



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

KONGMING: Performance Prediction in the Cloud via Multidimensional Interference Surrogates

Z. Bowen, G. Bronevetsky, M. Casas-Guix, S.
Bagchi

January 23, 2014

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

KONGMING: Performance Prediction in the Cloud via Multidimensional Interference Surrogates

Bowen Zhou[†], Greg Bronevetsky[‡], Marc Casas^{*}, and Saurabh Bagchi[†]

[†]Purdue University [‡]Lawrence Livermore National Laboratory ^{*}Barcelona Supercomputing Center
{bzhou,sbagchi}@purdue.edu, bronevetsky1@llnl.gov, marc.casas@bsc.es

Abstract—As more and more applications are deployed in the cloud, it is important for both the user and the operator of the cloud that the resources of the cloud are utilized efficiently. Virtualization and workload consolidation techniques are pervasively applied in the cloud to increase resource utilization while providing isolated execution environments for different users. While virtualization hides the architectural details of the underlying hardware, it can also increase the variability in application execution times due to heterogeneity in available hardware, and interference from other applications sharing the same hardware resources. This reduces both the productivity of cloud platforms and limits the degree to which software co-location can be used to increase its efficiency.

This paper describes KONGMING, a statistical approach to model the relationship between the resource availability on a given machine (a function of its design and any applications currently running on it) and the performance of applications that run on it. Based on a fixed number of training runs of a given application KONGMING can predict how long the application will run on a new machine on which an unknown set of applications are currently executing. Further, given models of two applications KONGMING can predict how long both will run when they are co-scheduled on the same machine. We demonstrate KONGMING’s effectiveness and accuracy by applying it to executions of the SPEC2006 and YCSB benchmarks on both a dedicated cluster and EC2 virtual machine instances.

I. INTRODUCTION

Accurate prediction of application behavior is a critical challenge in the design of efficient computing systems. Given finite computing resources it is important to map computational work to these resources in a way that maximizes overall system efficiency and/or guarantees some level of performance of each individual task. This is relevant in fields such as cloud computing where large numbers of users submit tasks to a pool of computing resources. The resource provider must assign these tasks to cores, caches, memories and networks in a way that uses the least resources to execute the given tasks while bounding the performance degradation caused by contention for shared resources by concurrently executing tasks. While providers such as Amazon have successfully shown the ability to protect some resources from contention (e.g. the computing ability of individual cores, the storage capacity of main memory), contention for other resources such as disks and memory hierarchies can be significant. The same problem appears in domains such as High-Performance Computing, where a single application is granted exclusive access to many nodes and their network. The individual tasks within a single application must share resources much like tasks from different applications, requiring either the application or an external task scheduler

(e.g. Charm++ [13]) to assign tasks to resources in a way that maximizes overall performance.

Effective resource scheduling in these and other computational domains require accurate prediction of application behavior that accounts for the effects of resource contention. Specifically, this paper focuses on the following prediction questions:

- Given application A and a system that is currently executing some set of applications, predict A ’s execution time on this system, and
- Given two applications A and B and a given unloaded system, predict how long A and B will execute if co-scheduled on the same system.

This task is very difficult both due to the general complexity of hardware and software and the lack of information available about their internal structure. To predict how long an application will run on a given system it is necessary to account for the way it uses the system’s resources, how other co-running applications use the same resources and how the system’s hardware and software arbitrates the use of these resources. For example, consider multiple processes accessing the same disk. The execution times of these I/O accesses depends on how they’re distributed in time (e.g. bursts of traffic or periodic access) and space (e.g. whether the disk head moves across cylinders). Concurrent accesses by multiple processes induces more complex temporal and spatial distribution of accesses that depends on the exact set of executing processes. Given perfect information about each process’s I/O accesses and the structure of the system’s I/O hardware and software it is possible to predict the performance of each disk access with a sufficiently detailed simulation. Unfortunately, because such information is rarely available in practice simulation-based predictions are difficult to carry out for realistic use-cases. In the particular case of cloud computing users have information about their applications but no knowledge of the hardware or co-running applications, while cloud providers have some understanding of the hardware and system software but no knowledge of the internals of the applications that execute on each node. The other issue with simulation is its high cost, which needs to be paid on every change in the properties of the hardware (e.g. CPU frequency) or software (e.g. set of executing applications).

The inherent limitations on the information available about the properties of hardware and software on a given machine at a given point in time motivate the need for techniques that learn as much of this information as possible based on

empirical observations and statistical modeling. This paper presents KONGMING, a statistical approach to predicting application behavior in a wide range of dynamic scenarios that accounts for variability in hardware properties and the impact of resource contention. KONGMING quantifies the effective performance of a given system at a given point in time by executing a suite of micro-benchmarks on it. The execution times of these micro-benchmarks quantify for the effective performance of a wide range of node resources, including CPU, memory hierarchy and disk for several representative ways of using these resources. To make predictions for a given application A KONGMING runs A a bounded number of times on differently-loaded systems, quantifying the type of load in each run using our micro-benchmarks. This data is used to train a statistical model that maps effective system performance to A 's execution time. KONGMING predicts how long A will run on a given system by running the micro-benchmarks on the system and applying A 's model to their execution times. To predict how long A will run when executing concurrently with application B on an otherwise unloaded system we use our micro-benchmarks to quantify B 's impact on the system's effective performance and then apply A 's model on this information.

To predict the behavior of arbitrary combinations of applications on different systems KONGMING must execute each application a bounded number of times, with the total training work scaling linearly in the number of applications. This is in contrast to much larger space of possible system configurations: polynomial number of possible configurations of node hardware (memory, CPU, frequency, etc.) multiplied by an exponential number of sets of applications that may be concurrently executing on a given system. KONGMING does not address the problem of predicting an application's behavior based on its inputs. Since this problem is orthogonal to managing the complexity of hardware and software interactions, solutions to this problem are complementary to our work on KONGMING. Finally, although the focus of this paper is on prediction of execution time, the KONGMING approach can be used to predict of other behaviors such as energy use, which is part of our future work.

