Fast and Accurate Workload Characterization Using Locality Sensitive Hashing

(Invited Paper)

Mohammad Shahedul Islam
Computer Science
University of Texas at San Antonio
rwk697@my.utsa.edu

Matt Gibson
Computer Science
University of Texas at San Antonio
matthew.gibson@utsa.edu

Abdullah Muzahid
Computer Science
University of Texas at San Antonio
abdullah.muzahid@utsa.edu

***Abstract*—Embedded applications are increasingly offloading their computations to a cloud data center. Determining an incoming application's sensitivity toward various shared resources is a major challenge. To this end, previous research attempts to characterize an incoming application's sensitivity toward interference on various resources (Source of Interference or SoI, for short) of a cloud system. Due to time constraints, the application's sensitivity is profiled in detail for only a small number of SoI, and the sensitivities for the remaining SoI are approximated by capitalizing on knowledge about some of the applications (i.e. training set) currently running in the system. A key drawback of previous approaches is that they have attempted to minimize the total error of the estimated sensitivities; however, various SoI do not behave the same as each other. For example, a 10% error in the estimate of SoI A may dramatically effect the QoS of an application whereas a 10% error in the estimate of SoI B may have a marginal effect. In this paper, we present a new method for workload characterization that considers these important issues. First, we compute an acceptable error for each SoI based on its effect on QoS, and our goal is to characterize an application so as to maximize the number of SoI that satisfy this acceptable error. Then we present a new technique for workload characterization based on *Locality Sensitive Hashing* (LSH). Our approach performs better than a state-of-the-art technique in terms of error rate (1.33 times better).**

***Keywords*-Application characterization, data center, locality sensitive hashing.**

I. Introduction

Embedded systems are increasingly becoming an inseparable part of modern life. Due to the advances in manufacturing and software industry, embedded systems are implementing various functionalities; however, they have limitations ranging from limited memory, bus speed, battery life, clock frequency etc. Fortunately, cloud computing can be used to alleviate some of the limitations. Devices like Amazon Echo [6], voice activated GPS, smart watch etc. are offloading most of their processing to a cloud data center. Therefore, it is imperative to classify embedded and cloud applications properly so that a better application response time (i.e. QoS) as well as data center utilization can be achieved.

Applications in a data center can interfere with each other due to various shared resources. Such interference can lead to performance degradation [25]. The situation gets worsened by continuous load fluctuation, application diversity, heterogeneity of the servers [23], [11] etc. Therefore, cloud operators often disallow application co-location or use an over-provisioning of resources for high-priority tasks. Due to such scheduling approaches, data center utilization has been found to be notoriously low [23], [11]. A major step toward successful scheduling while maintaining desired QoS and utilization would be a way to extract better insight about applications' characteristics.

Most prior techniques [25], [29] rely on a detailed offline approach or a long term monitoring and modeling approach for characterizing recurring applications. As a result, they are not effective for large data centers that receive tens of thousands of potentially unknown and often non-recurring applications each day. Recently, there is some work [16], [30], [17] that takes a two-fold approach. First, it characterizes an application's behavior towards various shared resources (referred to as Sources of Interference, or SoIs). Then, it schedules the application or adapts currently running applications accordingly. This is a promising direction. Inspired by this line of research, this paper ventures into the same overall approach but aims to increase characterization accuracy and scheduling efficiency. It takes inspiration from the domain of computational geometry.

When an application arrives, we would like to characterize the application by measuring its sensitivity toward various SoIs. Sensitivity of an application toward a particular SoI is denoted by its *Sensitivity Score*. It is measured as a fraction of the total available resource (corresponding to the SoI) which is required, at least, to maintain 95% of the application's stand-alone performance (i.e. QoS) in the best server [16]. Details of sensitivity score are discussed in Section II. Prior approaches treat each SoI equally; however, our experiments indicate that the same amount of error in estimating sensitivity scores for different SoIs can have significantly different effects on a scheduler's ability to pick the right server. Hence, the impact on QoS of applications can vary too. This is shown in Figure 1. The graph is obtained by artificially injecting inaccuracy in different SoI's sensitivity scores and measuring what fraction of applications achieve 95% or more QoS at the end. We used a state-of-the-art

scheduler, Paragon [16], for this experiment. 20% error in estimating sensitivity toward *Integer Processing Unit* and *Memory Capacity* causes 9% and 11% fewer applications to achieve 95% or more QoS. On the contrary, the same amount of inaccuracy for *Storage Capacity* causes only 5% applications to lose the QoS threshold. Thus, the experiment suggests that we can tolerate different ranges of errors in estimating sensitivity of different SoIs. To exploit this insight, we present a novel method for workload characterization that considers varying error tolerance intervals for different SoIs. For each sensitivity score, we determine the error interval that causes a 5% decrease in the number of applications achieving a QoS of 95% (or more). As seen in Figure 1, the memoryCapacity sensitivity score can be underestimated by 20% but cannot be overestimated by more than 10%. We would like to obtain a sensitivity score that is within this error range. We call this range the *Desired Error Interval* (DEI). We compute the DEI for each sensitivity score. We attempt to characterize an incoming workload so that our predicted sensitivity scores fall within the DEI of as many sensitivity scores as possible. To this end, we propose a new online workload characterization technique based on *Locality-Sensitive Hashing* (LSH) [8]. Given a set of n points in a d-dimensional Euclidean space, LSH is a hashing technique that allows us to find nearest neighbor points for any query point with nearly asymptotically optimal running time. We maintain a *Training Set* of applications for which all sensitivity scores are known. When a new application arrives, we quickly profile the new application for a few SoIs (the ones that need to be more accurate) and calculate sensitivity scores for those SoIs. We, then, use LSH to find applications in the training set with similar sensitivity scores so that we can predict the remaining sensitivity scores. We predict the remaining scores by taking the median of the similar applications' corresponding sensitivity scores. In order to take advantage of recurring nature of many embedded applications, we keep the sensitivity scores of recent applications in a table. When an application arrives, we check if it appears in the table. If so, it is a recurring application and we use the previously computed sensitivity scores. If, on the other hand, this is a new application, then we apply LSH based algorithm to compute the sensitivity scores.

