Performance Modeling for Short-Term Cache Allocation

Nathaniel Morris
AMD & The Ohio State University
United States

Christopher Stewart
The Ohio State University
United States

Lydia Chen
TU Delft
Netherlands

Robert Birke
ABB Future Labs
Switzerland

ABSTRACT

Short-term cache allocation grants and then revokes access to processor cache lines dynamically. For online services, short-term allocation can speed up targeted query executions and free up cache lines reserved, but normally not needed, for performance. However, in collocated settings, short-term allocation can increase cache contention, slowing down collocated query executions. To offset slowdowns, collocated services may request short-term allocation more often, making the problem worse. Short-term allocation policies manage which queries receive cache allocations and when. In collocated settings, these policies should balance targeted query speedups against slowdowns caused by recurring cache contention. We present a model-driven approach that (1) predicts response time under a given policy, (2) explores competing policies and (3) chooses policies that yield low response time for all collocated services. Our approach profiles cache usage offline, characterizes the effects of cache allocation policies using deep learning techniques and devises novel performance models for short-term allocation with online services. We tested our approach using data processing, cloud, and high-performance computing benchmarks collocated on Intel processors equipped with Cache Allocation Technology. Our models predicted median response time with 11% absolute percent error. Short-term allocation policies found using our approach out performed state-of-the-art shared cache allocation policies by 1.2–2.3X.

ACM Reference Format:
Nathaniel Morris, Christopher Stewart, Lydia Chen, and Robert Birke. 2022. Performance Modeling for Short-Term Cache Allocation. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545094

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 [5]. 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 [28]. 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 [7, 26, 35].

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 [5]. Online services can use short-term allocation to speed up slow queries and meet response time goals. Consider a social networking website [4]. 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 [23]. 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 [8, 12, 13, 19, 20]. 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 [35]. 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

ICPP '22, August 29-September 1, 2022, Bordeaux, France
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545094

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 [6], 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 [5] to implement short-term cache allocation by tracking query executions at runtime. We used Deep Forests [36] 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 95th-percentile response time by up to 2.6X. We also compared our approach to static allocation, workload-aware cache partitioning [31] and IPC-driven dynamic cache allocation [12]. 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

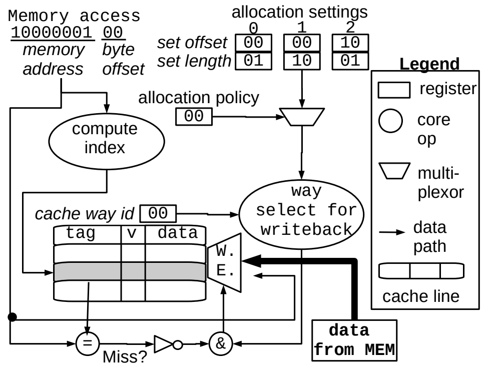


Figure 1: Data path for dynamic cache allocation.

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 2 is a primer on dynamic cache allocation, the targeted workload and system management goals. Section 3 presents the design of our modeling approach. Section 4 describes our implementation. Section 5 evaluates response time predictions, compares competing approaches and studies model-driven policy selection. Section 6 presents related work. Section 7 provides discussion and draws conclusions.

2 CACHE ALLOCATION TECHNOLOGY

Dynamic cache allocation manages which query executions can use cache lines at runtime [28]. Figure 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 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 [7, 27, 32]. LLC cache is a valuable resource that reduces response time by reducing main memory lookups [25].

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 [2, 8, 33]. 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 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 1 describes these properties for a cache line with offset v in the set of private cache lines $V_{(a,a')}$.

$$\begin{aligned} v \in V_{(a,a')} \rightarrow o_a \leq v < (o_a + l_a) \wedge \\ o_{a'} \leq v < (o_{a'} + l_{a'}) \wedge \\ \forall \hat{a} \in \left[A - a - a'\right] (v < o_{\hat{a}}) \vee (v > o_{\hat{a}} + l_{\hat{a}}) \end{aligned} \tag{1}$$

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

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 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[1] 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.