Prior work, which is detail in Section V, has considered the problem of predicting application performance while accounting for resource contention. Cuanta [11] employs a canonical set of benchmarks to simulate cache load patterns and attempts to match applications to the benchmarks that represent them, while Bubble-Up [14] uses a memory intensive "bubble" to simulate memory interference. Both focus on resource contention for a single type of memory resource and do not address the problem of interacting effects of contention for multiple resources. Further, since Cuanta is tied to hardware parameters, it is not portable across platforms. Paragon [8] identifies a representative set of applications and predicts the behavior of new applications across platforms by relating them to the established set. However, it needs to run all the reference applications on all the platforms of interest to make a prediction. Finally, Zhao et al [18] predict interference-induced performance degradation using a piecewise regression model but require users to manually partition the space of interference effects.

KONGMING advances the state of the art by presenting

an automated approach to predicting application execution time based on empirical measurement of a system's effective performance. It transparently accounts for effects of hardware, system software and resource interference by executing applications. Its ability to predict how long an application will run on a given system and how long pairs of applications will run when scheduled together provide a valuable capability for proactive and efficient workload scheduling. Section II presents a basic theoretical model of how applications behave on systems. In Section III this model is instantiated with a specific abstraction based on the insight that effective system performance should be quantified based on empirical measurements rather than detailed but impractically slow simulation models. Section IV presents the experimental evaluation of the KONGMING approach, showing that it accurately predicts application behavior for our two target use-cases on both Amazon EC2 systems as well as on systems within a dedicated cluster. Finally, related work is discussed in Section V.

II. MODEL OF APPLICATION PERFORMANCE

For a fixed input an application is a dependence graph of basic operations such as data accesses and computations. The execution time of an application depends on the execution times of its individual operations and how they are connected by the dependence graph. The execution time of each operation depends on the environment in which it executes, which includes the properties of the hardware and runtime software on which it runs and the behavior of any applications currently utilizing shared resources. More formally, let Ops be the set of all operations and Exe the set of all execution environments. Let $T_{Op}(o, e)$ be the execution time of operation $o \in Ops$ in environment $e \in Exe$ and let $\overrightarrow{T_{Op}}(e)$ be the vector of the execution times of all operations in environment e . For any application A let $T_{App}(Struct_A, \overrightarrow{T_{Op}}(e))$ be the execution time of A in environment e , which is a function of its structure $Struct_A$ (e.g. the dependence graph) and the mapping T_{Op} of operations to their execution times.

This very high-level formalization of application performance can be instantiated in various ways to make a concrete prediction of application performance. The most detailed instantiation is real hardware, where set Exe corresponds to all the electrical configuration of circuit wires and set Ops is the set of transitions between wire configurations. This accounts for all the electrical, power and performance phenomena of the hardware and software at hand but is clearly not useful for prediction. An approximation of this formalization may be a gate-level simulation (e.g. VHDL), where the set Exe_{gate} of execution environments corresponds to all on/off configurations of wires and set Ops_{gate} contains the transitions between them. Each element in Exe_{gate} and Ops_{gate} is a set of elements of sets Exe and Ops , respectively, grouping the many electrical wire states and transitions that are logically equivalent into a single set. While the gate-level model is predictive, its high cost makes it impractical for routine use. A micro-architectural model represents a somewhat coarser instantiation where each execution environment in set Exe_{micro} is a subset of Exe_{gate} that denotes all the wire configurations that correspond to the same micro-architectural state, and same for operations. This model produces very accurate estimates of execution times but is still very expensive. An architectural-level model where

execution environments are architectural states and operations correspond to opcodes and high-level cache states, are a further coarsening that gives up some predictive accuracy for a lower prediction cost.

Each coarser-level model partitions the set of possible execution environments and operations into a smaller number of subsets to provide the appropriate balance of performance and predictive accuracy. The above examples correspond to simulation-based models that take as input an appropriately detailed description of hardware and software and simulate how the combined system may behave at the appropriate level of granularity. KONGMING takes an inverse approach based on empirical observation and statistical modeling. It defines an approximation of *Exe* and *Ops* based on observed behaviors of software on real systems and uses statistical modeling to relate the properties execution environments specified at this granularity to the execution times of applications of interest. KONGMING’s high-level approximation of hardware and software behavior provides a useful degree of predictive accuracy, while being fast to compute and requiring no detailed knowledge of (i) the structure of the hardware and software on the machine on which a target application may run, or (ii) the internal structure of the application itself.

III. METHODOLOGY

KONGMING groups the operations an application may execute into equivalence classes and estimates how operations within a given set behave in different execution environments by actually executing them in such environments. Table I lists the sets of operations $Ops_{KONGMING}$ that KONGMING considers. Each operation set is instantiated as a micro-benchmark that repeatedly executes a representative operation from each set. These “measurement” benchmarks include (i) memory accesses that use either random or strided access patterns with blocks of size 1KB, 1MB and 1GB, (ii) arithmetic operations on values steamed to/from memory, (iii) random disk writes with blocks of size 4MB, 64MB and 1GB, and (iv) file creation and deletion. The operator $T_{Op}^{KONGMING}(o, e)$ is KONGMING’s estimate of the execution time of operation o in execution environment e , which is computed by running the measurement benchmark associated with o in this environment. Since KONGMING does not rely on knowledge of the internal details of the system to which it is applied, it partitions the set of execution environments based on the execution times of the measurement benchmarks when run in each environment. Thus, two execution environments $e, e' \in Exe$ are grouped into the same environment in $Exe_{KONGMING}$ if $\forall o_{KONGMING} \in Ops_{KONGMING}. T_{Op}^{KONGMING}(o_{KONGMING}, e) = T_{Op}^{KONGMING}(o_{KONGMING}, e')$.

The above formalization makes it possible to create a model of application behavior that can predict (i) how long an application will run when executed on a given system and (ii) given some system how long two specific applications will run when executed concurrently. These predictions are made based on a set of experiments that is linear in the number of applications for which predictions must be made and does not depend on the number of permutations of systems and the applications that may run on them. The first step of our approach is to compute the operator $T_{App}^{KONGMING}(Struct_A, T_{Op}^{KONGMING})$ for a given application A . Since KONGMING does not have

access to the application’s structure, this operator is computed statistically, by executing both the application and our set of measurement benchmarks in a wide range of execution environments. Each experiment produces a different observation of how $T_{Op}^{KONGMING}$ relates to A ’s execution time. These are used to train a statistical regression model that takes as input the vector of operation execution times in each environment ($T_{Op}^{KONGMING}(o, e)$ for all e) and as output the execution time of A in the same environment. The resulting model $T_{App}^{KONGMING}(T_{Op}^{KONGMING})$ approximates the relation T_{App} without knowledge of $Struct_A$. In this study we considered the models listed in Table II. The second step is to quantify the effect that the execution of application A has on the execution environment observed by other co-executing applications. This is done by running the measurement benchmarks concurrently with A to produce the vector T_{o, e_A} of their observed execution times (e_A is the execution environment induced by A on a given system).

