time requirements specified by SLOs [91, 51] and interactive applications [79]. DVFS [51, 115], core scaling [49], and CPU throttling [105] 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 [51]. 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 [99, 68, 49], (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 3.1 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 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

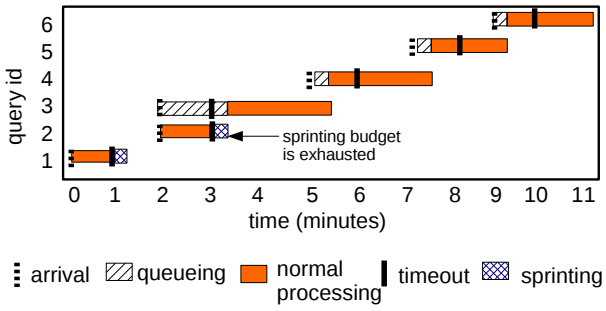

Fig. 3.1: Query executions under a tight sprinting budget. The first two queries drain the budget. Remaining queries can not sprint despite slow response time.

can 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 [51] and Few-to-Many [49].

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 [9]. 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 [26]. 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 3.1 outlines our design for model-driven computational sprinting. Section 3.2 evaluates our modeling approach across cloud benchmarks and sprinting hardware. Section 3.3 studies speedup and cost savings from improved sprinting policies. Section 3.4 frames open challenges to widely deploy and extend model-driven sprinting. Section 3.5 overviews related work in the area of computational sprinting and modeling for highly dynamic systems. Section 3.6 draws conclusions.

Fig. 3.2: Our approach combines workload profiling, queue simulation and machine learning (shown in red dotted squares). Black squares show inputs provided by users.

3.1 Design

Figure 3.2 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.

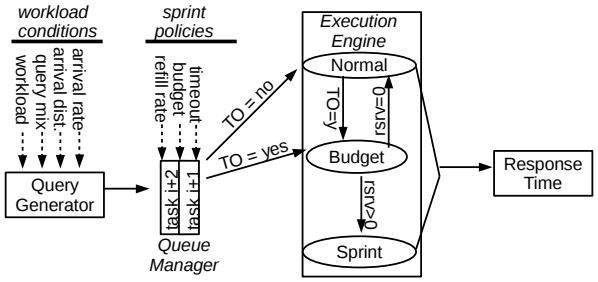

Fig. 3.3: Key instrumentation points for workload profiling in our approach.

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.

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.

3.1.1 Workload Profiling

As shown in Figure 3.3, 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 3.2, we evaluate our ability to linearly interpolate between clusters to predict response time.

3.1.2 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 [83]. 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 3.1, 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

```
// 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, μe, 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, μe)
  }
  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.

time. To be clear, all variables in this equation align with Algorithm 1 and clock is captured immediately after timeout.

$$depart = clock + (1-\tau)\cdot\bar{\mu}\cdot\frac{\mu_e}{\mu}, where \tag{3.1}$$

$$\tau = (clock - start)/\bar{\mu}$$

3.1.3 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

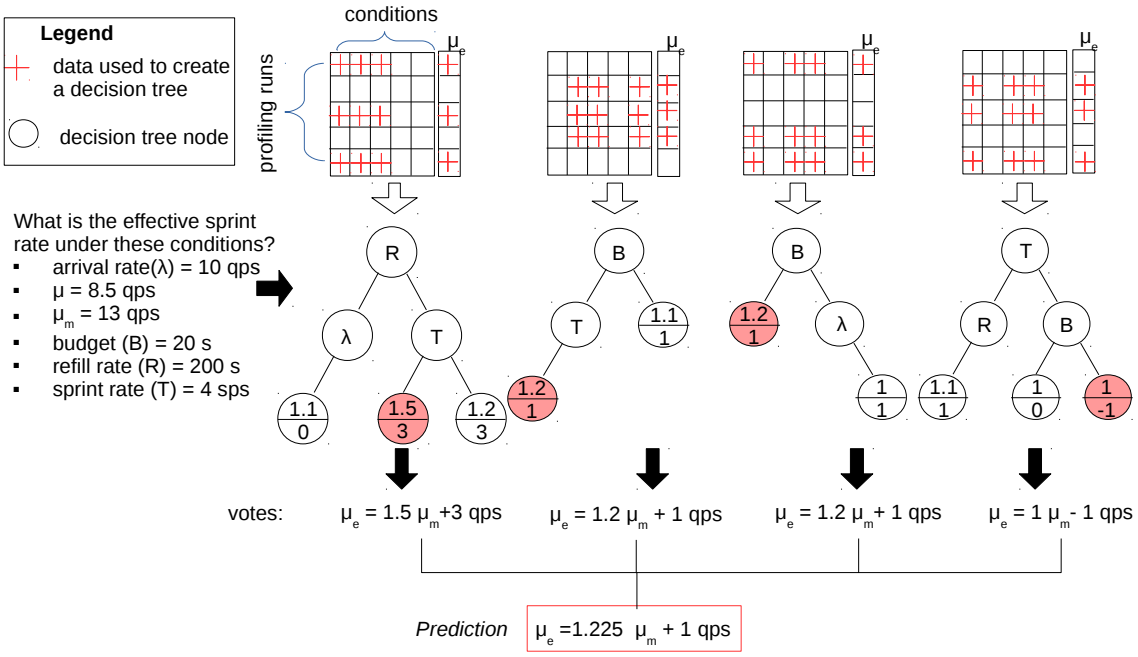

Fig. 3.5: Creating and using random decision trees to classify effective sprint rate.

sprinting, see Equation 3.1. 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 3.2 formalizes our model. RT_{wp} is the response time function for workload profiling. The input is tested workload conditions $\vec{F}$ 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.

$$\mu_e = \mu_m + \min|x|, \textit{where} \tag{3.2}$$

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

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.

3.1.4 Random Decision Forest

Random Decision Forest (RDF) is a combination of decision tree predictors [13]. 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 3.5 depicts the subsampling process. Columns represent workload conditions and sprinting policies ($\vec{F}$) 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 [77]. 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 $V(S)$ 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 ($V(S_{F_i=k})$ and (2) the complement ($V(S_{F_i\neq k})$). As shown in Equation 3.3, 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.

$$\max_F gain_i = V(S) - \frac{V(S_{F_i=k}) + V(S_{F_i\neq k})}{2} \tag{3.3}$$

As shown in Figure 3.5, 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 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.

3.2 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 3.1(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 3.1(B) describes sprinting hardware in our tests. **DVFS** and **CoreScale** use a Xeon 2660 processor. **DVFS** uses Pupil [115], 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 [46, 66]. **EC2DVFS** used an EC2 Extra Large C-class instance (circa 2017). Here, we sprint by changing P-States, i.e., we set DVFS directly.

(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

Table 3.1: Identifiers (IDs) for models, hardware and workload in our experiments.

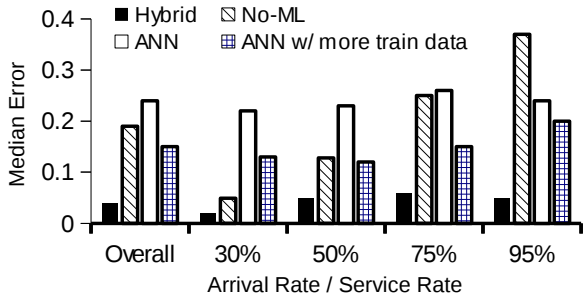

Fig. 3.6: Absolute relative error produced by competing performance modeling approaches.

Table 3.1(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)

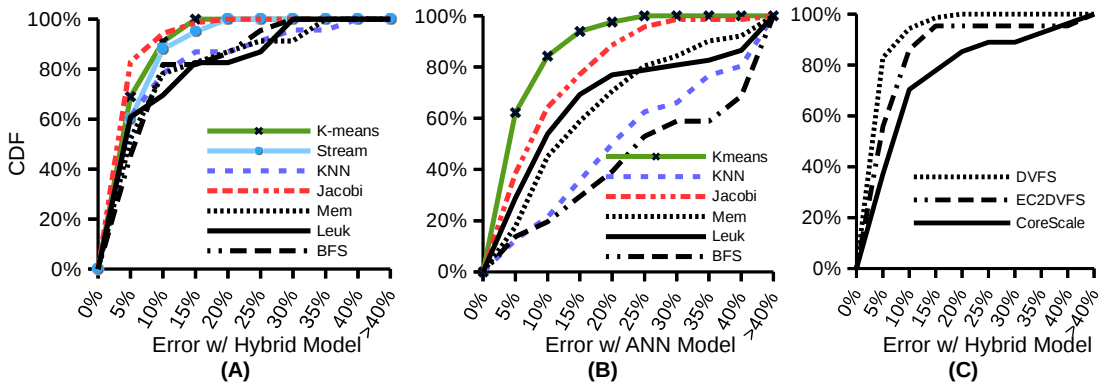

Fig. 3.7: CDFs of prediction error across workloads with hybrid approach.

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 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.

3.2.1 Impact of Modeling Approach

Figure 3.6 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%.

3.2.2 Impact of Query Execution Semantics

Figures 3.7(A) and 3.7(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.

3.2.3 Impact of Sprinting Hardware

Figure 3.7(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.

3.2.4 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

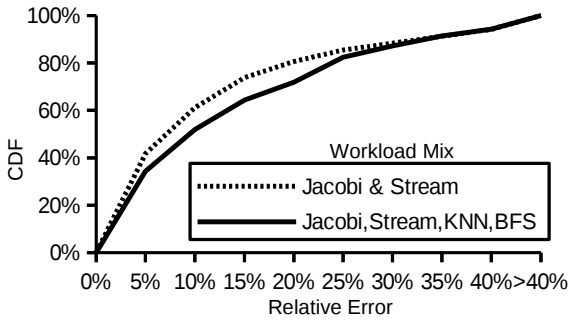

Fig. 3.8: The cumulative distributions of prediction error for two distinct mixed workloads.

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 3.8 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%.

3.2.5 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 3.9 shows average prediction error for each grouping. 75^{th} and 25^{th} 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.

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 3.9 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.

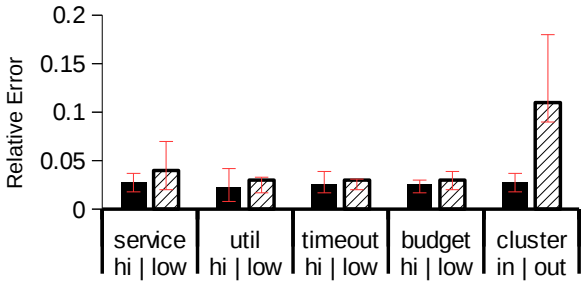

Fig. 3.9: Impact of service rate, arrival rate, timeout, budget and cluster sampling.