3.1 Stage 1: Profiling Cache Demands

As shown in Figure 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

[1]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.).

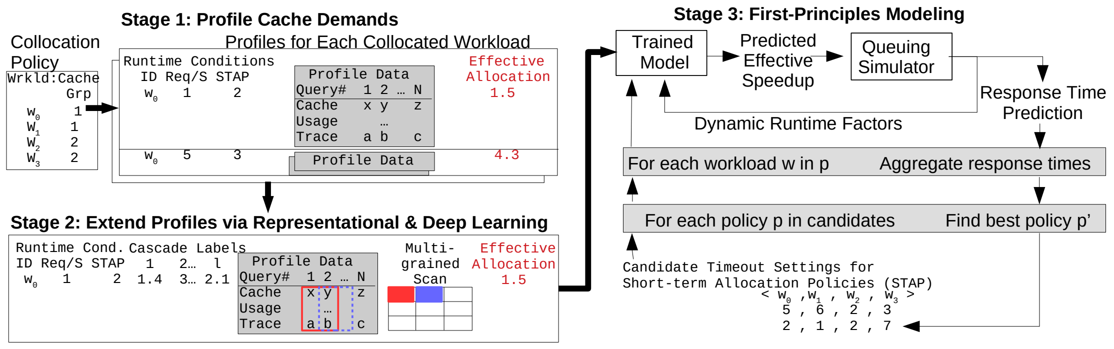


Figure 2: In stage 1, our approach collects cache usage data from each collocated workload and measures effective cache allocation. In stage 2, profile data is used to train deep-learning models. Finally, stage 3 models response time and explores policies.

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 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.

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

$$\text{P} = <\overrightarrow{\text{static}}, \overrightarrow{\text{dynamic}}, \overrightarrow{\text{query}_0}, ..., \overrightarrow{\text{query}_N}, \text{eff. allocation} > \quad s.t.|P| = K \tag{2}$$

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 3. Here, servicetime reflects average processing time for query execution under short-term allocation settings and timeout settings.

$$\text{EA} = \left(\frac{\text{servicetime}\left(\text{W}_{(\text{a,a}',\text{t})}\right)}{servicetime\left(W_{(a,a,0)}\right)}\right) / \left(\frac{l_{a'}}{l_a}\right) \tag{3}$$

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 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.

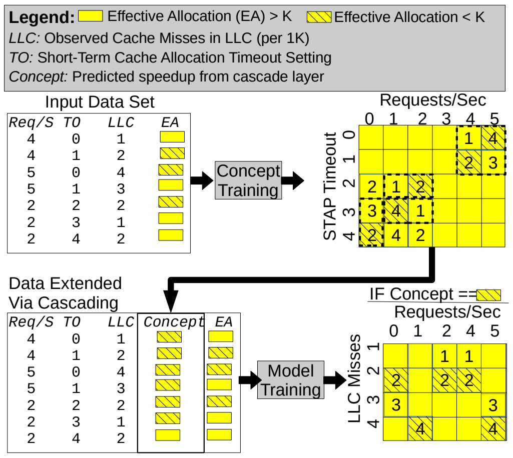
3.2 Stage 2: Deep Learning

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 [36].

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 [16].

Figure 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


Legend: 🟨 Effective Allocation (EA) > K ▨ Effective Allocation < K
LLC: Observed Cache Misses in LLC (per 1K)
TO: Short-Term Cache Allocation Timeout Setting
Concept: Predicted speedup from cascade layer
Input Data Set
Concept Training
Requests/Sec
STAP Timeout
Data Extended Via Cascading
Model Training
IF Concept == ▨
Requests/Sec
LLC Misses

Input Data Set

Req/S	TO	LLC	EA
4	0	1	🟨
4	1	2	▨
5	0	4	▨
5	1	3	🟨
2	2	2	▨
2	3	1	🟨
2	4	2	🟨

Data Extended Via Cascading

Req/S	TO	LLC	Concept	EA
4	0	1	▨	🟨
4	1	2	▨	▨
5	0	4	▨	▨
5	1	3	▨	🟨
2	2	2	▨	▨
2	3	1	▨	🟨
2	4	2	🟨	🟨

Figure 3: Deep learning uncovers concepts that reveal rich patterns and avoid over fitting.