We evaluated our scheme using SPEC, SPLASH2, PARSEC, PUMA Hadoop [5], CloudSuite [19], MiBench [21], AMB [26], MediaBench [3] as well as multiprogrammed workloads. We experimented with 10 different server configurations. Our approach predicts SoIs with higher accuracy. Compared to a state-of-the-art scheme, Paragon [16], we are **1.33** times more accurate. It takes only 0.011s for prediction. The result remains consistent across different servers.

The paper is organized as follows: Section II-A and II provide a background on sources of interference and locality-sensitive hashing; Section III explains the main idea; Section IV outlines the implementation issues; Section V provides the experimental results; Section VI discusses related work and finally, Section VII concludes.

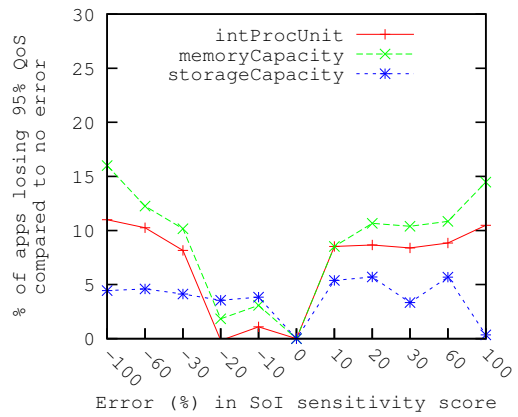

Figure 1: How application QoS is affected due to errors in different SoIs.

II. Background

A. Sources of Interference

We develop a set of kernels to characterize an application's sensitivity toward SoIs. For each SoI, we calculate two metrics - one to measure how much resource interference this application can *tolerate* from other applications and another to measure how much resource interference this application *causes* to other applications. Each of them are real numbers in $[0, 1]$. For each SoI, we develop a kernel whose intensity (i.e. its consumption of the particular resource expressed as a fraction of the total resource) can be progressively tuned up. The kernels are similar to iBench [15]. To calculate the *Tolerate* metric, we run the kernel for the SoI in parallel with the application and progressively tune up the intensity of the kernel until the application's performance degrades to 95% of its isolated performance in the best server. An application with a high tolerance toward the SoI will have a high value for this metric. To determine the *Cause* metric, we run the kernel in parallel with the application as before. This time, we tune up the intensity of the kernel until the *kernel's* performance degrades to 95% of its original isolated performance in the best server. The metric is then set to the value of $1 - intensity$. An application that consumes a lot of the resource will have a high value for this metric. We use the term *Sensitivity Score* to refer to both of these metrics. Since our kernels and measuring technique of sensitivity scores are similar to iBench, we are going to present only the high level idea of each kernel here. Interested readers can look into that paper for more details. We consider 10 SoIs. The SoIs are memory capacity and bandwidth, storage capacity and bandwidth,

network bandwidth, LLC capacity and bandwidth, TLB capacity, integer processing unit and floating point processing unit. The following shows a brief overview of how kernels for memory capacity and bandwidth can be written. Other kernels are developed similarly.

Memory Capacity (memCapacity): The kernel allocates $x\%$ of total memory (i.e. *mem_size*) to run at intensity level x. Memory is allocated at page granularity. The kernel runs for the same duration for each intensity level. After allocating memory, the kernel randomly accesses some of the allocated pages. The number of accesses is determined by the intensity level. At lower intensity, the kernel might need to remain idle after finishing all accesses. This is done to ensure equal processing time for each intensity level. The idle time is determined by the number of accesses as well as intensity level. After the duration has completed for the current intensity, the kernel tunes up its intensity. We consider intensity levels from the following discrete set: $\{0, 10, 20, \ldots, 90, 100\}$. Instead of increasing intensity level linearly, we use binary search in order to find an intensity x such that the performance level at intensity x is over 95% and at intensity $x + 10$ is under 95%.

```
mem_x=mem_size * x
while(coverage<mem_x)
   data=malloc(page_size)
   page_list.append(data)
   coverage+=page_size
t=0
while(t<duration)
   ts=time()
   data=page_list.get(r)
   temp+=data[r]
   wait(...)
   t+=time()-ts
```

Memory Bandwidth (memBandwidth): The kernel in this case performs a sequence of memory accesses. The number of memory accesses increases with intensity. At the highest intensity, the kernel occupies the entire memory bandwidth of the system. The accesses are done in an assembly function to ensure that the compiler cannot optimize them away. To reduce function-call overhead, the accesses are done in batch mode.