Instead of waiting to find the a broad range of execution environments on which to run its experiments KONGMING creates them synthetically. This is done by running different configurations of the measurement benchmarks on a given machine to place various fixed amounts of load on different system resources, while the target application and measurement benchmarks execute on it. Note that although these load inducers share source code with the measurement benchmarks their purpose is generate a variety of execution environments on which the main application and measurement benchmarks run. The same effect could be achieved in other ways by varying available power, adding or removing memory modules, or executing some other suite of applications.

The model trained on the resulting space of execution environments is used in two ways. First, consider a machine that is made available for application A to run on. The effective performance of the machine is unknown because both its hardware details and the applications that may be running on it are unknown. KONGMING predicts how long A will execute on this machine (in its execution environment e) by first executing the suite of measurement benchmarks on this machine to produce their execution times $T_{Op}^{KONGMING}(o, e)$. It then applying $T_{App}^{KONGMING}(T_{Op}^{KONGMING})$ to produce A ’s predicted execution time on this machine. If there is a change in the machine’s configuration (e.g. its frequency) or the set of applications that are running on it, the measurement benchmarks will need to be re-executed to quantify the new execution environment and adjust its predictions.

Second, consider two applications A and B . A scheduler may wish to co-locate them on the same compute node within a cluster but needs to know how long each will run when it is sharing resources with the other. Since the execution of each application creates the execution environment within which the other runs KONGMING can use its models to answer this question. The execution environment induced by A is e_A and it is characterized via vector T_{o, e_A} , which includes the execution times of all the measurement benchmarks when they were run concurrently with A . $T_{App}^{KONGMING}(T_{o, e_A})$ is then KONGMING’s prediction of the execution time of B when run concurrently with A , and the reverse procedure is used to predict A ’s execution time. This analysis needs to be performed separately for each type of co-location that is employed, since two

TABLE I: The List of Interference Surrogates of KONGMING.

Name	Description	Resource
memory.streaming.1K	Streaming access a 1KB array	Cache
memory.streaming.1M	Streaming access a 1MB array	Cache/Memory
memory.streaming.1G	Streaming access a 1GB array	Memory
memory.random.1K	Random access a 1KB array	Cache
memory.random.1M	Random access a 1MB array	Cache/Memory
memory.random.1G	Random access a 1GB array	Memory
stream.add	Add two vectors	Memory
stream.copy	Copy from one vector to another	Memory
stream.scale	Multiply a vectors with a scalar	Memory
stream.triad	Combination of add and scale	Memory
iobench.read.4M	Read random 4MB blocks from a file	IO
iobench.read.64M	Read random 64MB blocks from a file	IO
iobench.read.1G	Read random 1GB blocks from a file	IO
iobench.write.4M	Write random 4MB blocks to a file	IO
iobench.write.64M	Write random 64MB blocks to a file	IO
iobench.write.1G	Write random 1GB blocks to a file	IO
metadata.create	Create a thousand files	IO (metadata)
metadata.delete	Delete a thousand files	IO (metadata)

TABLE II: Performance Prediction Models.

Model	Description
LINEAR	Ordinary least square regression [4].
CART	Classification and regression tree [12].
SVM	Support vector regression machine [9].
GBM	Gradient boosting machine [10].

applications executing on the same core is different from different cores on a socket, different sockets on a motherboard, etc. For each type of co-location we run the given application and run the measurement benchmarks on the core(s) where we expect the co-running application to execute. This quantifies the execution environment that the given application induces on another application that runs on those compute resources.

IV. EXPERIMENT

We evaluate the effectiveness of KONGMING for two prediction scenarios:

a) Single-task prediction: : In the first scenario a user submits a task to run on a cloud and the cloud scheduler must map this task to some concrete system. Different systems may be built using different hardware (e.g. cache size), their hardware and software may be configured differently (e.g. power level or software cache size) and they may currently be executing different sets of applications. To choose the system to which this task should be mapped to achieve high efficiency and satisfy quality of service constraints the scheduler must predict how long it will run when executed on each of the available systems.

b) Task-pair prediction: : In the second scenario the scheduler is presented with a set of tasks that must be mapped to the available systems. Since it is mapping multiple tasks to the same system is very likely to improve overall efficiency the scheduler needs to know the impact of co-scheduling two tasks on the same system. The prediction task is thus, given two applications to predict their execution time when executing concurrently on the same system.

Our experiments focused on the following experimental platforms. EC2 corresponds systems within the Amazon EC2 cloud, running code in m1.small virtual machine instances.

These instances provide the computing power of approximately 1 GHz 2007 Xeon processor, with 1.7GB RAM and 160GB disk space. Cluster corresponds to systems in a dedicated cluster **Specs ???**. The applications we used in our experiments come from the SPEC2006 [5] and YCSB [7] benchmark suites. Applications in SPEC2006 are designed to test CPU performance and are primarily sensitive to contention for cache and memory. We selected from the overall set of SPEC2006 benchmarks a subset that includes both batch and latency-sensitive applications from YCSB provides a set of workloads to evaluate major key-value store systems used in the cloud, Cassandra, MongoDB and Voldemort. It are sensitive to contention for I/O resources such as disks and file systems. The complete set of benchmarks is detailed in Table III. The SPEC2006 benchmarks were executed on their test-class inputs and the YCSB benchmarks were run with 10,000 keys.

A. Single-task prediction

In this section, we evaluate the accuracy of our prediction models for the case where we wish to predict the execution time of a single task when it runs inside an EC2 instance that shares physical resources with other instances running on the same physical node. To collect data for training and evaluation we repeatedly executed each target application and all the measurement benchmarks (executed in sequence, not concurrently) on 10 different small Amazon EC2 instances for blocks of 24 hours, recording the execution times of each run of the target application and measurement benchmarks. The execution environment on EC2 varies naturally across different systems (different configurations of hardware) and time (different applications are co-located with our benchmark). As such, these experiments do not use our interference threads to create additional load.