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 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.

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 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.

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. 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 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 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.

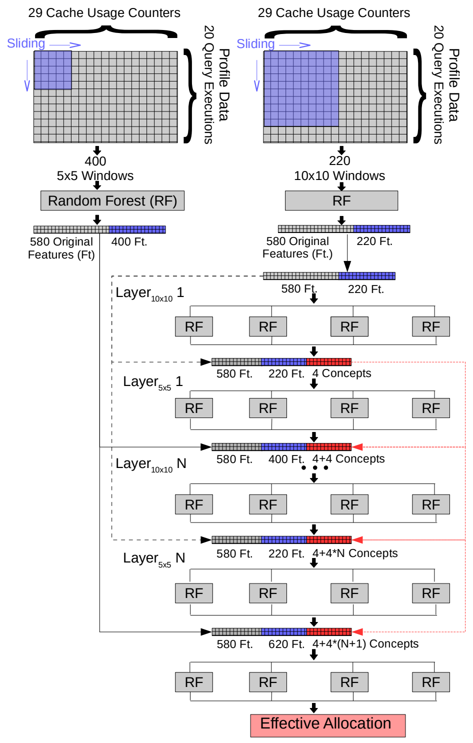


Figure 4: Multi-grained scanning and cascading support deep and representational learning in deep forests.

4.1 Deep Forest

Deep and representational learning can be implemented in many machine learning algorithms. We used deep forests [24, 36]. 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 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.

As seen in Figure 4, 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 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

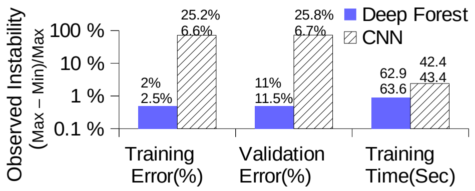


Figure 5: Random variation affects training accuracy, validation accuracy and training time for deep forests and CNNs. Numbers atop each bar reflect min and max.

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.

Each forest produces an estimate distribution function learned from training samples. In Figure 4, 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 5, we trained and evaluated CNNs and deep forests on profiled data 100 times. For the CNNs, we used PipeTune [22] 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 tradeoff 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 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 2: Runtime conditions studied.

5 EVALUATION

Table 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 [11]: 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 [29]: 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, it was 1 sec.

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 [3]: 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 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 and short-term allocation timeout for each online service relative to its average service time. If timeout was 150% and service time was 100 seconds, then short-term allocation would trigger at 150 seconds. Cache allocation is changed when a query times out.

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 [10]. 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 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.

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.

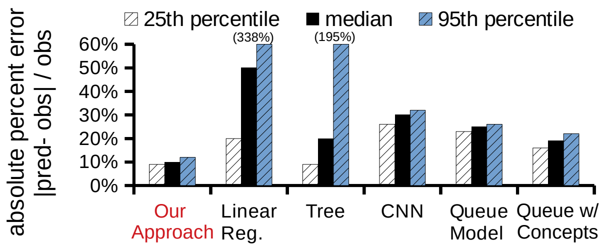


Figure 6: Accuracy of response time predictions for our approach, simple models, CNN, queuing simulator and a queuing simulator with concepts.