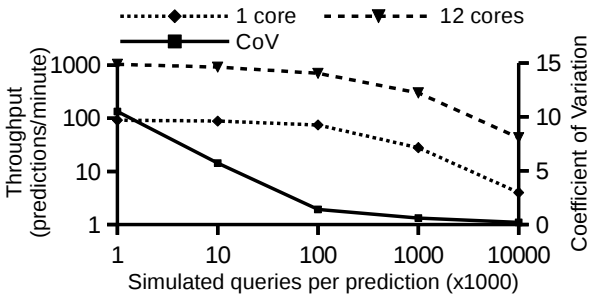

Fig. 3.10: Throughput and variance for response time predictions using our timeout-aware simulator.

or

3.2.6 Predictions Per Minute (Overhead)

Figure 3.10 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 3.10 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.

3.3 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.

3.3.1 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.

3.3.2 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 3.4 formalizes the problem.

$$MIN(RT) : \exists t \geq 0, RT = P_M(S_{F \neq t}, t) \tag{3.4}$$

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 3.5.

$$a = \begin{cases} 1 & \text{if } RT_o - RT_n > 0 \\ e^{\frac{RT_o - RT_n}{Z}} & \text{otherwise} \end{cases} \tag{3.5}$$

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.

3.3.3 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

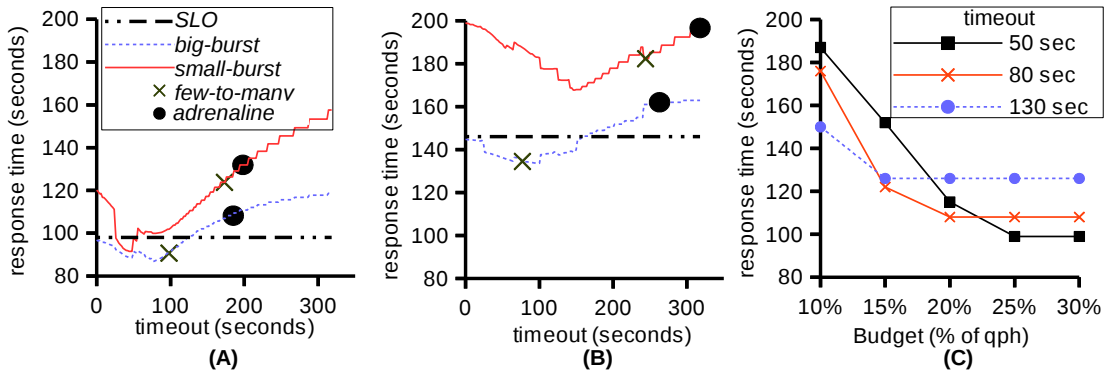

Fig. 3.11: Our model-driven approach was used for space exploration on DVFS. The vertical axes are the expected response times for a workload. Exploring timeout policies with (A) Jacobi kernel and (B) Mix I (Jacobi & Stream). (C) Response time as sprinting budget and timeout vary. We report budget as percent of sustained processing rate.

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 [49]. 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 [51]. Sets timeout to the $85^{th}\%$ percentile of non-sprinting response time.

Results: Figure 3.11(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 3.11(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 3.11(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 [49]: Under low utilization, all query executions should sprint aggressively but, under heavy utilization, resource managers must sprint for the most needy queries.

3.3.4 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 [9]. 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 3.12 compares revenue per node of the following approaches to set sprinting policy.

+ AWS: Sets a fixed sprint rate and budget 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 3.12 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 3.76X. By its nature, sprinting shrinks the tail [51]. 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 3.13 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

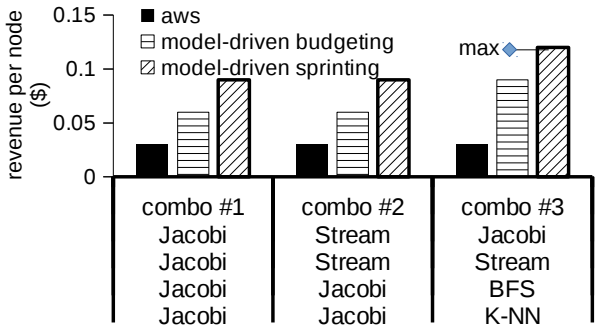

Fig. 3.12: Dollars earned for a burstable instances which optimizes the sprinting policy based on budget and timeout.

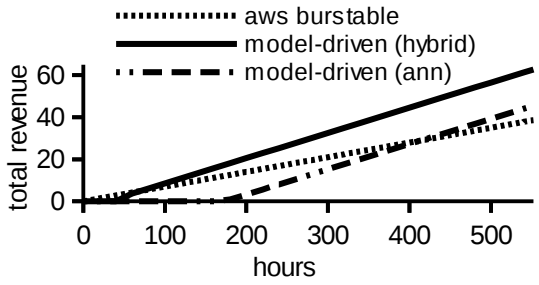

Fig. 3.13: The number of hours required to offset the profiling cost using our model.

longer, but this approach is eventually cost effective too. Over the lifetime of a virtualized server [26]. Model-driven sprinting with our hybrid model increases revenue by 1.6X.

3.4 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 [6, 7]. Intellectual descendants of these systems include Omega [104], 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 [19].

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 [47, 59, 57]. 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.

3.5 Related Work

Systems limited by *dark silicon* [36, 102] cannot sustain peak processing rates. Sprinting mechanisms enable peak processing in bursts and have been implemented across the whole system stack from transistors [40] to processors [98] to racks [81] to data centers [51, 119] and cooling systems [48]. Sprinting policies [79, 37] 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.

3.5.1 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 [115] runs a wide range of offline experiments to build a model of phases' execution to performance improvement. Coz [25] uses static program analysis to identify phases and reduce training time to characterize per-phase benefits. Bailey et. al [5, 49] 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 [37] 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.

3.5.2 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 [54, 55] 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 [104], 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 [92].

Verma et al. [103] 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 [105] 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 3.3 uses our model-driven approach to explore sprinting policies in this scenario.

3.5.3 Model-driven approach

Queuing models can predict response time for server systems [91, 60]. However, these models assume queuing delay and service time are independent. Interdependent queuing and service times lead to complicated models [70]. Recent research have created models for specific conditions and sprinting mechanisms, e.g., query replication [45, 76], admission control and dynamic voltage scaling [18]. Determining the optimal service rate for servers [75] 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 [18] model the dynamic voltage scaling problem of a single server as a discrete time Markov decision process by leveraging fluid approximation. Gardner et. al [45] 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 [76] 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 [100] 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 [41] 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 [101] 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 [41] and [101] 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.

3.6 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.

Chapter 4: Performance Modeling for Short-Term Cache Allocation

4.1 Introduction

Processor caches use SRAM cache lines to speed up main memory accesses. Modern processors can now allocate individual cache lines to specific workloads directly [53]. Such cache allocation can speed up workload execution, conserve cache lines during normal execution and enable workload collocation where multiple workloads share the CPU cache [97]. For example, online services can allocate a few cache lines for most query executions but allocate many lines to speedup targeted queries. However, if collocated services contend for shared cache lines or if a workload is allocated too few lines, performance suffers and response time goals, as stipulated in a service level objective (SLO), may not be met [118, 69, 34].

Intel Cache Allocation Technology (CAT) supports dynamic cache allocation for the last level cache (LLC). This enables *short-term allocation* wherein a workload gains temporary access to cache lines during its execution [53]. Online services can use short-term allocation to speed up slow queries and meet response time goals. Consider a social networking website [20]. A user query initiates processing across multiple Docker containers. If the query is still being processed after 800 milliseconds, the query execution could be in danger of violating response time goals. Short-term cache allocation policies may use

a timeout mechanism to allocate additional cache lines to the remaining Docker containers, speeding up their execution. Of course, allocating additional cache to one workload impacts other collocated workloads that share the cache [84]. Systems software can mitigate the slowdown by setting policies that manage how often workloads request short-term cache allocation.

This paper presents a performance modeling approach that, given a short-term allocation policy, predicts response time for collocated workloads. Our models can be used to compare policies and uncover settings that yield low response time for each collocated workload. Our approach combines workload profiling, machine learning and first-principles modeling, extending prior approaches to model short bursts in computational power [73, 52]. However, prior work did not consider a shared cache where a short burst can speed up a target workload but also slow down collocated workloads. If collocated workloads counter slowdowns by requesting short-term cache allocation more often, cache contention increases and further degrades response time.

Our approach models effective cache allocation, i.e., speedup under a short-term allocation policy normalized by the gross increase in resource allocation. Intuitively, effective cache allocation captures the effect of additional cache lines on response time. It is sensitive to dynamic runtime factors including cache usage during query execution, contention with collocated workloads, and queuing delay from concurrent executions. These factors can have large, non-linear effects [118]. For example, in some settings, we have observed workloads that manage a 2X increase in LLC cache misses without significant increases in response time. Linear models err on such settings by conflating cache usage counters with the underlying processes affecting response time. In our approach, we use deep learning techniques to group combinations with similar effective cache allocation but potentially

disparate cache usage. We also use representational learning to capture spatial relationships between cache usage counters. Combined, deep and representational learning yield powerful, new machine-learning features that uncover hidden but recurrent patterns of contention that capture the effects of cache allocation on response time better than hardware counters alone.

Our approach profiles cache usage for each collocated workload in a test environment. We use this data to train deep learning models of effective cache allocation. While deep learning techniques normally perform best with large training sets, online services may be collocated for only a short period [24], limiting time for profiling. A key challenge is to devise novel modeling techniques that have *low overhead*, i.e., they model response time well with limited profiling time. Specifically, our approach uses queuing theory principles to convert effective cache allocation to response time. This lowers the complexity of the deep-learning models and reduces profiling time.

We used Intel CAT tools [53] to implement short-term cache allocation by tracking query executions at runtime. We used Deep Forests [120] for deep and representational learning. We evaluated our modeling approach with a wide range of realistic online services, including: (1) Apache Spark executing the k-means clustering algorithm using iterative, parallelized stages. (2) Redis, a widely used key-value store with fast response time ($<$1 second) response. (3) Rodinia micro-benchmarks used to stress high performance computers and individual processor components. And (4) Social, a realistic macro benchmark that captures the workload of a social networking service. Social uses 36 micro-service components spread over 30 Docker containers. We collocated these services on Xeon processors, allowing them to share LLC cache, and evaluated the accuracy of our modeling approach to predict response time. We observed absolute percentage error

(<12%). Our modeling approach reduced error by 4.1X and 1.6X compared to approaches that eschew deep learning for simple linear regression and approaches that employ only deep learning respectively. In terms of overhead, our approach profiled workloads for 30 minutes. We observed that lower (15 minutes) and higher (2.5 hours) overhead produced 14% and 8.6% error.