Show some highlights of performance variability

We evaluate the accuracy of KONGMING’s predictions using cross-validation [12], where a model is trained on each application’s observations from 9 EC2 instances and used to predict its execution times on the remaining instance. This procedure was repeated 10 times to predict application execution times for each system and we report the average error of these predictions. Cross-validation provides a reliable estimation of model performance and can be used to prevent overfitting the model to the given dataset. Each observation used for training includes the execution times of the measurement benchmarks executed immediately before an application run, as well as the execution time of the application itself. Observations used for prediction only contained the execution times of the measurement benchmarks and the KONGMING model was used to predict the corresponding application execution time.

The distribution of prediction accuracy is plotted for each target application in Figure 1.

The large number of measurement benchmarks KONGMING uses enables it to account for the influence from many different aspects of a system on an application’s performance. However, since typical application’s performance is typically bottlenecked by a few system resources, its execution time will only be affected by the performance of and contention for only these resources and unaffected by the others. To better

Name	Description
Cassandra [1]	Apache Cassandra is an open source distributed database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.
MongoDB [3]	MongoDB is a cross-platform document-oriented database system.
Voldemort [6]	Voldemort is a distributed data store that is designed as a key-value store used by LinkedIn for high-scalability storage.

TABLE III: Storage applications from the YCSB benchmark suite.

TABLE IV: Median and inter-quartile range of percentage error for single-task predictions on EC2.

	MEAN	LM	LM.GA	CART	CART.GA	SVM	GBM
cassandra	67.3 / 5.2	14.5 / 16.4	72.4 / 131.7	9.1 / 14.6	7 / 3.8	24 / 32.3	34.8 / 36.8
mongodb	29.2 / 30.8	6.2 / 8.7	15.2 / 13.2	5.5 / 8.4	20.3 / 10.8	6.6 / 8.8	6.7 / 10.8
voldemort	6.9 / 6.7	5.2 / 8.7	14.8 / 14.9	6.1 / 9.6	3.8 / 5.7	5.8 / 8.3	5.5 / 8.7
spec.GemsFDTD	8.2 / 2.6	2.1 / 2.9	1.8 / 2	2 / 3.1	1.1 / 1.3	2.3 / 3.6	2 / 2.7
spec.astar	0.3 / 1.2	0.8 / 1.1	0.4 / 0.7	0.9 / 1.2	0.8 / 0.6	0.9 / 1.1	0.8 / 1
spec.bwaves	3.8 / 1	1.4 / 1.9	0.7 / 0.7	1.3 / 1.4	1 / 1	1.3 / 1.7	0.9 / 1.3
spec.bzip2	6.9 / 2.9	2.9 / 3.9	0.8 / 1	4.5 / 4.8	0.7 / 1	3.5 / 4.5	3.5 / 4.1
spec.cactusADM	4 / 4.9	2.5 / 3.2	1.1 / 1.4	3 / 4.6	2.2 / 2.5	2.4 / 3.6	3.5 / 3.8
spec.calculix	11.1 / 12	15.8 / 20.7	9.8 / 6.7	15.9 / 21.3	10.4 / 11.6	12.7 / 17.1	20 / 24.3
spec.dealII	0.3 / 0.5	0.5 / 0.7	0.3 / 0.3	0.7 / 0.8	0.4 / 0.3	0.5 / 0.7	0.6 / 0.8
spec.gamess	6 / 5	3.4 / 4	11.4 / 9.7	3.2 / 6.9	4.6 / 8.1	2.8 / 4	4.5 / 3.6
spec.gcc	1.8 / 1.6	1.6 / 2.1	0.8 / 1.3	1.8 / 2.5	1.2 / 1.8	1.7 / 1.9	1.7 / 2
spec.gobmk	0.5 / 1	0.9 / 1.3	0.5 / 0.4	1.3 / 1.2	0.9 / 0.9	1.1 / 1.5	1.1 / 1.2
spec.gromacs	3.1 / 2.5	1.4 / 1.7	1.1 / 1.1	1.5 / 1.8	1 / 0.7	1.6 / 1.7	1.4 / 1.8
spec.h264ref	0.4 / 0.6	0.9 / 1.3	0.5 / 0.3	0.9 / 1.3	0.5 / 0.5	0.8 / 1.2	0.8 / 1.1
spec.hmmmer	0.6 / 0.7	0.6 / 0.9	0.3 / 0.4	0.8 / 1	0.5 / 0.4	0.7 / 0.8	0.7 / 0.7
spec.lbm	1.3 / 1	1 / 1.6	0.8 / 1.1	1.6 / 1.9	1.1 / 1	1 / 1.4	1.1 / 1.5
spec.leslie3d	0.8 / 0.6	0.8 / 1.2	0.7 / 0.6	1 / 1.3	0.7 / 1.9	0.8 / 1.2	0.9 / 1.1
spec.libquantum	1.8 / 4.4	2.7 / 3.7	1.7 / 4.3	3.2 / 4.1	2.5 / 2.3	2.8 / 3.5	2.3 / 3.7
spec.mcf	2.7 / 1.9	2.2 / 2.6	0.9 / 0.9	2.1 / 2.9	1.2 / 0.8	2.1 / 2.6	2.2 / 2.5
spec.milc	4.1 / 2.9	2 / 2.7	1.1 / 1.5	2.6 / 2.9	1.3 / 0.9	2 / 2.4	2 / 2.3
spec.namd	2.3 / 1.8	0.4 / 0.6	0.4 / 0.6	0.5 / 0.8	0.8 / 1	0.5 / 0.6	0.5 / 0.6
spec.omnetpp	1.6 / 1.5	1.9 / 2.6	1 / 1.3	1.8 / 2.6	1.2 / 0.9	1.5 / 1.9	1.7 / 2.3
spec.perlbench	0.9 / 1.2	1.1 / 1.7	0.5 / 0.8	1.5 / 2.9	0.8 / 0.6	1.2 / 1.4	1.4 / 2.6
spec.povray	1.4 / 1.4	1.4 / 2.1	0.9 / 1	1.5 / 2.1	1 / 1.2	1.4 / 2.1	1.4 / 1.9
spec.sjeng	8.5 / 0.7	1.3 / 1.8	0.6 / 1	1.4 / 2	4.1 / 0.8	1.6 / 1.8	1.2 / 1.7
spec.soplex	14.4 / 20	11.6 / 17.3	14.4 / 16.1	17.9 / 25.9	17.3 / 21.1	11.5 / 16.8	11.1 / 15.7
spec.sphinx3	3.9 / 4.4	3.1 / 3.5	2.6 / 2.9	3.6 / 4.9	2.1 / 2.5	2.9 / 3.2	3.4 / 3
spec.tonto	3.8 / 1.8	1.7 / 2.2	1.4 / 2.4	2.2 / 6.6	1 / 1.2	1.7 / 2.2	3.1 / 2.8
spec.wrf	2.2 / 1.1	0.8 / 1.1	0.3 / 0.3	0.9 / 1.3	0.3 / 0.7	0.9 / 2	0.8 / 1.1
spec.xalanbmk	1.1 / 2.9	1.8 / 2	1.5 / 1.4	1.6 / 2.3	1.3 / 2	1.7 / 1.7	2.2 / 2.4
spec.zeusmp	0.9 / 1.6	1 / 1.1	0.4 / 0.4	1.2 / 1.3	0.8 / 0.8	0.9 / 1.2	0.9 / 1.1
geo.mean	2.6 / 2.2	1.8 / 2.5	1.4 / 1.6	2.1 / 2.9	1.5 / 1.5	1.9 / 2.5	2.0 / 2.5