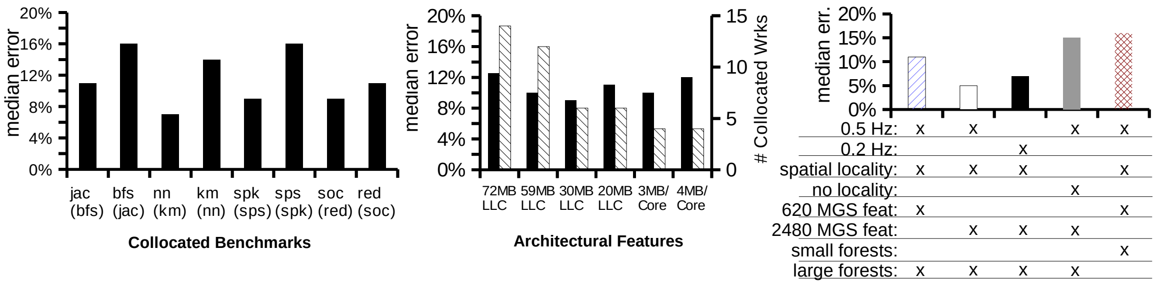
For our model, testing data outnumbered training data by 2 to 1, i.e., 33% training data and 66% testing data. For competing models, we used 70% training data and 30% testing data. We placed our approach at a disadvantage to ensure low profiling overhead. Later in this section, we evaluate accuracy of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

Accurate Response Time Prediction: In Figure 6, 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 6 arranges competing approaches from simple to complex. First, we compared to a linear regression model. As expected, this approach produced median error of 50% 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 6 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 [17] 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 6, achieved 26% median error. The Queuing Model approach used only our queuing simulator as described in section 3.3. This approach had 23% error.

Generalization: Figure 7(a) details our model's error for each workload in Table 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 7(b), we tested generalization across processor architectures. In addition to our default platform Xeon E5-2683 (40 MB LLC), we ran experiments on a two socket Xeon Platinum 8275 (72 MB LLC and 59MB LLC), Xeon 2650 (30 MB LLC) and on a Xeon 2620 (20 MB LLC). In all



(a) (b) (c)

Figure 7: (a) Accuracy of response time predictions for specific collocations. (b) Accuracy (black bar) and number of collocated workloads (striped bar) across processor cache sizes (c) Evaluation of multi-grained scanning parameters (sampling rate, spatial locality, window size and forest size).

cases, we fully utilized processor cores by running workloads concurrently (secondary Y-axis). We also changed collocation settings by allowing each workload to reserve 3 MB and 4 MB of LLC with respect to the latter two processors. Only 3 MB of LLC was reserved per workload for the Xeon Platinum 8275 (left two columns). The other processors reserved 2 MB of LLC. 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 6.

Implementation of Multi-Grain Scanning Strategy: Multi-grained scanning (MGS) learns representational features from the cache usage trace. In Figure 7(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 7(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 deep learning.

Figure 7(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.

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.

$$\frac{\text{response time}}{\text{exp. service time}} > T \tag{4}$$

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. Step 2: we chose policies that intersected both collocated services.

In Figure 8, we reported speedup from our model-driven approach against competing cache allocation approaches across multiple collocations. The competing allocation approaches are:

1. **No cache sharing**: Each workload has access to only its private cache. In Figure 8(a–d), all results are normalized to this baseline.
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 only private cache [31].
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, i.e., a random forest. These results show the effects of using simpler machine learning models.

In Figures 8(a–d), we reported speedup in 95^{th} percentile response time across 4 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. We also compared our full approach to an approach based on simple models. Even though simple models yield greater absolute percentage error, Figure 8(e) shows that, for most workloads, simple ML models can exceed dCat and match dynaSprint. Our approach outperforms dCat in most scenarios and achieves greater speedups.