We used our models to explore short-term allocation policies in collocated settings. For each service, we used our models to find policies with low response time. We compared response time under our policies to static cache partitioning policies used widely in practice. Our policies lowered 95^{th}-percentile response time by up to 2.6X. We also compared our approach to static allocation, workload-aware cache partitioning [108] and IPC-driven dynamic cache allocation [52]. Our approach sped up average response time by 1.3X, 1.3X and 1.2X respectively. Social networking, Redis and Spark workloads achieved up to 2X speedup. Finally, we used the concepts learned by our deep-learning models to cluster workloads with similar cache behaviors and identified a complex interaction between arrival rate, service time and timeout that affects response time for short-term allocation. Clustering using only the hardware cache counters did not reveal the interaction.

This paper uses deep learning techniques to model and manage increasingly dynamic processor caches. Our approach represents cache usage as a large, multi-dimensional vector, richly characterizing the entirety of query executions. Deep learning techniques allow us to extract patterns hidden in these vectors. Our contributions are:

1. We consider short-term cache allocation, where dynamic cache allocation speeds up a targeted query execution by providing access to shared cache lines. This mechanism adds a temporal dimension to cache allocation.

2. We present a modeling approach to predict response time for workload collocation and cache allocation policies.
3. Our approach uses deep and representational learning to characterize the complex relationship between cache usage counters and response time.
4. We show that our model predicts response time well, can be used to find good policies, and provides insight on key factors affecting performance.

This paper is organized as follows: Section 4.2 is a primer on dynamic cache allocation, the targeted workload and system management goals. Section 4.3 presents the design of our modeling approach. Section 4.4 describes our implementation. Section 4.5 evaluates response time predictions, compares competing approaches and studies model-driven policy selection. Section 4.6 presents related work. Section 4.7 provides discussion and draws conclusions.

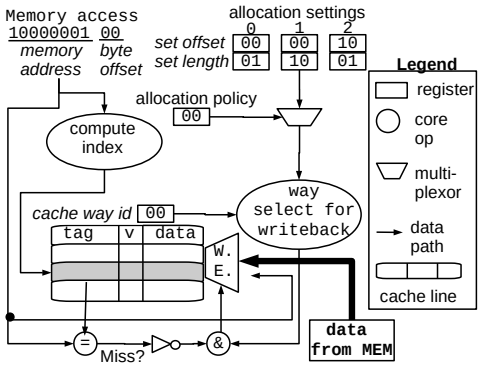

Fig. 4.1: Data path for dynamic cache allocation.

4.2 Cache Allocation Technology

Dynamic cache allocation manages which query executions can use cache lines at runtime [97]. Figure 4.1 depicts digital logic for dynamic cache allocation. Memory accesses trigger TLB translation, then the address is split into set (index), tag and offset. Cache lines with matching sets are called ways. A cache hit occurs when a cache line in a way stores a tag matching the memory address. On a miss, the cache installs the data read from memory. Dynamic cache allocation controls write enable (WE) logic. In Figure 4.1, query executions can write to contiguous cache ways defined by way offset and length. The first allocation setting installs data to cache way 00 or 01. The second setting allows writes to ways 00, 01 and 10.

The allocation policy decides which allocation setting applies. For static allocation, systems software can map a process id to an allocation setting, creating classes of service. For dynamic allocation, systems software can change settings at runtime.

Short-Term Cache Allocation for Online Services: Query executions that complete slowly can hurt revenue for online services. Increasingly, online services use response

time objectives to drive resource management [34]. LLC cache is a valuable resource that reduces response time by reducing main memory lookups [90].

Online services can monitor query executions, flag slow executions and try to speed them up. For example, computational sprinting methods use short, unsustainable bursts from DVFS and core scaling [114, 37, 3]. Short-term cache allocation speeds up queries by granting temporary access to additional cache lines, for example, by switching from *allocation setting 0* to *1* in Figure 4.1 for the remainder of a query execution.

Short-term allocation presents two competing goals: (1) slow query executions should receive short-term cache allocation and (2) baseline response time for normal query executions should not be affected by short-term allocations for collocated workloads. To achieve these goals, allocation settings should support (1) *private* LLC cache lines that ensure baseline performance and cannot be accessed by collocated workloads and (2) *shared* lines that can be allocated to speed up slow executions.

The Impact of Contiguous Cache Allocation: With Intel Cache Allocation Technology, cache allocation settings must be contiguous. This design has important consequences if collocated workloads reserve private cache for baseline performance.

Conjecture: Under contiguous allocation, private cache are disjoint.

Proof: Let A reflect the finite, set of cache allocation settings supported on a processor. Each contiguous allocation can be represented as an order pair of offset and length $(\mathrm{o_a}, \mathrm{l_a})$ where $0 \leq a < |A|$. A short-term allocation policy is a pair of allocation settings (a, a, t) where a timeout t triggers a temporary switch from default a to a'. The proofs below elide the dynamic timeout t to simplify the notation for static analysis. Intuitively, private cache lines must be allocated in a and a'. Further, collocated workloads can not access the private cache lines in (a, a'). Equation 4.1 describes these properties for a cache line with offset v

in the set of private cache lines $V_{(a,a')}$.

$$v \in V_{(a,a')} \rightarrow o_a \leq v < (o_a + l_a) \wedge$$
$$o_{a'} \leq v < (o_{a'} + l_{a'}) \wedge \forall \hat{a} \in \left[A - a - a'\right] (v < o_{\hat{a}}) \vee (v > o_{\hat{a}} + l_{\hat{a}}) \quad (4.1)$$

To prove by contradiction that private cache are disjoint, assume there exists private cache in short-term allocation $\left(\bar{a}, \bar{a'}\right)$ that falls between private cache lines v_0 *and* $v_1 \in V_{(a,a')}$. The following must hold: $\exists v_c \in V_{\left(\bar{a},\bar{a'}\right)} : o_a \leq v_0 \leq o_{\bar{a}} \leq v_c < o_{\bar{a}} + l_{\bar{a}} < v_1 < (o_a + l_a)$. However, by Equation 4.1, $o_{\bar{a}}$ can not fall within $[\mathrm{o_a}, \mathrm{o_a} + \mathrm{l_a})$. QED.

Conjecture: If all policies include private cache then short-term allocations share cache with at most two other settings.

Sketch of the proof: Since private caches are disjoint and shared cache must immediately precede or proceed private cache (due to contiguous allocation), it is not possible for two private caches to both appear after (or before) an allocation's private cache. One setting can share cache lines preceding a private cache allocation and another can share lines after the private cache.

Under contiguous allocation, 3 or more workloads can not share cache while also reserving private cache for baseline performance. This constrains the structure of cache sharing. First, cache contention emerges from pairwise interactions of collocated workloads. Second, the size of reserved and shared cache regions can affect performance. Also, we observe that the mapping of service components to allocation settings affects performance. With Intel Cache Allocation Technology, multiple OS processes and threads can map to one allocation setting. In this context, private cache allocation ensures baselines performance in aggregate for all processes mapped to the setting. Sharing cache in this way is

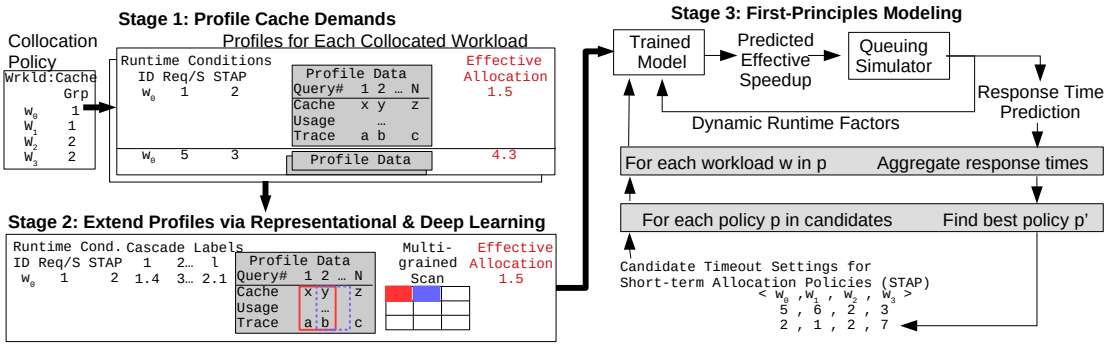

Fig. 4.2: In stage 1, our approach collects cache usage data from each collocated workload and measures effective cache allocation. In stage 2, profile data is used to train deep-learning models. Finally, stage 3 models response time and explores policies.

also relevant to non-contiguous cache allocation because multiple workloads interact with shared cache.

4.3 Design

It is hard to set timeout values for short-term allocation in collocated settings. Long timeout settings decrease the frequency of short-term allocation and may reduce speedup for each query. But, short timeout settings trigger short-term allocation more often, potentially slowing down collocated workloads that share the cache lines used for short-term allocation. In this paper, we present a model-driven approach to find a vector of timeouts (one for each workload) that provides low response time for all collocated workloads. We seek to characterize the speedup achieved by our approach compared to (1) baseline performance, (2) static and dynamic cache partitioning approaches and (3) competing timeout settings for short-term allocation.

Figure 4.2 outlines our modeling approach. Stages 1 and 2 collect profile data, employ deep and representational learning techniques and train a model that characterizes effective

cache allocation across policies. A key design challenge is to extract a large number of features (i.e., multi-dimensional inputs) from online query executions. Performance counters that capture cache usage data can produce training data with large input dimensions d but profiling runs n is constrained in collocated settings (*i.e., overhead*), especially if workloads are collocated for only short periods of time. Stage 3 integrates effective cache allocation into discrete queuing theory simulation[2] to predict response time. Effective cache allocation is a key intermediate metric that: (1) can be learned using small n and (2) integrates with first principles models straightforwardly.

4.3.1 Stage 1: Profiling Cache Demands

As shown in Figure 4.2, our approach runs online services in a test environment, captures cache usage during each query execution and computes the slowdown caused by collocation. In the test environment, we can control static runtime conditions, e.g., query arrival rate, short-term allocation policies, query mix and workload type. Dynamic runtime conditions, e.g., queue length, can not be controlled directly. Given runtime conditions, a profiling run uses lightweight architectural performance counters to trace cache usage during query execution. Depicted by the light gray box in Figure 4.2, our profiler samples counters 12–60 times per minute during each query execution. We fill zero values to pad traces and ensure profiles are equally sized.

Our design presumes a mechanism for request containers, wherein operating systems track query executions across context. Early approaches relied on applications to label

[2]Our queuing theory simulator extends a G/G/k model by adjusting the service rate based on a short-term allocation policy (i.e. we use a timeout and a resource budget to manage speedups.).

queries before and after system calls [6], gaining precise accounting at the cost of programming overhead. Recent work tracked query executions entirely in the operating system, but relied on heuristics to transfer context during network activities. Our approach samples architectural performance counters, a user-level process, making either approach applicable.