B. Nearest Neighbor Search

Our approach for workload profiling utilizes techniques from computational geometry, namely nearest neighbor search. In this subsection we give our motivation for considering these techniques, and we then provide some background on these techniques.

Motivation. In our profiling scheme, we will maintain a training set T of n applications for which we will determine all sensitivity scores in an offline preprocessing step. As described above, for each SoI we obtain sensitivity scores for two different metrics which gives us a total of twenty sensitivity scores. For each application $t_j \in T$, we denote its sensitivity scores $s_j^1, s_j^2, \ldots s_j^{20}$ where s_j^1 and s_j^2 are the tolerate and cause sensitivity scores for SoI-1, s_j^3 and s_j^4 are similar scores for SoI-2, etc. When a new application a_i arrives, we choose a small number of SoI and determine the exact sensitivity scores for these SoIs and approximate the remaining scores in an effort to save time. These approximate scores are determined by identifying the applications in T which are the most similar to a_i with respect to the computed scores.

To illustrate our high level idea, consider the following example. For simplicity, suppose that there are six total sensitivity scores, and suppose the applications $t_1, \ldots, t_5$ in Table I are the applications in T for which we know all six scores. Suppose for a new application a_i we determine $s_i^1 = 0.2$ and $s_i^2 = 0.3$, and we now wish to approximate the remaining four scores for a_i. The applications in T that are the most similar with respect to s_1 and s_2 are t_1 and t_3. We obtain the approximations for a_i as a function of the known scores of t_1 and t_3.

T	s_1	s_2	s_3	s_4	s_5	s_6
t_1	0.1	0.3	0.7	0.6	0.1	0.3
t_2	0.7	0.6	0.4	0.5	0.9	0.7
t_3	0.2	0.4	0.6	0.5	0.3	0.2
t_4	0.5	0.2	0.4	0.1	0.8	0.6
t_5	0.7	0.6	0.1	0.2	0.5	0.5

Table I: Training set example.

In order for this approach to be successful, we need to maintain a large enough training set T so that all incoming applications will have a few applications in T that are "similar" with respect to the sensitivity scores. At the same time, it is important that we can efficiently search T for the applications which are the most similar to an incoming application a_i. Thus the challenge is to model the problem in a way that we can simultaneously maintain a large, representative training set while being able to perform fast queries. To this end, we model the problem as a geometric nearest neighbor search problem. As each application has twenty sensitivity scores, each of which is a real number, it is natural to view each application as a point in $\mathbb{R}^{20}$ where the sensitivity scores are the coordinates of the point. Formally, for each application a_j we have a point $p_j = (s_j^1, s_j^2, \ldots, s_j^{20})$ (in the example, we have $p_1 = (0.1, 0.3, 0.7, 0.6, 0.1, 0.3), p_2 = (0.7, 0.6, 0.4, 0.5, 0.9, 0.7)$, etc.). Now consider two such points p_j and $p_{j'}$ in $\mathbb{R}^{20}$. If these points are "close" to one another in the geometric space, then their sensitivity scores are also "close" to one another, and therefore the corresponding applications a_j and $a_{j'}$ perform similarly with respect to the SoI. Likewise, if p_j and $p_{j'}$ are "far apart" from one another then a_j and $a_{j'}$ perform differently with respect to the SoI. Therefore we can find applications in T which are similar to a_i by performing nearest neighbor queries on these points.

Locality-Sensitive Hashing. One technique for nearest neighbor search is *Locality-Sensitive Hashing* (LSH). The high level idea behind LSH is to build a hash table on the points in P such that points that are nearby in P get hashed to the same "bucket" and points that are far apart in P get hashed to different buckets. Then nearest neighbor queries can be performed by determining which bucket the query point q lies in and then scanning the points of P that were hashed to the same bucket. LSH was originally introduced in 1999 [20] and several improvements have since been given [7], [13]. A C++ implementation given by the authors of [7] has been made available [2], and this is the implementation that we use.

LSH can be used to solve several variants of nearest neighbor search, and the variant that will be considered in this paper is *randomized R-near neighbor reporting*. That is, given parameters $R > 0$ and $\delta \in [0, 1)$, we will use LSH to report points in P whose distance to a query point q is at most R, and each such point will be reported with probability $1 - \delta$. Note that every point in P could be reported if R is large enough and could return no points if R is small enough; however, our motivation for considering this variant is that for an appropriately chosen R, all points whose distance to the query is at most R will be strong candidates for approximating sensitivity scores. Additionally we can obtain this set of candidates very quickly as the hashing does not depend on the size of T, and it scales very nicely for high-dimensional data (e.g. thousands of dimensions). This is sufficient background on LSH for understanding our workload profiling and scheduling techniques (in particular, knowing how it is implemented is not important for this paper); we refer the interested reader to see [8] for a nice introduction to LSH.

III. Main Idea: Characterizing Workloads

In this section, we give the details of our approach to workload profiling using locality-sensitive hashing. We will present two procedures in this section. The first is an offline procedure which is given a pool of applications for which we know the exact sensitivity scores, and it carefully chooses a subset of applications from the pool to serve as the training set. The procedure then outputs the associated LSH data structure. The second procedure is an online procedure which, given a new application, uses the LSH hash table given by the offline procedure to provide a fast and accurate approximation of the new application's sensitivity scores.