understand the factors that control the behavior of our target applications we attempted to identify the measurement benchmarks and interference threads that are most important for accurately predicting each application’s execution time. The choice of key benchmarks helps to highlight each application’s bottleneck resources and operations and can help developers improve application performance or help them identify the types of systems (e.g. fast CPU vs large cache vs SSD storage) that are most suited for their applications. This was done by running a genetic algorithm [15] to identify the subset of micro-benchmarks that should be used as measurement benchmarks and interference threads to produce an accurate model

for each application. The state of the genetic algorithm was set to be a bit vector with a single bit for each measurement benchmark and interference thread. For a given bit vector we trained the GLM and CART models on the selected set of observations (choice of interference thread) and features of these observations (choice of measurement benchmark). We then used the resulting model to predict the execution times of this application, as above. The genetic algorithm implementation in the R `genalg` package was then used to find the bit vector that produced high accuracy with these models with a small number of terms, using vector crossover and mutation operations.

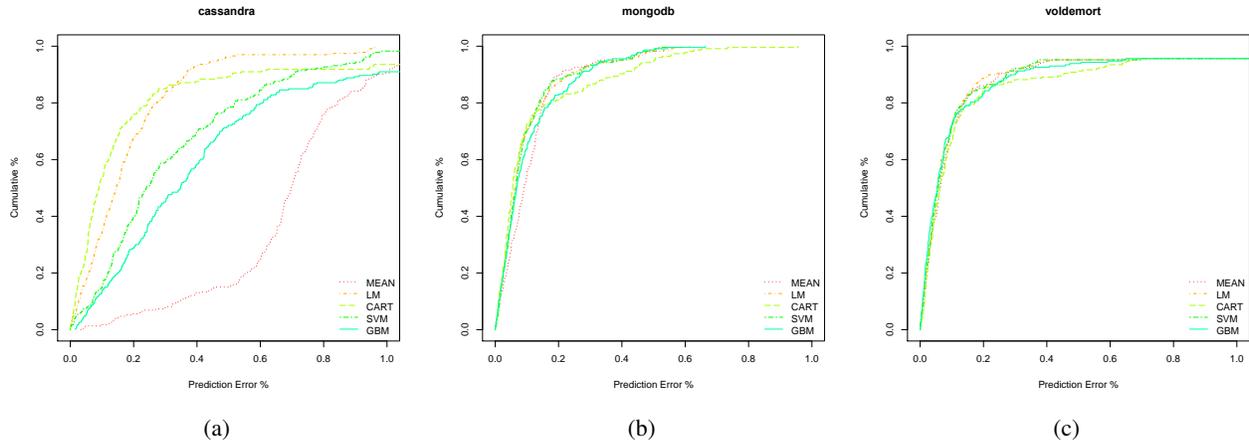


Fig. 1: CDF of prediction accuracy on EC2 estimated by 10-fold cross validation for YCSB applications.

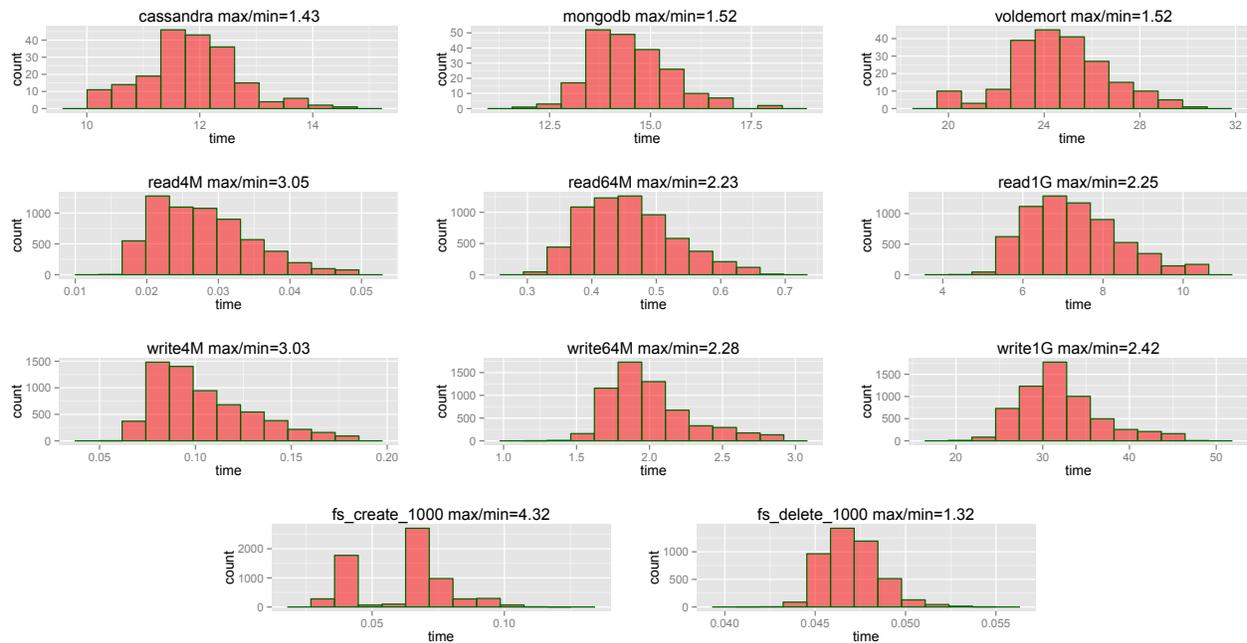


Fig. 2: Histograms of execution times of the I/O-intensive YCSB and measurement benchmarks on EC2. The title of each plot displays the name and the ratio between the longest and the shortest execution times of each benchmark.