We flatten cache usage for each query, making a long $1xK$ vector, comprising the following sub-components:

$$\text{P} = ¡\overrightarrow{\text{static}}, \overrightarrow{\text{dynamic}}, \overrightarrow{\text{query}_0}, ..., \overrightarrow{\text{query}_N}, \text{eff. allocation } ¿ \quad s.t. |P| = K \tag{4.2}$$

Effective Cache Allocation: It is well known that cache allocation can speed up workload execution and that cache contention can cause slowdown in collocated settings. Time series analysis can reveal coincident spikes in last-level cache (LLC) accesses, i.e., contention. However, in practice, the effects of cache contention vary greatly. The presence of contention alone is insufficient to predict response time.

As a result, using contention alone to trigger short-term cache allocation may not provide much speedup. For example, collocations between memory-bound workloads can tolerate larger spikes in LLC misses. Likewise, under low arrival rates, response time is less sensitive to contention. Even though the effects of cache contention can be explained intuitively, it is hard to find the exact runtime conditions where short-term allocation can provide speedup. Dynamic factors, e.g., queuing delay, also affect the point in execution where short-term cache is allocated. The time a query spends queuing can trigger the SLO warning before execution or have a combined effect with service time that triggers it during execution.

Effective cache allocation (EA) is the ratio of (1) speedup from a short-term allocation policy (STAP) and (2) increased resource allocation during short-term allocations, Equation 4.3. Here, servicetime reflects average processing time for query execution under short-term allocation settings and timeout settings.

$$\text{EA} = \left(\frac{\text{servicetime}\left(\text{W}_{(\text{a},\text{a}',\text{t})}\right)}{servicetime\left(W_{(a,a,0)}\right)} \right) / \left(\frac{l_{a'}}{l_a} \right) \tag{4.3}$$

Effective allocation varies depending on: (1) the amount of cache allocated, (2) the frequency of short-term allocation requests and (3) contention from collocated executions. Heavy cache contention drags effective allocation below 1, whereas low contention and high data reuse produce values close to 1.

As shown in Figure 4.2, effective cache allocation aggregates response time for all queries under a tested runtime condition. However, dynamic cache allocation has temporary effects that can be amortized over long runs. Our profiling runs capture dynamic runtime conditions during execution, allowing us to split long running tests into multiple smaller measurements of effective cache allocation. This increases the number of rows (N) in our profile data.

4.3.2 Stage 2: Deep Learning

Our profiling approach yields feature-rich images of collocated online services. The effects of short-term allocation and cache contention are represented but hidden within these profiles. Representational and deep learning techniques, widely used in artificial intelligence, enhance multi-dimensional data by adding features that capture non-linear patterns in the data.

Deep and representational learning improve many machine learning approaches, from neural networks to support vector machines to decision trees. The design of our model can be implemented with any underlying learning approach. Still, the proceeding discussion may be influenced by our implementation based on deep forest [120].

Deep Learning: Machine learning uses historical (training) data to map input to a target output. For this paper, input are runtime conditions and cache usage demands and target output is effective cache allocation. With multi-dimensional data, machine learning struggles to find accurate mappings on training data that do not over fit test data. Deep learning addresses this challenge by first mapping input to concepts, i.e., groups of input with similar output attributes, and then mapping input and concepts to target output [62].

Figure 4.3 illustrates deep learning on our profile data. The input data comprises 3 features (query arrival rate, timeout and last-level cache misses). Machine learning approaches bound by these features look for settings where all matching input data exhibit anomalous effective allocation. To avoid over fitting, at least 2 observed executions are required to label a group of settings. The dotted lines surround settings that may be labeled anomalous by classic machine learning. No setting achieves over 50% accuracy.

Deep learning first learns concepts. For example, the result of the machine learning described above can be considered a concept. Concepts combine input data that seem far away in the initial feature space, but produce similar outcomes (e.g., anomalous EA). Using the concept as a feature, the deep learning approach can avoid over fitting and improve accuracy.

Representational Learning: Convolutional neural networks have become the standard bearer in computer vision. Representational learning, implemented via convolutions, underlies their success. A convolution computes a kernel, i.e., a function defined over a set

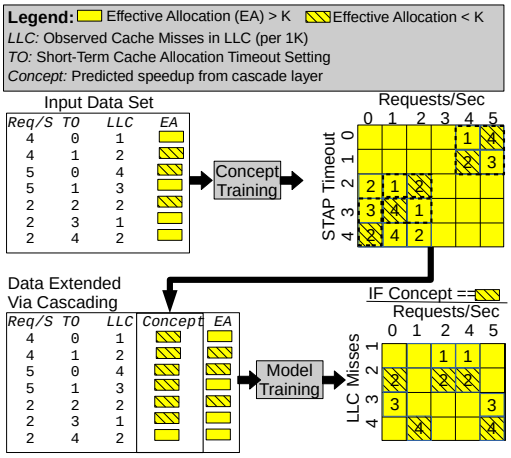

Fig. 4.3: Deep learning uncovers concepts that reveal rich patterns and avoid over fitting.

of features. These kernels produce new features for machine learning. In computer vision, convolutions can capture the presence of simple object components, e.g., handles, eyes or logos. In our context, convolutions capture correlated events that impact effective cache allocation, e.g., an L1 miss and an LLC cache access or multiple LLC miss events happening to collocated executions.

Figure 4.4 depicts representational learning. We define a square window size of 5x5 for the kernel. We apply the kernel to profile data from 20 query executions where cache usage was sampled 29 times per query. We slide the window across the whole data, computing the kernel in each window. This yields 400 new features that can augment cache usage data.

A convolution extracts spatial-temporal information from features. The spatial relationship between features affects the effectiveness of the convolutional process. Structuring features such that highly correlated features are close to each other in the data can increase

the number of patterns extracted by representational learning. Our evaluation will compare the impact of ordering features for spatial representation.

Simple Modeling Approaches Don't Work: Deep and representational features are transformations of data collected during profiling. A reader may ask why these transformations are important. Can a non-linear but simple modeling approach, e.g., decision trees, replace Stage 2 in our approach (Figure 4.2)?

Simple modeling approaches are prone to over fit for effective cache allocation, because these approaches tie concepts to rigid, inflexible representations. Consider a concept that captures the availability of short-term cache resources. Key features would include arrival rate, service time and timeout for each collocated workload. Simple models use hard parameters to represent the concept, over fitting when hidden factors like micro-service queuing delays affect response time. In contrast, deep learning approaches learn multiple representations of each concept and prioritize the most robust representations. On training data, deep learning representations may be less accurate than a simple model, but they generalize well, out performing under rigorous K-fold cross validation schemes.

In the evaluation section, we will show that simple modeling approaches are much less accurate in predicting response time and yield allocation policies that perform worse than our approach. We will also show that deep learning concepts can provide useful system insights.

4.3.3 Stage 3: First Principles Modeling

We consider a system that uses a queue to store incoming requests. Requests are removed from the queue and executed using a set of compute resources. First principles queuing theory enables the modeling of distributions for queuing delay and response time.

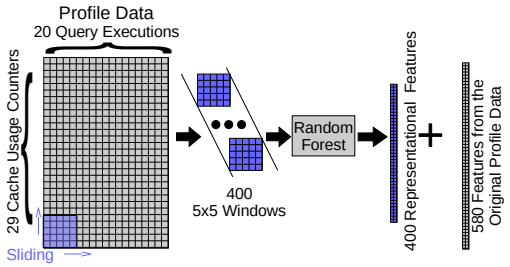

Fig. 4.4: Representational learning for spatial relationships.

This modeling is possible by representing the queuing problem as a Markovian process. A Markov process assumes past events and future events are independent. Short-term cache allocation breaks this fundamental assumption by creating interdependence between the queuing delay and service rate. Traditional closed-form queuing models exploit interdependence to compute average queue length and response time. However, enforcing similar assumptions for short-term cache allocation produces diverging results between the empirical data and the model output.

We overcome this limitation with a discrete event simulator that models the speed up afforded by short-term allocation. The simulator accepts the workload conditions, the cache-allocation policy, queuing delay, and the effective cache allocation as input and generates a simulation trace, i.e., processing needs, processing rate, short-term allocation processing rate and arrival time. This structure pairs with an internal structure that captures simulated events relative to the query, e.g., its current execution state, allocated cache, etc. Our full implementation jumps multiple steps at time to the next execution event affecting a query in the trace. When a query begins processing, the time waiting in the system (current time minus arrival time) is checked at each step and compared to the response time warning. A

total time exceeding the response time warning triggers a speed up for the remaining execution (i.e., short-term allocation processing rate). The simulator stops once a predetermined number of queries complete. The response time for each query is computed from execution events and the instantaneous queuing delay is outputted as dynamic condition feedback for future simulations.

4.4 Implementation

Our profiling system comprises software for runtime condition sampling, workload generation, short-term allocation and query execution with performance counter tracking.Our first implementation used uniform random sampling without replacement to assign settings. However, random sampling over sampled some settings. With limited time for profiling, we turned to stratified sampling to cover a wide range of representative samples. Specifically, our implementation randomly selected experiment settings as seeds. After executing seed experiments, we clustered them according to effective cache allocation and computed the centroid settings for each cluster. We created random settings near each centroid, executed them and updated cluster centroids. In our tests, stratified sampling reduced profiling time by 67%.

A workload generator sent runtime conditions, collocation settings, and the cache-allocation policy to our management software. Collocated executions ran in Docker containers bound to specific processor cores and ports. Configuration settings also defined cache lines for default allocation, cache lines for short-term allocation and the timeout settings for each collocated execution. The workload manager also sent queries at the configured arrival rate. We implemented a proxy service for each collocated execution. Proxy services queued queries waiting to access CPU resources.

Intel CAT was used to dynamically assign cache lines to process ids represented by Docker images. During configuration, we defined two service classes on each processor core: default allocation and a short-term allocation (see Figure 4.1). The proxy service monitored the response time of each outstanding query. When the STAP timeout was triggered, the class of service was switched. To keep the overhead low, if multiple queries were

outstanding for the same online service, all had access to short-term cache. Upon completion of the targeted query, i.e., a reply was received by proxy, the service class switched back to default.

We collected architectural performance counters related to cache usage throughout query execution. Counters were sampled as frequently as once per second by process id. The proxy service further differentiated counters by query execution. Cache usage counters, response time and computed effective cache allocation were captured as profile data.

4.4.1 Deep Forest

Deep and representational learning can be implemented in many machine learning algorithms. We used deep forests [120]. Deep forests use an approach called cascading to implement deep-learning atop random forests. Representational learning in deep forest uses random-forest kernels with sliding window inputs, an approach called multi-grain scanning. In this section, we describe our application of deep forest for modeling effective cache allocation from profile data.

Multi-Grain Scanning (MGS): Figure 4.5 depicts inference. First, cache usage profile data is transformed to spatially correlated representational features. Each feature predicts effective cache allocation. A random forest implements a convolutional kernel, mapping the window to a predicted value. Before inference, deep forests train the random forest. Sliding windows are computed and paired with corresponding effective cache allocation. Classic machine learning algorithms for random forests and decision trees are then used to create the forest.