Offline Procedure. Let A denote a set of applications for which we know all twenty sensitivity scores. The offline procedure begins by choosing a training set $T \subseteq A$ of cardinality n for some parameter n. Recall that these applications naturally map to points in $\mathbb{R}^{20}$. Intuitively, we want the points associated with the training set applications to be distributed throughout $\mathbb{R}^{20}$, so that any new application received in the online procedure will have a few points in the training set nearby. To achieve this, we use the well-known *k-means* clustering algorithm [22] to partition A into k clusters for some constant k (i.e. $k = 10$). We interpret the applications assigned to the same cluster as being of the same "type", and we choose our training set to contain several applications from each of the different "types". To do this, we randomly choose n/k applications from each cluster to be in T.

Now that we have chosen our training set T of n applications, we are ready to build the LSH data structure. In the online procedure we will receive a new application a_i, and we choose α SoI to compute the associated 2α sensitivity scores (for some parameter α, e.g., $\alpha = 2$) and then approximate the remaining scores. Recall that each sensitivity score has a DEI, and our goal is to maximize the number of approximate sensitivity scores which fall within their DEI. The DEI has the form $(-Y, X)$ which implies that we can underestimate the score by at most $Y\%$ and we can overestimate the score by at most $X\%$. We define the *width of a DEI* to be $X + Y$. Since each SoI has two sensitivity scores, it also has two associated DEIs. Let w_1 and w_2 denote the two corresponding DEI widths of a SoI. We define the *width of an SoI* to be the minimum of w_1 and w_2. The SoI with the smallest widths have the least room for error, and accordingly we want to compute exact scores for the SoI with the smallest widths. To this end, we use the α SoI with the smallest widths.

Let S denote the set of α SoI with the smallest widths. When a new application arrives in the online procedure, we will exactly compute the 2α sensitivity scores associated with S. From this we obtain a point $p_i \in \mathbb{R}^{2\alpha}$, and therefore we want to perform nearest-neighbor queries in a 2α-dimensional space. For each application in T, we construct a point with 2α dimensions to be inserted into the LSH table. The coordinates of this point consists of the 2α sensitivity scores associated with S. We call this point the *projection* onto the SoI of S. Recall that for some parameters R and δ, the LSH table will return any point within distance R from a query point with probability $1 - \delta$. We choose R to be $\frac{2\alpha}{10}$ so that training set points whose distance is at most .1 away from our query point in each coordinate (on average) are returned. We choose δ to be 0.05 so that each point within distance R is returned with probability 0.95.

For example, again consider the training set given in Table I. Suppose $\alpha = 1$, and that the first SoI has the smallest width. Then we will exactly compute the first two sensitivity scores associated with this SoI in the online procedure, and therefore we want to build an LSH table based on these scores in the offline procedure. The projection of application t_1 onto these scores gives us the point $(0.1, 0.3)$, the projection of t_2 gives us $(0.7, 0.6)$, and the remaining points are constructed similarly. See Algorithm 1 for a formal description of our offline procedure. We remark that when implemented, one could execute this offline procedure

several times per day (e.g. every hour) to ensure that the training set is a good representation of the applications that are being received.

Algorithm 1 Offline Procedure

Let A be a set of applications for which all sensitivity scores are known. Use the k-means algorithm to partition A into k clusters. Let $\mathcal{C}$ denote output clusters.

$T \leftarrow \emptyset$

for all clusters $C \in \mathcal{C}$ **do**

Let $C' \subset C$ be a randomly-chosen subset of elements in cluster C such that $|C'| = n/k$.

$T \leftarrow T \cup C'$

end for

Let S denote the α SoI with the smallest widths, and let P denote the n points in $\mathbb{R}^{2\alpha}$ obtained by projecting the applications in T onto the SoI in S.

Build and save the LSH hash table P for parameters $R = \frac{2\alpha}{10}$ and $\delta = 0.05$.

Online Procedure. Now we assume that we have the LSH hash table stored in memory, and we are given a new application a_i for which we currently do not know any of its sensitivity scores. We compute the exact sensitivity scores of a_i for the 2α sensitivity scores associated with S, and this gives us a point $p_i \in \mathbb{R}^{2\alpha}$. We use the hash table to obtain a set of points N' in the training set within distance R of query point p_i. We then let N be the subset of N' consisting of the at most c points in N' that are closest to p_i for some parameter c (e.g., $c = 5$). We take the median of the scores of the applications in N to determine our estimate of the remaining scores for a_i. See Algorithm 2 for a formal description.

Algorithm 2 Online Procedure

Let a_i denote a new application for which we have no prior no knowledge of its sensitivity scores.

Let S denote the α SoI with the narrowest DEIs, and generate the 2α associated sensitivity scores for a_i. Let p_i denote the associated point in $\mathbb{R}^{2\alpha}$.

Let N' be the set of points returned by LSH when using the query point p_i, and let $N \subseteq N'$ be the at most c points from N' that are closest to p_i, breaking ties arbitrarily. If $N' = \emptyset$ then randomly choose scores for a_i and exit.

for all choices r such that s_i^r is unknown **do**

Let N^r denote the set of all scores s_j^r for each $p_j \in N$.