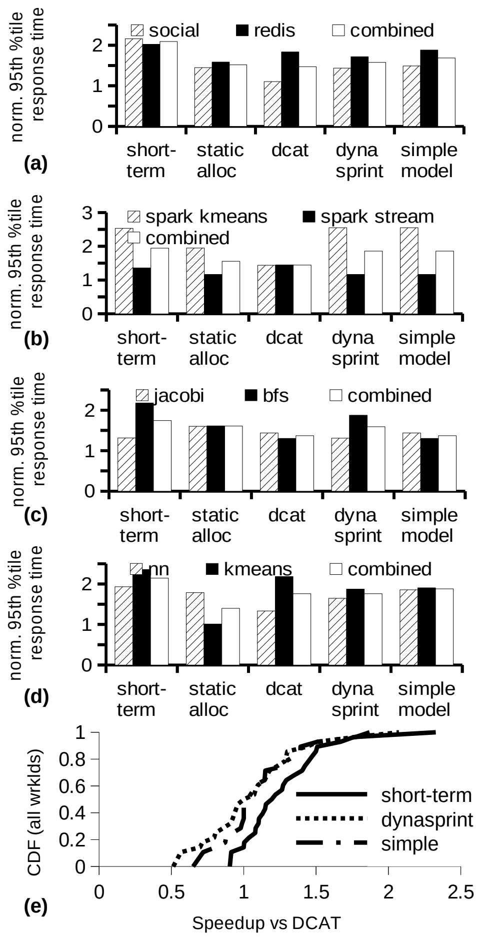


Figure 8: Comparing speedup in 95th percentile response time for competing cache allocation techniques.

6 RELATED WORK

Cache allocation that considers workload needs can often improve performance compared to workload agnostic allocation. Prior work studied cache allocation to reduce interference, maximize throughput and improve tail latency. Recently, dynamic cache allocation allows systems software to assign cache at runtime [5]. This section overviews prior work.

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. [31] 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. [4] proposed a resource manager that dynamically adjusts resources including cache for online services suffering from performance loss. Zhang et al. [34] propose a similar software-oriented solution. 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. [15] 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.

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 [14]. 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 [9, 30]. Prior works measure cache interference online [12, 18] and make decisions at an execution phase granularity.Qureshi et al. [21] 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. [12] 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.

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. Our approach uses deep learning to extract subtle relationships between application-level metrics (response time, query arrival and service rate) and micro-architectural metrics (cache misses and LLC allocation). Given 30 minutes to profile workloads, our approach can be used directly to manage short-term allocation. Our approach can also provide insights that yield better first-principles models.

Acknowledgments: We acknowledge NSF grants #1730043 and #1350941. We also thank Jayson Boubin, Eduardo Romero Gainza, Jianru Ding and Ruiqi Cao for their assistance.