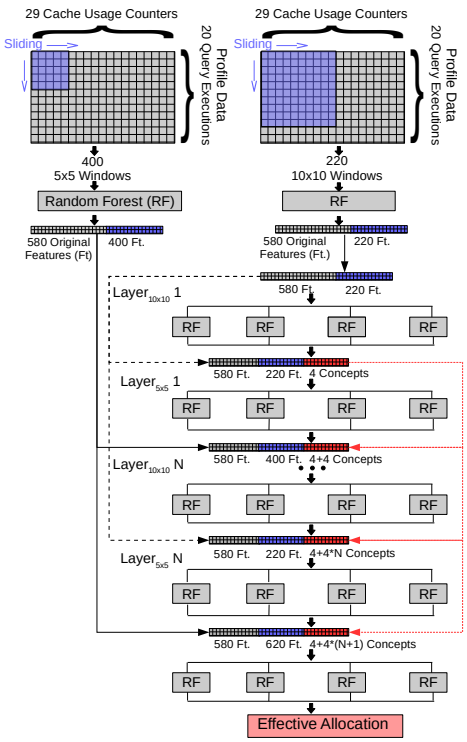

Fig. 4.5: Multi-grained scanning and cascading support deep and representational learning in deep forests.

As seen in Figure 4.5, sliding windows are used to scan the raw features. Suppose there is an input matrix of size 29 features x 20 query executions and a window size of 5x5. Sliding the window for 1 feature across spatial-temporal data produces a 5x5-dimensional feature matrix. A complete scan generates 400 (25x16) 5x5-dimensional feature matrices. Multiple sliding windows can be used to extract different details from the features. Figure 4.5 shows 2 sliding windows scanning the raw data. The instances generated by 1 sliding window are inputs to a random forest. Each instance has a corresponding predicted value

that is concatenated to the transformed feature vector. The new representation of the features is propagated to each layer in the cascaded structure. Instances generated by other window sizes are transformed using an identical process but with a different random forest.

Deep Forest Cascades: Cascade Modeling is a form of deep learning. Deep learning learns complicated functions by building representations that are expressed in terms of simpler representations. This layering of representations is known as cascades. Cascade modeling automatically learns representations at different levels of abstraction (e.g., colors-¿edges-¿objects) which allow a model to directly map the input to the output of a complex function without depending on human-crafted features. Each cascade is an ensemble of learners that specializes in identifying certain patterns in the input. The output from one cascade acts as additional information for the next ensemble of learners.

Deep forest employs a cascade structure where each level of cascade is an ensemble of decision forests. Feature information processed by a cascade and the transformed features are passed to the next cascade for further processing. Different type of forests are used to encourage diversity. Diversity is important to ensemble models to avoid over fitting. Each forest within a cascade level may contain 100s of trees. Some forests have random trees while others have completely random trees. Each completely-random forest contain 100s of completely-random trees, generated by randomly selecting a feature at each node for split. Trees are grown until all leaves are pure, that is leaves contain 1 value for regression or the same class for classification. Each random forest also contains 100s of trees. A tree is generated by randomly selecting $\sqrt{f}$ (where f is the number of input features) features with the best gini value for split. This process is repeated for each node until pure leaves are obtained.

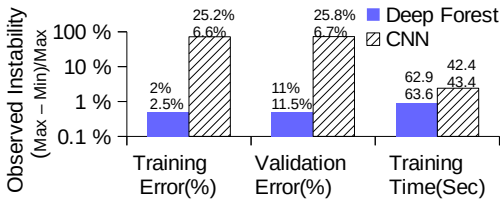

Fig. 4.6: Random variation affects training accuracy, validation accuracy and training time for deep forests and CNNs. Numbers atop each bar reflect min and max.

Each forest produces an estimate distribution function learned from training samples. In Figure 4.5, a 29x20 feature sample is transformed to a 400x1 feature sample. The first cascade level receives the extended feature data containing 580 original features plus 400 transformed features. The number of sliding windows determine how many layers are in a cascade level. For example, using 2 sliding windows produces 2 separate feature vectors that are passed in at different points in a cascade level. The first layer uses the samples with 580 original features plus 220 transformed features to make inferences. The outputs from that layer are combined with the original samples and passed to the next layer. The second layer has samples with 980 (580 + 400) features plus 4 concepts, where the additional 4 come from the 4 random forests from the first layer, and makes additional inferences. The output from the second layer is 4 more concepts added onto samples from the second sliding window. The final output from the first cascade level is 980 (580 + 400) features plus 8 concepts, which is passed to the second level. This process is repeated for N levels. The output from the cascade structure is passed to 4 more random forests where their results are averaged to give the effective cache allocation.

Choosing Between Deep Forest and CNN: CNNs are widely used for deep and representational learning. There is a wide range of tools that simplify their use. However, CNNs are

subject to random variation, especially on relatively small datasets and if hyper parameters are not known. Through back propogation, neural networks overwrite prior weights during the learning process which causes variability in accuracy and training time. In contrast, deep forests are trained layer by layer, i.e., each layer appends to the results of prior layer. As reported in Figure 4.6, we trained and evaluated CNNs and deep forests on profiled data 100 times. For the CNNs, we used TUNE [65] to find good hyper parameters. The best training results for neural networks can outperform deep forests, but deep forests reliably provide low error. The worst training results for neural networks can be twice as inaccurate as deep forests. We chose deep forests for their stability. Note, our evaluation compared only traditional CNNs. In future work, we will explore the reliability and accuracy trade-off with more complicated neural network structures, e.g., residual and long short-term memory (LSTM) networks.

Query Execution Workloads		
Wrk ID	Description	Cache Access Pattern
Jacobi	Solves the Helmholtz equation	Memory intensive Moderate cache misses
KNN	K-neaest neighbors	High data reuse Low cache misses
Kmeans	cluster analysis in data mining	High data reuse Low cache misses
Spkmeans	Spark cluster analysis	Higher cache misses b/c of tasks execution
Spstream	Spark extract words from stream	I/O intensive High cache misses
BFS	Breadth-first-search	Limited data reuse Moderate cache misses
Social	Social network implemented with loosely-coupled microservices	Moderate data reuse Moderate cache misses
Redis	YCSB: Session store recording recent actions	Low data reuse High Cache misses

Table 4.1: Benchmarks used in our experiments.

Static Runtime Conditions for Each Online Service	
Description	Supported Settings
Collocated services sharing cache lines	Jacobi, NN, Kmeans, Spkmeans, Spstream, BFS, Social or Redis
Query inter-arrival rate (rel. to service time)	25% – 95%
Timeout policy (rel. to service time)	0% (always use shared cache) – 600% (never use short-term allocation)
Cache usage sampling	1 Hz – every 5 seconds

Table 4.2: Runtime conditions studied.

4.5 Evaluation

Table 4.1 shows the micro- and macro-benchmarks used in our evaluation. Collectively, these benchmarks have diverse cache usage profiles, computational demands, parallelism, and software composition. We ran experiments on an Intel Xeon E5-2683 processor with 16 cores, 40 MB of last-level cache and 64 GB main memory. For baseline performance, we provisioned 2 cores and 2 MB LLC cache.

Social [42]: This realistic macro-benchmark composes 36 microservices running in 30 Docker containers and mimics the behavior of a social networking site where users are

composing and posting messages. The service supports up to 2000 requests per second. Baseline response time is 7.5 ms. All microservices in Social shared one short-term cache allocation policy. As such, we use social to study the effect of 36 concurrent processes sharing cache.

Spark Spkmeans and Spstream [106]: These benchmarks use the Apache Spark platform for parallel data processing. They execute 16 concurrent threads. For Spkmeans, these threads partition cluster assignment in the k-means algorithm. For Spstream, these threads execute windowed word count. Like Social, all threads share allocation settings. The k-means algorithm reuses cached data more often than windowed word count. The Spark executor was configured with 1 thread which managed worker threads. The worker received text from one raw network-stream that generated data at 10 MB/s. Baseline response time for Spkmeans was 81 seconds. For Spstream, baseline response time was 1 second.

Redis: We used the YCSB benchmark suite to generate a realistic trace for Redis, a widely used key-value store. Under this workload, Redis exhibits low data reuse and high cache misses. There were 200,000 records comprising of 1KB of data. The baseline response time for Redis is 1 ms per query.

Rodinia [17]: The Rodinia benchmarks used OpenMP, a shared memory multi-processing API, to create parallel threads for Jacobi, KNN, Kmeans, and BFS. These benchmarks were configured to run with 16 OMP threads. These benchmarks capture computational demands in HPC environments. Note, Rodinia also includes a implementation of the k-means algorithm that does not use the Spark platform. KNN and Kmeans exhibited high cache hit rates. Jacobi and BFS are memory-intensive workloads with moderate cache miss rates.

For each experiment, we fully utilized processor cores by collocating 8 concurrent services and allowing each service to use 2 cores and 2 MB LLC for baseline performance. Recall, Intel CAT requires *contiguous cache allocation*. Proxy service scripts configured pairwise shared cache lines. For example, if Jacobi is collocated with BFS, Jacobi could reserve private cache lines #1 & #2 and BFS could reserve cache lines #5 & #6. During short-term allocation, query executions for either or both services could use cache lines 3 & 4 in addition to their private cache. We defined query inter-arrival rate for each online service relative to its average service time. Short-term allocation timeout was also defined relative to the average service time. If the timeout was 150% and the average service time was 100 seconds, the short-term allocation would trigger at 150 seconds.

We sampled L1 data cache stores and misses; L1 instruction cache stores and misses; L2 requests, stores and misses; LLC loads, misses, stores; and other architectural counters related to cache usage (29 in total). We used the official Deep Forest implementation [39]. Our Deep Forest contained 4 cascade layers with each layer hosting 4 random forests. Each random forest was configured to have 100 estimators. The MGS component consisted of 4 sliding windows with window sizes 5x5, 10x10, 15x15, and 35x35. We used 1 random forest per window with 50 estimators each.

We report two types of experiments. First, we investigate the accuracy of our modeling approach. For given runtime conditions, we executed online services and measured average and 95^{th} percentile response time. We compare the average response time against the prediction from our modeling approach using the same runtime conditions. Note, that our modeling approach could *not* use an observed profile from the runtime condition to train the deep forest. We also compare our approach to competing modeling approaches using the same methodology. The second type of experiment calibrated our model with training

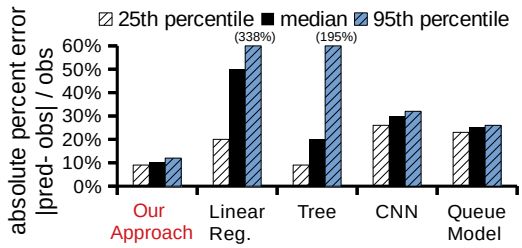

Fig. 4.7: Accuracy of response time predictions for our approach, simple models, CNN and a queuing simulator.