Set s_i^r to be the median of N^r.

end for

Extending to Many Server Configurations. In a heterogeneous data center, the effect of interference on an application may be quite different on different server configurations. Accordingly, the sensitivity scores for an application with respect to some SoI may be quite different for different configurations. In a data center with 10 server configurations, an application will have 200 sensitivity scores (20 scores for each server configuration). Our approach is similar to our previous one for a single server configuration. First, we choose the α SoI with the smallest widths, and then we compute the sensitivity scores with respect to these SoI for *three* different server configurations. We choose the configurations so that we are obtaining the scores for the "best" configuration, "median" configuration, and "worst" configuration. The scheduler keeps apriori list of "best", "median" and "worst" server configuration for a given SoI. The intuition is sensitivity scores for "good" server configurations may not be effective for predicting the scores for "bad" server configurations. For example, two applications which do not cause much interference on the best configuration may perform quite differently on the worst server configuration, and therefore we may not be able to accurately predict the scores for the worst configuration from the score of the best configuration. Given our choice of α SoI and three server configurations, we compute the 6α corresponding sensitivity scores (two scores per SoI per configuration). We then perform a LSH query to find points in the training set whose scores are similar to our computed scores, and we take the median of the scores of these points to approximate the remaining $200 - 6\alpha$ scores.

IV. Implementation

We design a Profile Manager (PM) to characterize an incoming application. PM perfoms short initial profiling runs and then, applies Algorithm 1 and 2 to predict the sensitivity scores of the application. PM starts by extracting architecture specific information (e.g., number of cores, available memory, LLC capacity, TLB size etc.) for the server where it is running. A data center can have 8-10 different server configurations at a time [16]. For each configuration, one server (of that configuration) is randomly picked and a copy of PM runs on it. When an application arrives, each PM profiles it for α SoIs that have the narrowest DEIs. DEIs are assumed to capture the significance of various shared resources in a particular data center and hence, remain fixed for a given data center. The profiling is done quickly and in parallel (in a minute) and then Algorithm 2 is used to predict sensitivity scores for the rest of the SoIs. All scores are stored in a local disk of the server. Periodically (every hour or so), each PM applies Algorithm 1 on the locally stored sensitivity scores to build the training set and LSH data structures. This is done to ensure that the training set remains a good representation of the data center applications. Locally stored sensitivity scores for older applications are removed in every few hours to keep the storage consumption under a limit. When sensitivity scores of an incoming application

are calculated by PMs, the data center scheduler considers those scores to schedule the application.

One might wonder what happens to applications that go through different phases. For those applications, the sensitivity scores might not characterize them properly once they enter into a different phase. This can lead to wrong server selection. We can handle such situations by monitoring the performance of applications, repeating the classification when performance degrades by some factor, and then reschedule them if necessary.

Cloud architecture (e.g., ZeroVM [27]) that runs on top of the storage clusters (e.g., Openstack Swift [4]) often targets short lived jobs (e.g., duration less than 2 minutes) using MapReduce model. To handle this type fo architecture, PM characterizes short lived map jobs from the complete run of a stage and all the later stages of the application get the same sensitivity scores.

V. Evaluation

The characterization parameters and server configurations for our experiments are given in Table II. Our data center contains 10 different server configurations and 5 servers for each configuration. We use all applications from SPEC CPU 2000 & 2006, Splash2, Parsec, PUMA Hadoop, CloudSuite, MiBench, AMB, and MediaBench benchmarks. We also generate 100 multiprogrammed workloads, each consisting of 4 applications from SPEC.

Character-ization parameter	Num. cluster, k = 5, **10**, 15, 20 Num. train. set, n = 50, 100, **150**, 200 Num. near. neighbor, c = **5**, 6, 7,..., 15 Num. train. SoI, α = **2**, 3, 4,..., 10
Def. Server	**core i5, 2.3GHz, 8 core, 8GB mem., 8MB LLC**
Best Server	xeon E5, 2GHz, 12 core, 32GB mem., 15MB LLC
Worst Server	P4, 2.8GHz, 1 core, 1GB mem., 1MB LLC

Table II: Parameters for experiments. Bold values are the defaults.

Sensitivity Score: Figure 2 shows how kernels are used to measure sensitivity scores. Figure 2(a) shows how an application, namely *adpcmEncode*, behaves with different intensity of the kernel for *intProcUnit*. "*" denotes the 95% performance point. This graph used for calculating the *Tolerate* sensitivity score. Figure 2(b) shows how the kernel's performance (w.r.t. its standalone performance) changes with different intensity. This graph is used for calculating the *Cause* sensitivity score. The Tolerate and Cause score for intProcUnit are 0.12 and (1-0.9)=0.10 respectively. Figure 2(c) and (d) show similar graphs for *memCapacity*.