Figure 2 shows

B. Task-pair prediction

This section evaluates KONGMING’s accuracy for the use-case where the execution time of two applications that are co-located on the same system. Algorithm 1 shows how the training data for KONGMING is collected. In the first phase we repeatedly execute each application and the measurement benchmarks on an otherwise unloaded system in Cluster. During each iteration we execute in the background one or more copies of a single interference benchmark to induce a specific amount of load on both the application and measurement benchmark, for a total of 10 observations for

each combination of application and interference thread. This produces a heterogeneous set of execution environments that enables KONGMING to train a robust model that accounts for a wide range of application behaviors. In the second phase we repeatedly run each application on one core while on another core we execute and time each measurement benchmark. The resulting execution times quantify the execution environment induced by A on applications that concurrently execute on a different core, sharing the disk and lower levels of the memory hierarchy but not computational resources and L1 cache. We focused on cross-core interference because prior work [?] and own experiments have demonstrated that cloud systems such as EC2 are very effective at isolating performance interference to computation but are less effective at isolating memory

TABLE V: Median and inter-quartile range of percentage error for task-pair predictions on the local cluster.

	MEAN	LM	LM.GA	CART	CART.GA	SVM	GBM
cassandra	117.6 / 7.1	20.9 / 27.3	6.5 / 8.5	13.2 / 4.1	3.1 / 10.8	29.7 / 18.7	38.9 / 134.8
mongodb	6.6 / 6	2.4 / 2.8	2.4 / 2.3	2.1 / 2.1	2.1 / 1.9	2 / 2.3	2.2 / 2.2
voldemort	28.3 / 16.8	7.1 / 11.3	6.3 / 8.2	13.3 / 18.5	5.3 / 9.1	4.4 / 8	22 / 21.1
spec.GemsFDTD	56.9 / 9.3	7 / 8.4	1.7 / 2.3	17.2 / 5.3	1.6 / 2.4	40 / 22.1	5.1 / 9.5
spec.astar	10.4 / 1.2	0.4 / 0.6	0.4 / 0.6	0.4 / 0.6	0.3 / 0.5	0.6 / 0.6	0.5 / 0.6
spec.bwaves	11.1 / 1.8	0.8 / 1	0.8 / 1	0.7 / 0.8	0.7 / 0.9	0.7 / 0.8	0.6 / 0.6
spec.bzip2	11.4 / 40.9	0.9 / 1.1	0.9 / 1.1	1 / 1	0.9 / 1	1.6 / 2.1	1.2 / 1.2
spec.cactusADM	23.4 / 55.5	6.3 / 8	6.7 / 8	5.6 / 8.9	5.3 / 8.2	5.1 / 7.2	5.4 / 6.8
spec.calculix	13.5 / 40.7	9.9 / 14.2	10.2 / 11	9 / 13.7	9.3 / 12	10.3 / 12.1	10.2 / 14.5
spec.dealII	11.6 / 36.6	0.7 / 0.9	0.5 / 0.9	0.6 / 0.4	0.3 / 0.2	0.9 / 1.1	0.4 / 0.6
spec.gamess	7 / 4.5	2.7 / 2.9	1.8 / 2.2	2.3 / 2.3	1.9 / 2.1	2.2 / 2.3	2.2 / 2.6
spec.gcc	11 / 44.9	2.8 / 4.4	2.2 / 4	4.3 / 4.2	1.9 / 3.4	3.5 / 3.8	2.7 / 4
spec.gobmk	13.8 / 4	1.7 / 2.1	1.4 / 1.9	1.8 / 1.9	1.4 / 1.8	1.7 / 2.2	1.3 / 1.7
spec.gromacs	15.9 / 16.5	4.5 / 8.2	5.2 / 7.6	4.7 / 7.6	4.3 / 9	4 / 9	4.9 / 8.7
spec.h264ref	22.5 / 4.4	2.7 / 3.2	2.2 / 2.6	3.2 / 3.2	1.4 / 1.7	3.4 / 3.8	1.5 / 2.4
spec.hammer	5 / 1	0.6 / 0.7	0.4 / 0.6	0.7 / 0.5	0.5 / 0.5	0.8 / 0.7	0.5 / 0.6
spec.lbm	13.1 / 6.3	2.2 / 2.9	1.9 / 2.1	1.5 / 2.6	1.7 / 2.2	2.2 / 2.9	1.9 / 2.5
spec.leslie3d	16.9 / 2.5	1.1 / 1.3	1 / 1.3	0.9 / 1.3	0.8 / 1.4	1.2 / 1.3	1 / 1.4
spec.libquantum	9.1 / 32.7	4.4 / 4.7	3.7 / 3.9	3.9 / 3.1	3.9 / 3.9	4 / 4	4.1 / 7.2
spec.mcf	13 / 12.3	3.6 / 4.8	3.9 / 4.8	4.1 / 4.5	3.4 / 4.4	4.5 / 5.5	3.7 / 5.5
spec.milc	14.3 / 79.4	2.1 / 2	2.2 / 2.8	2.3 / 2.9	1.6 / 2.9	2.6 / 3.3	1.4 / 3.1
spec.namd	17.4 / 1.7	0.7 / 1.1	0.7 / 0.9	0.9 / 0.8	0.9 / 1	1.2 / 1.3	0.6 / 1
spec.omnetpp	1.8 / 5.5	2.6 / 3	1.6 / 1.6	1.4 / 2.3	1.4 / 1.7	2.1 / 2.6	3.1 / 3.7
spec.perlbench	5 / 14.5	1.6 / 2.9	1.9 / 2.4	1.2 / 2.5	1.7 / 2.5	1.2 / 2.8	2 / 2.5
spec.povray	6.5 / 28.8	1.2 / 1.8	1 / 1.1	1 / 1.3	0.9 / 1.3	1.1 / 1.6	1.5 / 1.9
spec.sjeng	17.5 / 52.5	4 / 5.1	3.8 / 5.2	3.7 / 6	3.2 / 4.6	3.4 / 5	3.6 / 3.5
spec.soplex	30.3 / 33.7	15.5 / 19.2	14.3 / 15.8	13.8 / 14.2	14.6 / 16	14.7 / 16.4	13 / 14.9
spec.sphinx3	17.1 / 35.9	1.9 / 2.3	2 / 2.3	2.6 / 2.6	2.2 / 2.1	2.5 / 2.3	2.3 / 2.5
spec.tonto	3.2 / 30.1	4.9 / 5.7	2.2 / 2.7	2.3 / 3.6	1.9 / 2.6	2.5 / 3.9	3 / 4.4
spec.wrf	11 / 42.6	0.6 / 0.8	0.5 / 0.7	0.7 / 1.1	0.5 / 0.4	0.8 / 1	0.6 / 0.7
spec.xalancbmk	14.7 / 59.6	5.3 / 6.4	4.6 / 5.9	4.1 / 6.1	3.9 / 4.3	4.9 / 5.9	4.7 / 5.7
spec.zeusmp	13.8 / 7.1	2 / 3.1	1.9 / 2.5	1.9 / 2.1	1.8 / 2.3	2.2 / 2.7	1.5 / 2.2
geo.mean	12.8 / 12.7	2.5 / 3.2	2.0 / 2.6	2.4 / 2.7	1.8 / 2.3	2.7 / 3.2	2.3 / 3.2