data and then the computed expected response time for a wide range of randomly sampled conditions. We examined the performance gains from having a model available.

4.5.1 Model Accuracy

We profiled 14,220 runtime conditions that included every pairwise collocation of our benchmarks and a wide range of runtime conditions and timeout settings. Profile data was separated into distinct training and testing sets. Testing data was not used during training to ensure models accurately extrapolated to new, unseen conditions.

For our model, testing data outnumbered training data by 2 to 1, i.e., 33% training data and 66% testing data. For competing models, we used 70% training data and 30% testing data. We placed our approach at a disadvantage to ensure low profiling overhead. Later in this section, we evaluate accuracy of our approach under high profiling overhead.

Accurate Response Time Prediction: In Figure 4.7, we used absolute percent error (accuracy) to compare our modeling approach against competing approaches. Our full modeling approach achieved 11% median error and 12% error at the 95^{th} percentile.

Figure 4.7 arranges competing approaches from simple to complex. First, we compared to a linear regression model. As expected, this approach produced median error of 50%

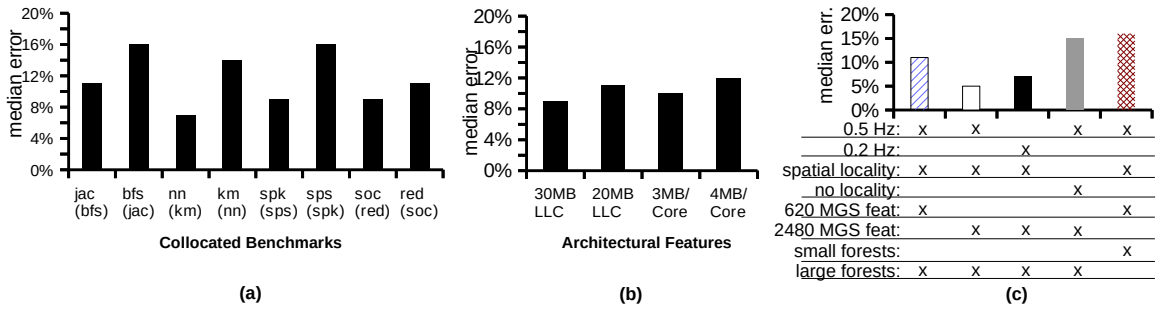

Fig. 4.8: (a) Accuracy of response time predictions for specific collocations. (b) Accuracy across processor cache sizes. (c) Evaluation of multi-grained scanning parameters (sampling rate, spatial locality, window size and forest size).

and the 95^{th} percentile error was greater than 300%. A decision tree model achieved 20% median error and 95^{th} percentile error above 100%.

Recall, our approach combines deep and representational learning with first-principles queuing theory. In contrast, the CNN approach reported in Figure 4.7 uses only deep and representational learning to map directly from runtime conditions to response time. We used PyTorch to train a CNN. Unlike our model that is calibrated using only one collocation pairing, the CNN had access to all training data. Further, unlike our deep forest, the CNN had many hyper parameters affecting accuracy. We used TUNE [65] to explore the following hyper parameters: epoch, batch size, learning rate, number of neurons and drop rate. The best setting, which we reported in Figure 4.7, achieved 26% median error. The Queuing Model approach used only our queuing simulator as described in section 4.3.3. This approach had 23% median error.

Generalization: Figure 4.8(a) details our model's error for each workload in Table 4.1. Collocated workloads are listed in parenthesis. To be sure, the targeted collocation settings are not included in training for test. For example, the label jac(bfs) is the median

error for predicting response time for Jacobi with BFS collocated, and bfs(jac) is the opposite meaning. Our model predicted average response time with median error below 15%. In Figure 4.8(b), we tested generalization across processor architectures. In addition to our default platform (Xeon E5-2683), we ran experiments on an Intel Xeon 2650 (30 MB LLC) and on a Xeon 2620 (20 MB LLC). In all cases, we fully utilized processor cores by running workloads concurrently. We also changed collocation settings by allowing each workload to reserve 3 MB and 4 MB of LLC respectively. In all cases, our median error for response time prediction remained below 15%.

Profiling Time: We also studied profiling time and model accuracy. For collocated Apache Spark workloads, profiling cache usage under a short-term allocation policy took 3 minutes. We profiled 3 collocations in parallel. Our full model profiled workloads for 30 minutes and acquired roughly 100 profiles for training and validation. However, longer profiling time provided additional data and improved results. We observed that with 2.5 hours for profiling, the median error fell to 8.6%. In general our approach was robust to reduced profiling time because (1) the use of first-principles queuing simulation bounded model error and (2) stratified sampling improved accuracy given limited samples. Our approach can profile collocations briefly and still yield predictions that perform better than other modeling techniques, as shown shown in Figure 4.7.

Implementation of Multi-Grain Scanning Strategy: Multi-grained scanning (MGS) learns representational features from the cache usage trace. In Figure 4.8(c), we studied (1) the organization of performance counters in the cache usage trace, (2) the MGS window size and (3) the sampling rate for cache usage data. Recall, multi-grained sampling exploits spatial locality. In computer vision, spatial locality is inherent. However, for effective cache allocation, performance counters may not be organized to exhibit spatial locality. Figure 4.8(c)

compared two approaches to order cache usage traces. The first ordering randomly shuffled performance counters, removing locality. The second ordering grouped counters by type (spatial locality). For example, L1 load misses and L3 load misses for collocated service A appeared close to each other in the cache usage trace whereas L1 store misses and L3 store misses for service B were a separate group. We also studied the effect of changing the window size to quadruple the number of MGS features from 620 to 2480. We also compared performance counter sampling rates of 0.5 Hz and 0.2 Hz. Finally, we also studied the impact of the model size, defined as the number of estimators used in the deep forest model. Estimators constrain the number of features included for representational and deep learning.

Figure 4.8(c) shows median error in the predicted response time across MGS settings. The categories at the bottom of the figure represent multi-grain settings used during response time prediction. The "X" directly below a bar plot indicates the settings applied to achieve the corresponding response time error. Only 4 settings can be used for any bar, i.e., 1 setting per category. The response time error for our model incurred a 2% increase by collecting cache-counters at 1 sample every 5 seconds compared to every 2 seconds. Our model error increased from 5% to 15%, if an spatial ordering among the cache-counters was removed. A 4X decrease in window size doubled response time error, but also reduced training time significantly. Lastly, we observed that using too few estimators (small forests) yielded accuracy comparable to the Queue Model approach.

4.5.2 Managing Short-Term Allocation

We used short-term cache allocation to speed up slow query executions. The short-term allocation policy (STAP) set a timeout defined using Equation 4.4.

$$\frac{\text{response time}}{\text{exp. service time}} > T \tag{4.4}$$

Here, service time normalizes the timeout across workloads. Our model-driven approach allowed us to explore settings for T under given collocation and runtime conditions. We explored 25 settings for T for each pair of cache-sharing collocated workloads, i.e., 5 independent settings per workload. We set the query arrival rate to 90% of each workload's service time. Query inter-arrival times were exponential.

Recall, we seek a vector representing settings of T for each collocated workload. To balance performance for each workload, we implemented a simple SLO-driven matching policy. Step 1: we searched for settings of T where the response time was within 5% of the lowest response time found across all settings for a target service. Step 2: we chose policies that intersected both collocated services.

In Figure 4.9, we reported speedup from our model-driven approach against competing cache allocation approaches across multiple collocations. The competing allocation approaches are described below:

1. **No cache sharing**: Each workload has access to only its private cache (i.e., baseline performance). In Figure 4.9, all results are normalized to this default approach.
2. **Static allocation**: Services can (1) share cache lines fully or (2) use only private cache—whichever yields best performance.

3. **Workload-aware allocation (dCat)**: Shared cache is allocated to the workload that achieves the greatest speedup (i.e., throughput profiling with fixed workload phases). Other collocated services use private cache to reduce contention [108].

4. **Dynamic-allocation based on IPC (dynaSprint)**: Timeout T is used to allocate shared cache for maximum performance, like our model-driven approach. However, the settings found under low arrival rate are reused (ignoring queuing delay) under high arrival rate.

5. **Dynamic-allocation based on simple ML models:** In this approach, we hide deep and representational features from our full model, eliding stage 2. Here, settings for timeout T represent error caused by using inaccurate, simple decision-tree models (see Figure 4.7).

In Figure 4.9, we reported speedup in 95^{th} percentile response time across 6 collocation settings that include Redis, Spark, Rodinia and micro-service workloads. Compared to the default setting, our model-driven approach achieved median speedup of 2X, speeding up the Spark Kmeans workload by up to 2.6X. Compared to state of the art approaches (dCat and dynaSprint), our approach achieved speedup of 1.3X while speeding up the micro-service social networking site and Redis by 1.38X and 1.4X respectively. Under heavy arrival rate, Redis has low effective cache allocation with micro-services in Social. dynaSprint fails to capture increased variability in Social, leading to a poor timeout setting. Further, Redis benefits greatly from additional cache lines. dCat allocates additional cache lines to Redis to achieve a high speedup but does not speed up social. Our approach sets a low timeout for Redis and moderate timeout for Social. This finds an excellent balance by speeding up Redis but affording short-term allocation when Social suffers from high queuing delay.

In Figure 4.9, we also compared our full approach to an approach based on simple models. Even though simple models yield greater absolute percentage error, the simple-model approach produced comparable response time in 3 of the 6 collocation settings.

Deriving Useful System Insights: Figure 4.10 uses non-linear, dimensionality reduction (t-distributed stochastic neighbor embedding-TSNE) to characterize response time across 5,000 collocation and runtime settings. Figure 4.10(a) uses data collected from our profiling run, i.e., it does not use deep learning concepts. Response time, intuitively, correlates with workload. However, we observed multiple settings where LLC cache misses were comparable (within 5%) but response time differed by 2X or more. Dimensionality reduction expected similar response time in such cases (see points marked by *X*). Figure 4.10(b) uses deep learning concepts for dimensionality reduction, we observed such settings were separated. We observed that a key deep learning concept was weighted heavily: The availability of short-term cache resources. Underlying trees captured arrival rate, service rate and timeout. While inaccurate by themselves, combined with representational data on L2 and LLC cache misses, the dimensionality reduction proved much more accurate. In general, analysis with and without deep learning features can reveal important system traits that simple models alone cannot.