DEI Analysis. We experiment with different errors injected in SoI scores (SS) and measure what fraction of applications achieve at least 95% QoS. We construct DEI of an SoI as the error interval outside which 5% or more applications (compared to the ones obtained by using accurate SS) fail to reach the QoS threshold. Table III shows

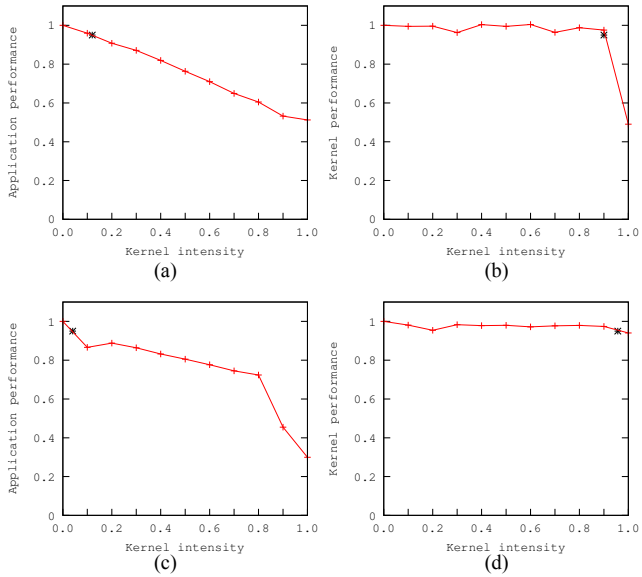

Figure 2: Sensitivity scores - (a) and (b) for intProcUnit and (c) and (d) for memCapacity.

the DEIs for different SoIs. *memBandwidth* and *tlbCapacity* have the smallest widths. Hence, we choose them for initial profiling.

SoI	Tolerate (%, %)	Cause (%, %)	Width
floatProcUnit	(-100, 100)	(-60, 10)	70
intProcUnit	(-100, 100)	(-60, 100)	160
llcBandwidth	(-20, 0)	(-20, 60)	20
llcCapacity	(-20, 100)	(-20, 10)	30
memBandwidth	(-10, 0)	(-30, 20)	10
memCapacity	(-20, 0)	(-100, 100)	20
netBandwidth	(0, 20)	(-10, 10)	20
storageBandwidth	(-60, 30)	(-60, 20)	80
storageCapacity	(0, 100)	(-100, 0)	100
tlbCapacity	(0, 10)	(-10, 30)	10

Table III: DEI for different SoIs.

Prediction Ability. Figure 3a shows the comparison of LSH based approach against two versions of Paragon. One version chooses the same initial SoIs as LSH while the other (Paragon(r)) chooses SoIs randomly. For implementing Paragon, we used a hand tuned version of matrix factorization algorithm available from Apache Mahout [1]. For illustration, the figure shows the case for an ideal predictor. For each approach, we calculate how many sensitivity scores (SSs) fall within the corresponding DEIs. LSH based approach consistently performs better or similar to both versions of Paragon. The first version of Paragon performs better than Paragon(r). Therefore, from now on, we will only consider the first version. For the default choice of 2 SoIs, LSH predicted 14 out of 16 SSs correctly. This is 1.33 times

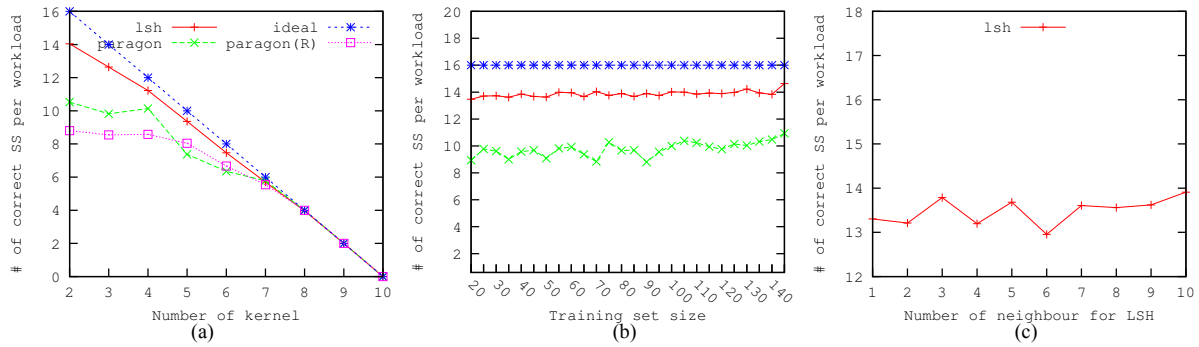

Figure 3: Prediction error for lsh and Paragon.

more accurate than Paragon.

Figure 3(b) shows correct SSs as we increase the training set size. As before, we keep other parameters fixed at the default values. In LSH-based approach, with larger training set, we are likely to find many workloads that very similar to the incoming one and hence, the accuracy increases. For Paragon the accuracy tends to increase too. However, it always remains below the one for LSH. Figure 3(c) shows the accuracy as we increase the number of nearest neighbors in LSH-based approach. The accuracy increases with more neighbors since we can find many similar applications. For the default value of 5 neighbors, it predicts 14 SSs per workload.

Figure 4(a) shows the number of correctly predicted SSs per workload for different server configurations. In each configuration, our approach works better than Paragon. For the best (i.e. Server 10) and worst (i.e. Server 1) server LSH predicts 14 and 10.5 SSs correctly. Figure 4(b) shows, for each SS, what fraction of applications has been correctly predicted with LSH based approach and Paragon. Here, we are considering the worst server. Out of 16 SSs, LSH based approach and Paragon predicts 5 scores with similar accuracy. Among the remaining scores, LSH predicts 8 scores more accurately whereas Paragon predicts 3 scores more accurately. Other servers provide similar results.