REFERENCES

[1] D. H. Albonesi. 1999. Selective cache ways: on-demand cache resource allocation. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Israel, 248–259.
[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2018. Cedule: A scheduling framework for burstable performance in cloud computing. In *International Conference on Autonomic Computing (ICAC)*. IEEE, IEEE, Italy, 141–150.
[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC)*. IEEE, USA, 44–54.
[4] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *ASPLOS*. ACM, USA, 1–10.
[5] Intel Corporation. 2015. Improving real-time performance by utilizing cache allocation technology. https://01.org/cache-monitoring-technology. (2015).
[6] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. 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)*. ACM, USA, 1–14.
[7] Jianru Ding, Ruiqi Cao, Indrajeet Saravanan, Nathaniel Morris, and Christopher Stewart. 2019. Characterizing Service Level Objectives for Cloud Services: Realities and Myths. In *International Conference on Autonomic Computing*. IEEE, Sweden, 1–6.
[8] Songchun Fan, Seyed Majid Zahedi, and Benjamin C. Lee. 2016. The Computational Sprinting Game. In *ASPLOS*. ACM, USA, 561–575.
[9] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. 2010. Managing Contention for Shared Resources on Multicore Processors. *Commun. ACM* 53, 2 (Feb. 2010), 49–57.
[10] K Fengji. 2018. gcForest Version 1.1.1. https://github.com/kingfengji/gcForest. (2018).
[11] Gan, Yu et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *ASPLOS*. ACM, New York, NY, USA, 1–6.
[12] Ziqiang Huang, José A. Joao, Alejandro Rico, Andrew D. Hilton, and Benjamin C. Lee. 2019. DynaSprint: Microarchitectural Sprints with Dynamic Utility and Thermal Management. In *International Symposium on Microarchitecture*. IEEE, USA, 1–12.
[13] J. Kelley, C. Stewart, N. Morris, D. Tiwari, Yuxiong He, and S. Elnikety. 2015. Measuring and Managing Answer Quality for Online Data-Intensive Services. In *IEEE ICAC*. IEEE, France, 1–10.
[14] David George Kendall. 1959. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain. *Matematika* 3, 6 (1959), 97–112.
[15] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi. 2020. CuttleSys: Data-Driven Resource Management for Interactive Services on Reconfigurable Multicores. In *53rd International Symposium on Microarchitecture*. IEEE, Global Online Event, 1–12.
[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.
[17] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. *arXiv preprint arXiv:1807.05118* 1 (2018), 1–12.
[18] Amiya K. Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. 2014. Mitigating Interference in Cloud Services by Middleware Reconfiguration. In *Proceedings of the 15th International Middleware Conference*. ACM, USA, 1–12.
[19] N. Morris, S. M. Renganathan, C. Stewart, R. Birke, and L. Chen. 2016. Sprint Ability: How Well Does Your Software Exploit Bursts in Processing Capacity?. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, Germany, 173–178.
[20] N Morris, C Stewart, L Chen, R Birke, and et al. 2018. Model-driven computational sprinting. In *Eurosys*. ACM, Portugal, 1–12.
[21] Moinuddin K. Qureshi and Yale N. Patt. 2006. 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*. IEEE, USA, 423–432.
[22] Isabelly Rocha, Nathaniel Morris, Lydia Y Chen, Pascal Felber, Robert Birke, and Valerio Schiavoni. 2020. PipeTune: Pipeline Parallelism of Hyper and System Parameters Tuning for Deep Learning Clusters. In *Proceedings of the 21st International Middleware Conference*. ACM/IFIP, Global Online Event, 89–104.
[23] Jennie Rogers, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *SIGMOD International Conference on Management of Data*. ACM, USA, 337–348.
[24] Eduardo Romero, Christopher Stewart, Angela Li, Kyle Hale, and Nathaniel Morris. 2022. Bolt: Fast Inference for Random Forests. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. ACM, Canada, 94–106.
[25] Alan Jay Smith. 1982. Cache Memories. *ACM Comput. Surv.* 14 (1982), 473–530.
[26] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. 2013. Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs. In *IEEE ICAC*.
[27] C. Stewart, T. Kelly, and A. Zhang. 2007. Exploiting Nonstationarity for Performance Prediction. In *EuroSys Conf.*
[28] G Edward Suh, Larry Rudolph, and Srinivas Devadas. 2004. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing* 28, 1 (2004), 7–26.
[29] Yangjun Wang et al. 2016. Stream processing systems benchmark: Streambench. In *MS Thesis*. Aalto University, Finland, 1–66.
[30] Chi Xu, Xi Chen, Robert P. Dick, and Zhuoqing Morley Mao. 2010. Cache Contention and Application Performance Prediction for Multi-Core Systems. (2010).
[31] C Xu, K Rajamani, A Ferreira, W Felter, and et al. 2018. dCat: dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *ACM Eurosys*. ACM, Portugal, 1–12.
[32] Zichen Xu, Christopher Stewart, Nan Deng, and Xiaorui Wang. 2016. Blending On-Demand and Spot Instances to Lower Costs for In-Memory Storage. In *IEEE INFOCOM*.
[33] Seyed Majid Zahedi, Songchun Fan, Matthew Faw, Elijah Cole, and Benjamin C Lee. 2017. Computational Sprinting: Architecture, Dynamics, and Strategies. *ACM Transactions on Computer Systems (TOCS)* 34, 4 (2017), 12.
[34] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*. ACM, Germany, 89–102.
[35] Qin Zhao, David Koh, Syed Raza, Derek Bruening, and Weng-Fai Wong. 2011. Dynamic cache contention detection in multi-threaded applications. In *International conference on virtual execution environments*. ACM, USA, 1–12.
[36] Zhi-Hua Zhou and Ji Feng. 2019. Deep forest. *National Science Review* 6, 1 (2019), 74–86.