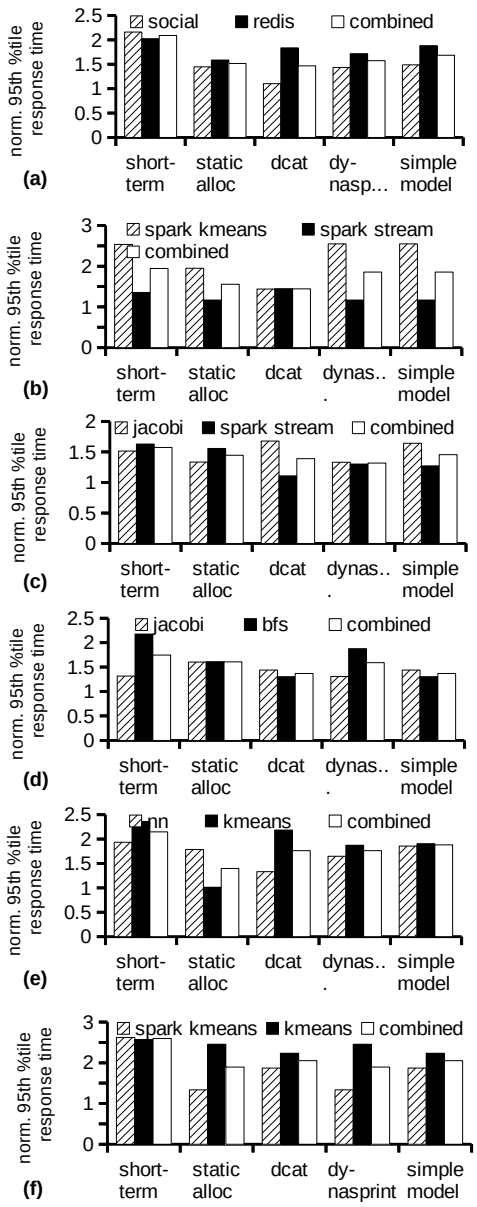

Fig. 4.9: Comparing speedup in 95th percentile response time for competing cache allocation techniques. Data normalized to response time under no-sharing, static allocation policy.

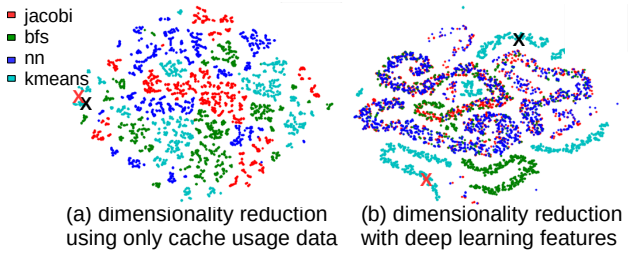

Fig. 4.10: Why we need representational learning.

4.6 Related Work

Cache allocation that considers workload needs can often improve performance compared to workload agnostic allocation. Allocation policies determine how much cache to allocate to each workload, especially when the LLC is shared. Prior work studied cache allocation to reduce interference [2, 74, 116], maximize throughput [14, 23, 22, 51, 121, 61] and improve tail latency [1, 2, 74, 22, 61]. Recently, dynamic cache allocation allows systems software to assign cache at runtime [53]. This section outlines work on cache management for online services and modeling approaches.

4.6.1 Cache Management for Online Services

Albonesi et al. [1] proposed the idea to disable select cache ways during periods of low demand and re-enable them during intense memory periods. Their goal was to reduce energy consumption with a small performance degradation. Our work uses a technique that increases performance for a query execution by accessing additional cache shared with collocated executions. We focus on reducing response time not reducing energy.

Xu et al. [108] presented a dynamic cache allocation approach for query executions sensitive to noisy neighbors. Their dynamic approach offered a strong cache isolation while maintaining a minimum performance bound. Our approach relaxes the cache isolation requirement during periods when query executions are suffering performance loss.

Chen et al. [20] proposed a resource manager that dynamically adjusts resources including cache for online services suffering from performance loss. While their solution finds good online policies, their solution cannot explore policies a priori. Our work can quickly explore collocation settings and policies online and offline after the profiling stage. In contrast, Chen et al. requires performance feedback during online operation.

Kulkarni et al. [61] proposed an online resource manager that uses machine learning to determine the performance and power for a core and cache configuration. They find the best configuration to reduce latency and increase throughput. Their work models collocated workloads sharing the same ways. Our work models collocated workloads that have private and shared cache ways.

4.6.2 Cache Modeling Approaches

First principles modeling can predict response time for a query's execution as a function of the arrival and service time distributions [60]. However, modeling the effects of cache on response time with first principles is challenging. The cache behavior is hardware dependent which changes from one processor to the next. Collocating executions further complicates this problem by introducing cache interference in the LLC [109, 38]. Prior works measure cache interference online [52, 67] and make decisions at an execution phase granularity while others characterize LLC performance [107] for each query execution before making decisions.

Qureshi et al. [78] implemented utility based cache allocation at the hardware level. This work ignores queuing delay since it is implemented below the software stack. Huang et al. [52] presented a runtime software that managed cache allocations dynamically by predicting cache-utility. Similarly, our work uses hardware performance counters to profile cache-utility. However, we rely on offline profiling to collect the data needed for training our model. We aggregate the effects of execution phases into a single factor, which enables us to model performance for collocated executions under low and high query arrival rates.

El-sayed et al. [35] proposed a cache-sharing hybrid approach. They grouped workloads into clusters and allocated the cache among these clusters [86]. Workloads were

clustered based on their cache-sharing compatibility. Our work allocates the same cache allocation to multiple services. Our effective cache allocation represents the speedup experienced by the collocated executions and has a similar purpose to their clustering.

4.7 Conclusion

Short-term cache allocation grants and then revokes access to processor cache lines dynamically. Online services can use short-term cache allocation to speed up queries as they execute, targeting queries likely to suffer high response time. However, when multiple services collocate by sharing cache, their query executions can contend for short-term allocation, causing recurring slowdowns that degrade response time. This paper presented a model-driven approach to choose cache allocation policies that yield low response time for collocated services. We used representational and deep learning techniques to extract hidden features that capture the effect of short-term cache allocation on response time, producing a novel performance modeling approach that accurately predicted average and 95^{th} percentile response time. We showed that our approach can uncover short-term cache allocation policies that yield 2.4X speedup.

Academic acknowledgements: The research presented in this thesis was conducted in the ReRout Lab at The Ohio State University under the supervision of Professor Christopher Stewart. The ideas and concepts studied here build upon years of research by prior lab members. I have cited many of those papers in the bibliographic references to acknowledge their impact.

Bibliography

[1] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–259, Nov 1999.

[2] David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, and et al. Dynamically tuning processor resources with adaptive processing. *Computer*, 36(12):4958, December 2003.

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Cedule: A scheduling framework for burstable performance in cloud computing. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*, pages 141–150. IEEE, 2018.

[4] Danilo Ardagna, Giuliano Casale, Michele Ciavotta, Juan F Pérez, and Weikun Wang. Quality-of-service in cloud computing: modeling techniques and their applications. *Journal of Internet Services and Applications*, 2014.

[5] Peter E. Bailey, Aniruddha Marathe, David K. Lowenthal, Barry Rountree, and Martin Schulz. Finding the limits of power-constrained application performance. In *SC*, pages 79:1–79:12, 2015.

[6] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, 1999.

[7] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[8] Chloe M Barnes, Kirstie Bellman, Jean Botev, Ada Diaconescu, Lukas Esterle, Christian Gruhl, Christopher Landauer, Peter R Lewis, Phyllis R Nelson, Anthony Stein, et al. Chariot-towards a continuous high-level adaptive runtime integration testbed. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 52–55. IEEE, 2019.

[9] Jeff Barr. `https://aws.amazon.com/blogs/aws/low-cost-burstable-ec2-instances/`, 2014.

[10] Jayson Boubin, Naveen Babu, Christopher Stewart, John Chumley, and Shiqi Zhang. Managing edge resources for fully autonomous aerial systems. In *ACM Symposium on Edge Computing*, 2019.

[11] Jayson Boubin, Codi Burley, Peida Han, Bowen Li, Barry Porter, and Christopher Stewart. Programming and deployment of autonomous swarms using multi-agent reinforcement learning. *arXiv preprint arXiv:2105.10605*, 2021.

[12] Jayson Boubin, John Chumley, Christopher Stewart, and Sami Khanal. Autonomic computing challenges in fully autonomous precision agriculture. In *IEEE International Conference on Autonomic Computing*, 2019.

[13] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[14] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. Optimal cache partition-sharing. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*, ICPP 15, page 749758, USA, 2015. IEEE Computer Society.

[15] Aniket Chakrabarti, Srinivasan Parthasarathy, and Christopher Stewart. A pareto framework for data analytics on heterogeneous systems: Implications for green energy usage and performance. In *IEEE International Conference on Parallel Processing*, 2017.

[16] Aniket Chakrabarti, Christopher Stewart, and Srini Parthasarathy. Green- and heterogeneity-aware partitioning for data analytics. In *IEEE International Workshop on Green and Sustainable Networking and Computing*, 2016.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[18] Lydia Y. Chen and Natarajan Gautam. Server frequency control using markov decision processes. In *INFOCOM*, pages 2951–2955, 2009.

[19] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia 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.

[20] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *ASPLOS*, New York, NY, USA, 2019.

[21] David Chiu, Christopher Stewart, and Bart McManus. Electric grid balancing through low-cost workload migration. In *ACM GreenMetrics Workshop*, 2012.

[22] Intel Corporation. Intel vtune amplifier 2018 users guide. `https://software.intel.com/en-us/vtuneamplifier-help-hardware-event-skid`, 2018.

[23] Intel Corportation. Intel 64 and ia-32 architectures software developers manual, 2016.

[24] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*, October 2017.

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

[26] Datadog. 8 surprising facts about real docker adoption, Oct 2015.

[27] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *ACM SOCC*, 2014.

[28] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. Ubl: unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *IEEE ICAC*, 2012.

[29] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2), 2013.

[30] Nan Deng. Carbon-aware cloud application. In *IEEE Network Operations and Management Symposium*, 2012.

[31] Nan Deng, Christopher Stewart, Jaimie Kelley, Daniel Gmach, and Martin Arlitt. Adaptive green hosting. In *IEEE International Conference on Autonomic Computing*, 2012.

[32] Nan Deng, Christopher Stewart, and Jing Li. Concentrating renewable energy in grid-tied datacenters. In *IEEE International Symposium on Sustainable Systems Technology*, 2011.

[33] Peter Derosa, Kai Shen, Christopher Stewart, and Jonathon Pearson. Realism and simplicity: Disk simulation for instructional os performance evaluation. In *ACM SIGCSE Technical Symposium*, 2006.

[34] Jianru Ding, Ruiqi Cao, Indrajeet Saravanan, Nathaniel Morris, and Christopher Stewart. Characterizing service level objectives for cloud services: Realities and myths. In *IEEE International Conference on Autonomic Computing*, 2019.

[35] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sanchez. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117, Feb 2018.

[36] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, 2012.