QoS and Utilization. We measure the QoS of different applications running in the data center. Figure 5 shows distribution of QoS using actual sensitivity scores as well as LSH and Paragon based scores. We used Paragon scheduler for our data center. Each stack corresponds to the % of applications that suffer from a certain level of QoS degradation. Using actual scores, 87% applications suffer from QoS degradation of at most 5%. For LSH and Paragon, the number is 83% and 79% respectively. So, compared to Paragon, LSH based prediction allows 4% more applications to achieve QoS of 95% or more. Note that the numbers for Paragon look different than those reported due to the large disparity among the server configurations and their

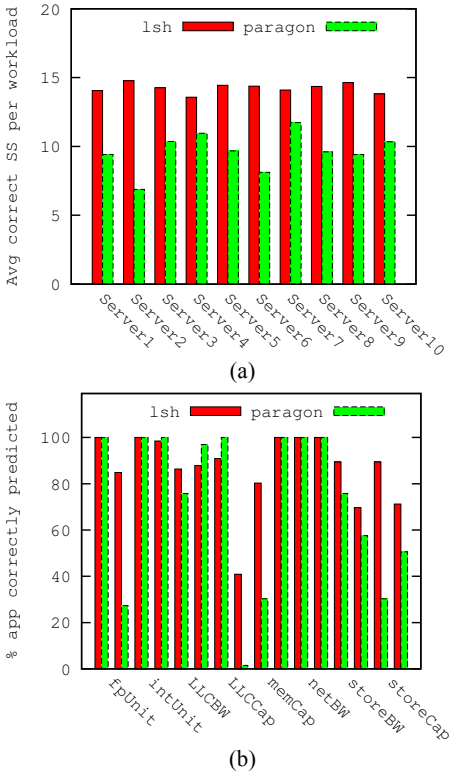

Figure 4: Breakdown of prediction ability.

quantities.

Figure 6 shows data center wide server utilization. The graph shows utilization for each server configuration. Figure 6(b) & (c) show utilization using LSH and Paragon based prediction. This is contrasted with utilization achieved using actual scores (Figure 6(a)). The graphs are similar except for server configuration 4 and 5. Paragon over utilizes configuration 4 while under utilizes configuration 5. LSH based approach distributes the load between these two con-

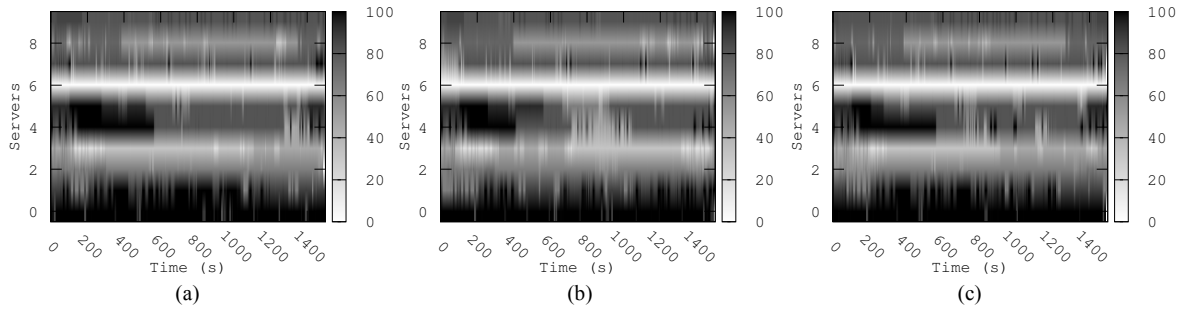

Figure 6: Server utilization using (a) actual scores, (b) LSH, and (c) Paragon based scores.

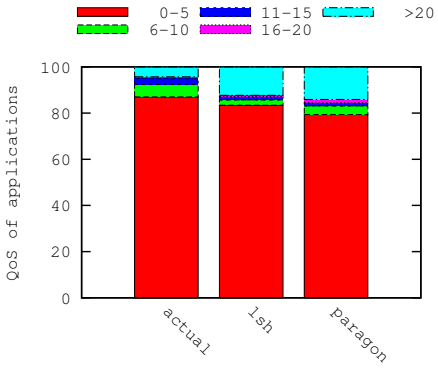

Figure 5: QoS achieved using different approaches.

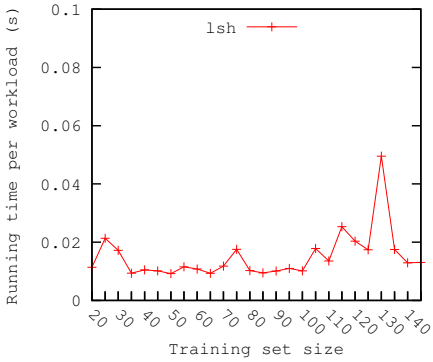

Figure 7: Running time analysis.

figurations more evenly. Overall, average server utilization using actual sensitivity scores, LSH, and Paragon based scores are 59.4%, 59.3%, and 57.3% respectively.

Time Analysis. Finally, we measure how the prediction time per workload changes as we change the size of training set (Figure 7). Prediction time tends to remain the same for training set size up to 100. After that, it tends to increase slightly. For our default training set size of 150, the prediction time is 0.011 seconds for each workload. This is extremely fast. Our initial profiling takes (≈)60 seconds.