hierarchies, storage and networks.

We evaluate KONGMING’s predictions by executing every pair of applications on a single Cluster system. Given two applications A and B , KONGMING uses the execution times of the measurement benchmarks when executed concurrently with A to predict the execution time of B while being co-executed with A , and vice versa to predict A ’s execution time.

The average prediction accuracy is shown in Figure 3.

C. Efficacy Analysis of Interference Injector with Genetic Algorithm

To accuracy of KONGMING’s predictions depends on choosing a set of micro-benchmarks, both for measurement and interference, that are representative of application and system behaviors. A given benchmark can be instrumental for KONGMING’s ability to accurately predict a given application’s execution time in two ways. First, if a measurement benchmark’s execution is helps to predict the application’s execution time, this means that the operations executed by the benchmark utilize the same hardware and software resources

Algorithm 1 Collecting Training Data

```

1:  $M \leftarrow \{\text{measurement benchmarks}\}$ 
2:  $I \leftarrow \{\text{interference threads}\}$ 
3:  $A \leftarrow \text{target application}$ 
4: while True do
5:   for  $i$  in  $I$  do
6:     run  $i$  in the background
7:     for  $m$  in  $M$  do
8:       run  $m$  and record its running time
9:     end for
10:    run  $A$  and record its running time
11:    stop current execution of  $i$ 
12:   end for
13: end while
14: while True do
15:   run  $A$  on one core
16:   for  $m$  in  $M$  do
17:     run  $m$  on another core and record its running time
18:   end for
19:   stop current execution of  $A$ 
20: end while

```

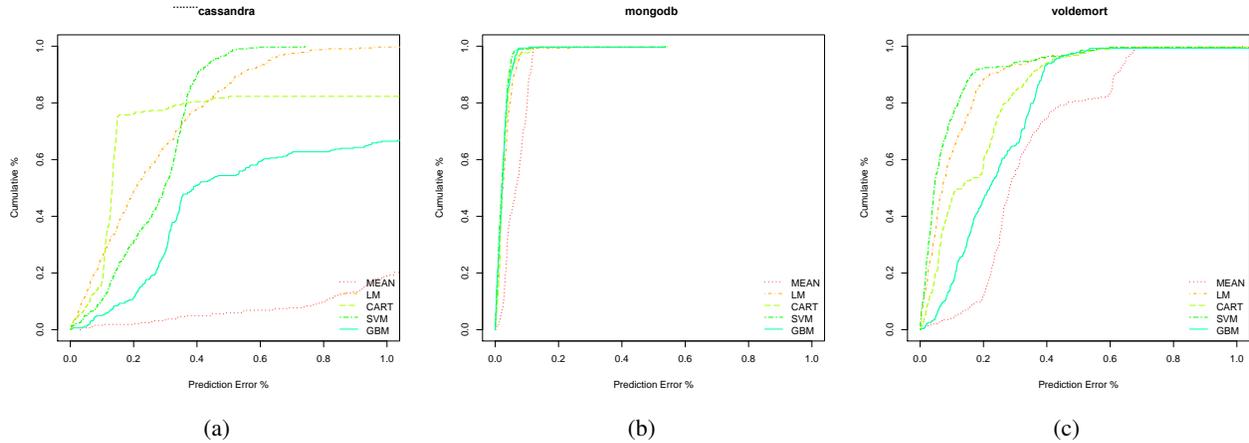


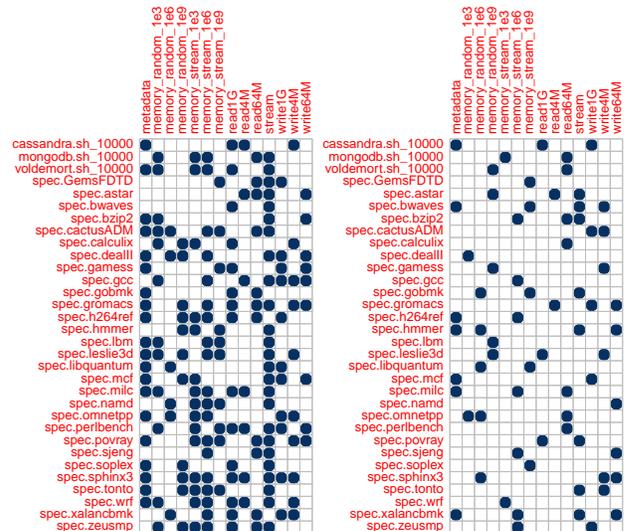
Fig. 3: CDF of prediction accuracy on the local cluster for YCSB applications.

as a major component of the application itself. Similarly, if the inclusion of experimental runs that use a given a benchmark for interference helps to improve KONGMING’s accuracy, this serves as an independent indication that the benchmark uses the same resources as the application. This section analyzes the performance properties of our target applications by considering the benchmarks that were most instrumental in producing accurate KONGMING models when used either as a measurement benchmark or as an interference thread.