[37] Songchun Fan, Seyed Majid Zahedi, and Benjamin C. Lee. The computational sprinting game. In *ASPLOS*, pages 561–575, 2016.

[38] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53(2):4957, February 2010.

[39] K Fengji. gcForest Version 1.1.1. `https://github.com/kingfengji/gcForest`, 2018.

[40] Gianluca Fiori, Francesco Bonaccorso, Giuseppe Iannaccone, Tomás Palacios, Daniel Neumaier, Alan Seabaugh, Sanjay K Banerjee, and Luigi Colombo. Electronics based on two-dimensional materials. *Nature nanotechnology*, 9(10):768–779, 2014.

[41] 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.

[42] Gan, Yu et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. In *ASPLOS*, New York, NY, USA, 2019.

[43] Anshul Gandhi, Sherwin Doroudi, Mor Harchol-Balter, and Alan Scheller-Wolf. Exact analysis of the m/m/k/setup class of markov chains via recursive renewal reward. In *ACM SIGMETRICS*, 2013.

[44] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Adaptive, model-driven autoscaling for cloud applications. In *IEEE ICAC*, 2014.

[45] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytiä. Reducing latency via redundant requests: Exact analysis. In *Sigmetrics*, pages 347–360, 2015.

[46] Hamid Reza Ghasemi and Nam Sung 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.

[47] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ASPLOS*, 2015.

[48] Iñigo Goiri, Thu D. Nguyen, and Ricardo Bianchini. Coolair: Temperature- and variation-aware management for free-cooled datacenters. In *ASPLOS*, pages 253–265, 2015.

[49] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. *SIGPLAN Not.*, 50(4):161–175, March 2015.

[50] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *IEEE ICAC*, 2013.

[51] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas F. Wenisch, Jason Mars, Lingjia Tang, and Ronald G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *HPCA*, 2015.

[52] Ziqiang Huang, José A. Joao, Alejandro Rico, Andrew D. Hilton, and Benjamin C. Lee. Dynasprint: Microarchitectural sprints with dynamic utility and thermal management. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 52, page 426439, New York, NY, USA, 2019. Association for Computing Machinery.

[53] CAT Intel. Improving real-time performance by utilizing cache allocation technology. `https://01.org/cache-monitoring-technology`, 2015.

[54] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L. Cox. TPC: target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *ASPLOS*, 2016.

[55] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. Predictive parallelization: taming tail latencies in web search. In *SIGIR*, 2014.

[56] Jaimie Kelley and Christopher Stewart. Balanced and predictable networked storage. In *IEEE International Workshop on Data Center Performance*, 2013.

[57] Jaimie Kelley, Christopher Stewart, Nathaniel Morris, Devesh Tiwari, Yuxiong He, and Sameh Elnikety. Obtaining and managing answer quality for online data-intensive services. In *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 2017.

[58] Jaimie Kelley, Christopher Stewart, Devesh Tiwari, and Saurabh Gupta. Adaptive power profiling for many-core hpc architectures. In *IEEE International Conference on Autonomic Computing*, 2016.

[59] Jamie Kelley, Christopher Stewart, Devesh Tiwari, Yuxiong He, Sameh Elnikey, and Nathaniel Morris. Measuring and managing answer quality for online data-intensive services. In *IEEE International Conference on Autonomic Computing*, 2015.

[60] 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.

[61] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi. Cuttlesys: Data-driven resource management for interactive services on reconfigurable multicores. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 650–664, 2020.

[62] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[63] Kangwook Lee, Ramtin Pedarsani, and Kannan Ramchandran. On scheduling redundant requests with cancellation overheads. In *Allerton Conference*, 2015.

[64] Roy Levin, Ellis Cohen, William Corwin, Fred Pollack, and W Wulf. Policy/mechanism separation in hydra. In *ACM SIGOPS Operating Systems Review*, volume 9, 1975.

[65] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[66] Yanpei Liu, Stark C Draper, and Nam Sung Kim. Sleepscale: runtime joint speed scaling and sleep states management for power efficient data centers. In *ACM SIGARCH Computer Architecture News*, 2014.

[67] Amiya K. Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the 15th International Middleware Conference*, Middleware 14, page 277288, New York, NY, USA, 2014. Association for Computing Machinery.

[68] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 11(2):146–178, May 1993.

[69] D. Meisner, C. M. Sadler, L. A. Barroso, W. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 319–330, June 2011.

[70] 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.

[71] Nathaniel Morris, Siva Renganathan, Christopher Stewart, Robert Birke, and Lydia Chen. Sprint ability: How well does your software exploit bursts in processing capacity? In *IEEE International Conference on Autonomic Computing*, 2016.

[72] Nathaniel Morris, Christopher Stewart, Robert Birke, and Lydia Chen. Early work on modeling computational sprinting. In *ACM Symposium on Cloud Computing*, 2017.

[73] Nathaniel Morris, Christopher Stewart, Lydia Chen, Robert Birke, and Jaimie Kelley:. Model-driven computational sprinting. In *ACM European Conference on Computer Systems*, 2018.

[74] Srinivas Pandruvada. Intel running average power limit, 2014.

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

[76] Zhan Qiu, Juan F. Pérez, and Peter G. Harrison. Variability-aware request replication for latency curtailment. In *INFOCOM*, pages 1–9, 2016.

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

[78] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, page 423432, USA, 2006. IEEE Computer Society.

[79] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo 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.

[80] Rashmi Rao, Christopher Stewart, Arnuflo Perez, and Siva Renganathan. Assessing learning behavior and cognitive bias from web logs. In *IEEE Frontiers in Education*, 2018.

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

[82] Siva Renganathan, Christopher Stewart, Arnulfo Perez, Rashmi Rao, and Bailey Braaten. Preliminary results on an interactive learning tool for early algebra education. In *IEEE Frontiers in Education*, 2017.

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

[84] Jennie Rogers, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 337–348, 01 2011.

[85] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan, and Eliezer Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 2012.

[86] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gmez. Application clustering policies to address system fairness with intels cache allocation technology. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 194–205, 2017.

[87] Kai Shen, Christopher Stewart, Chuanpeng Li, and Xin Li. Reference-driven performance anomaly identification. In *ACM International Conference on Measurement and Modeling of Computer Systems*, 2009.

[88] Kai Shen, Alex Zhang, Terence Kelly, and Christopher Stewart. Operational analysis of processor speed scaling. In *Brief Announcement at the Symposium on Parallelism in Algorithms and Architectures*, 2008.

[89] Rahul Singh, David E Irwin, Prashant J Shenoy, and Kadangode K Ramakrishnan. Yank: Enabling green data centers to pull the plug. In *USENIX NSDI*, 2013.

[90] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14:473–530, 1982.

[91] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *IEEE International Conference on Autonomic Computing*, 2013.

[92] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. In *European Conference on Computer Systems*, 2007.

[93] Christopher Stewart, Matthew Leventi, and Kai Shen. Empirical examination of a collaborative web application. In *IEEE International Symposium on Workload Characterization (special session on Benchmark Innovation)*, 2008.

[94] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *Symposium on Networked Systems Design and Implementation*, 2005.

[95] Christopher Stewart and Kai Shen. Some joules are more precious than others: Managing renewable energy in the datacenter. In *ACM Workshop on Power Aware Computing and Systems*, 2009.

[96] Christopher Stewart, Kai Shen, Arun Iyengar, and Jian Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.

[97] G Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[98] Steven Swanson and Michael Bedford Taylor. Greendroid: Exploring the next evolution in smartphone application processors. *IEEE Communications Magazine*, 49(4):112–119, 2011.

[99] Andrei Tchernykh, Denis Trystram, Carlos Brizuela, and Isaac Scherson. Idle regulation in non-clairvoyant scheduling of parallel jobs. *Discrete Applied Mathematics*, 157(2):364 – 376, 2009.

[100] Eno Thereska and Gregory 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.

[101] 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.

[102] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS*, pages 205–218, 2010.

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

[104] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[105] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, Lydia Y. Chen, Robert Birke, and George Kesidis. Effective capacity modulation as an explicit control knob for public cloud profitability. In *ICAC*, pages 95–104, 2016.

[106] Yangjun Wang et al. Stream processing systems benchmark: Streambench. In *MS Thesis*, 2016.

[107] Carole-Jean Wu and Margaret Martonosi. Characterization and dynamic mitigation of intra-application cache interference. *(IEEE ISPASS) IEEE INTERNATIONAL SYMPOSIUM ON PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE*, pages 2–11, 2011.

[108] C Xu, K Rajamani, A Ferreira, W Felter, and et al. dcat: dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *ACM Eurosys*, 2018.

[109] Chi Xu, Xi Chen, Robert P. Dick, and Zhuoqing Morley Mao. Cache contention and application performance prediction for multi-core systems, 2010.

[110] Zichen Xu, Nan Deng, Christopher Stewart, and Xiaorui Wang. Cadre: Carbon-aware data replication for geo-diverse services. In *IEEE International Conference on Autonomic Computing*, 2015.

[111] Zichen Xu, Christopher Stewart, Nan Deng, and Xiaorui Wang. Blending on-demand and spot instances to lower costs for in-memory storage. In *IEEE International Conference on Computer Communications*, 2016.

[112] Zichen Xu, Christopher Stewart, and Jiacheng Huang. Elastic, geo-distributed raft. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–9. IEEE, 2019.

[113] Ming-Der Yang, Jayson G Boubin, Hui Ping Tsai, Hsin-Hung Tseng, Yu-Chun Hsu, and Christopher C Stewart. Adaptive autonomous uav scouting for rice lodging assessment using edge computing with deep learning edanet. *Computers and Electronics in Agriculture*, 179:105817, 2020.

[114] Seyed Majid Zahedi, Songchun Fan, Matthew Faw, Elijah Cole, and Benjamin C Lee. Computational sprinting: Architecture, dynamics, and strategies. *ACM Transactions on Computer Systems (TOCS)*, 34(4):12, 2017.

[115] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *ASPLOS*, pages 545–559, 2016.

[116] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102, 2009.

[117] Zichen Zhang, Jayson Boubin, Christopher Stewart, and Sami Khanal. Whole-field reinforcement learning: A fully autonomous aerial scouting method for precision agriculture. *Sensors*, 20(22):6585, 2020.

[118] Qin Zhao, David Koh, Syed Raza, Derek Bruening, and Weng-Fai Wong. Dynamic cache contention detection in multi-threaded applications. In *VEE 2011; Proceedings of the 7th ACM SIGPLAN/SIGOPS International conference on virtual execution environments*, pages 27–37, New York, NY, 2011.

[119] Wenli Zheng and Xiaorui 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.

[120] Zhi-Hua Zhou and Ji Feng. Deep forest. *National Science Review*, 6(1):74–86, 2019.

[121] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ACM Sigplan Notices*, 45(3), 2010.