VI. Related Work

There has been significant work on data center workload characterization and modeling [14], [18], [24], [9], [10]. This line of work generates workloads with characteristics that closely resemble those of the original applications. The generated workloads are then used in system studies. Although this is a viable approach, the generated workloads sometimes cannot capture every aspects of the applications. Moreover, they are not suitable for unknown applications. Mars et al. [25], [30] designed two kernels that create tunable contention in memory capacity and bandwidth to quantify the sensitivity of a workload to memory interference. The kernels are used during either offline profiling [25] or online profiling [30]. Tang et al. [28] designed SmashBench, a benchmark suite for cache and memory contention. Delimitrou et al. [15] proposed iBench, a benchmark suite to obtain sensitivity curve of a workload for 15 shared resources. The kernels that we used in our approach are influenced by iBench. None of the work [25], [30], [28], [15] predict sensitivity of a workload based on its similarity to existing workloads. Paragon [16] is the first work to predict a workload's sensitivity based on the knowledge about existing applications. The prediction is done using Netflix [12] algorithm. Once the sensitivity scores are predicted, Paragon applies a greedy algorithm for sever selection to maximize utilization while minimizing interference. The work is later extended in Quasar [17] where server selection is done based on predicted sensitivity scores and performance constraints given by a user.

VII. Conclusion

We have presented new method of evaluating workload characterization and have presented a new technique that is based on locality-sensitive hashing. Given a new application, we are interested in approximating its dependence on certain

resources. Due to time constraints, we can only spend a small amount of time profiling the application. After this profiling, we are able to identify similar applications from a training set extremely quickly using locality-sensitive hashing. We then use these similar applications to approximate the remaining information for the new application. We demonstrated the effectiveness of our approach with respect to our new evaluation metric by comparing our results with that of Paragon, a technique based on the Netflix algorithm. We demonstrate that our approach predicts more accurately (by a factor of **1.33**) than Paragon.

REFERENCES

[1] "Apache Web Server," http://www.apache.org/.

[2] "LSH Algorithm and Implementation," http://http://www.mit.edu/~andoni/LSH/.

[3] "Mediabench," http://mesl.ucsd.edu/spark/benchmarks.shtml.

[4] "Open Source Software for Building Private and Public Clouds," http://www.openstack.org.

[5] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," *Technical Report, Purdue ECE Tech Report TR-ECE-12-11*, 2012.

[6] Amazon, "Amazon echo," http://www.amazon.com/echo.

[7] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Foundations of Computer Science, 2006. FOCS '06. 47th Annual IEEE Symposium on*, Oct 2006, pp. 459–468.

[8] ——, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12. New York, NY, USA: ACM, 2012, pp. 53–64. [Online]. Available: http://doi.acm.org/10.1145/2254756.2254766

[10] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '98/PERFORMANCE '98. New York, NY, USA: ACM, 1998, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/277851.277897

[11] L. A. Barroso, "Warehouse-scale computing: Entering the teenage decade," in *ISCA*, June 2011.

[12] R. Bell, Y. Koren, and C. Volinsky, "The BellKor 2008 Solution to the Netflix Prize," *Technical report*, vol. AT&T Labs, October 2007.

[13] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *SCG*, June 2004.

[14] C. Delimitrou, S. Sankar, A. Kansal, and C. Kozyrakis, "Echo: Recreating network traffic maps for datacenters with tens of thousands of servers," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, Nov 2012, pp. 14–24.

[15] C. Delimitrou and C. Kozyrakis, "ibench: Quantifying interference for datacenter applications," in *IISWC*, September 2013.

[16] ——, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ASPLOS*, March 2013.

[17] ——, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 127–144. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541941

[18] C. Delimitrou, S. Sankar, K. Vaid, and C. Kozyrakis, "Decoupling datacenter studies from access to large-scale applications: A modeling approach for storage workloads," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, ser. IISWC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 51–60. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2011.6114196

[19] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '12. New York, NY, USA: ACM, 2012, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150982

[20] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB*, September 1999.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15

[22] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Applied statistics*, pp. 100–108, 1979.

[23] U. Hoelzle and L. A. Barroso, *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[24] Y. Joo, V. Ribeiro, A. Feldmann, A. C. Gilbert, and W. Willinger, "Tcp/ip traffic dynamics and network performance: A lesson in workload modeling, flow control, and trace-driven simulations," *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 2, pp. 25–37, Apr. 2001. [Online]. Available: http://doi.acm.org/10.1145/505666.505670

[25] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations," in *MICRO*, December 2011.

[26] J. Peleska, A. Honisch, F. Lapschies, H. Löding, H. Schmid, P. Smuda, E. Vorobev, and C. Zahlten, "A real-world benchmark model for testing concurrent real-time systems in the automotive domain," in *Proceedings of the 23rd IFIP WG 6.1 International Conference on Testing Software and Systems*, ser. ICTSS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 146–161. [Online]. Available: http://dl.acm.org/citation.cfm?id=2075545.2075556

[27] P. Rad, V. Lindberg, J. Prevost, W. Zhang, and M. Jamshidi, "ZeroVM: Secure Distributed Processing for Big Data Analytics," in *World Automation Congress*, August 2014.

[28] L. Tang, J. Mars, and M. L. Soffa, "Compiling for niceness: Mitigating contention for qos in warehouse scale computers," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259018

[29] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "Dejavu: Accelerating resource allocation in virtualized environments," in *ASPLOS*, March 2012.

[30] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ISCA*, June 2013.