To evaluate this question we used a generic algorithm described in Section IV-A to identify the sets of benchmarks that the LM and CART models should use for measurement and interference within their training on the task-pair prediction problem on Cluster. The results are shown in Figure 5, with Figure 5(a) focusing on LM with interference threads, Figure 5(b) focusing on LM with measurement benchmarks and Figures 5(c) and 5(d) focusing on CART with interference and measurement, respectively.

V. RELATED WORK

Cuanta [11] employs a canonical set of synthetic cache loaders (scl), which exhibit distinct access patterns with respect to the sets and ways of last level cache on a given platform, to emulate the effect of interference caused by actual applications. The performance degradation imposed by one scl on another co-running scl is measured and recorded in the interference profile of the former. In a similar manner, the interference profile of an application is obtained and then used to identify its cache clone, the scl that most closely matches its interference profile. This way, Cuanta reduces the prediction of performance degradation caused by potentially unbounded number of applications to a finite set of scl. To predict the performance degradation of an application co-located with N other applications, Cuanta needs to collect the performance degradation table for the application where the slowdown ratio of the application is recorded when running with every possible combination of N scl, and then looks up the degradation table by the cache clones of these co-located applications to find out the actual performance degradation in the application. A

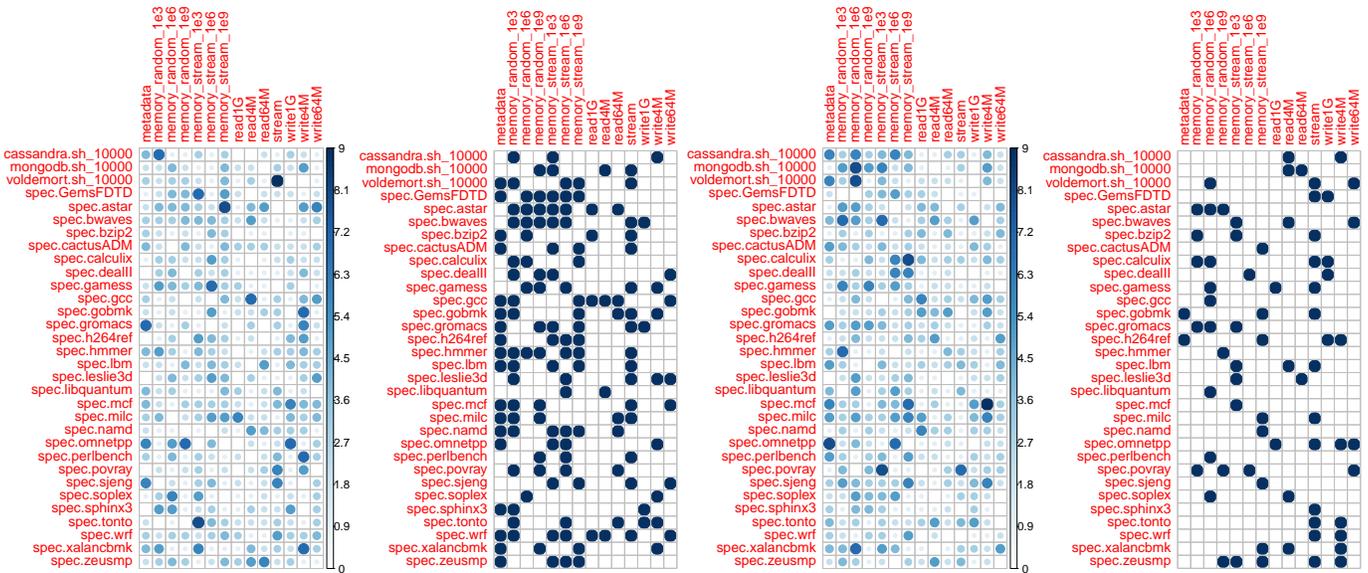


(a) Measurement benchmarks selected by LM.GA. (b) Measurement benchmarks selected by CART.GA.

Fig. 4: Measurement benchmarks selected by GA for the single-application experiments on EC2. Each column corresponds to a measurement benchmark. Each row corresponds to an application. Each cell indicates whether or not the corresponding measurement benchmark is selected for the corresponding application by GA.

key limitation of Cuanta is that it ties its features to hardware parameters, which makes it non-transferrable across platforms. Further, since it only considers memory behaviors, it ignores interactions between contention for different resource types. In contrast KONGMING considers all the major resources of individual systems and because it measures each system’s effective performance is portable across architectures.

Bubble-Up [14] is a profiling-based static approach to predicting the performance degradation of a latency sensitive application caused by interference imposed by a batch appli-



(a) Sources of interference selected by LM.GA. (b) Measurement benchmarks selected by LM.GA. (c) Sources of interference selected by CART.GA. (d) Measurement benchmarks selected by CART.GA.

Fig. 5: Sources of interference and measurement benchmarks selected by GA for the colocation experiments on Cluster. Each column corresponds to a source of interference or a measurement benchmark. Each row corresponds to an application. For source of interference, each cell depicts the number of training runs selected by GA for the particular source of interference. For measurement benchmark, each cell indicates whether or not the corresponding measurement benchmark is selected for the corresponding application by GA.

cation when co-located. Bubble-Up consists of three primary steps. In the first step, the QoS sensitivity of the latency sensitive application is profiled by the *bubble*, a memory subsystem microbenchmark, designed to generate interference in last-level cache and memory bandwidth. The bubble is configured to apply a series of different levels of interference in the memory subsystem and the performance degradation of the latency sensitive application is recorded along with the interference level to form a QoS sensitivity curve. In the second step, the batch application is profiled by a *reporter*, whose QoS sensitivity curve is profiled in advance, and the level of interference introduced by the batch application is obtained by looking up on the reporter’s QoS sensitivity curve by its performance degradation. Finally, the level of interference of the batch application is checked against the QoS sensitivity curve of the latency sensitive application to predict the performance degradation of the latency sensitive application when running concurrently with the batch application. In a follow-up work, Bubble-Flux [17], Bubble-Up is augmented with dynamic bubble and continuous online QoS management to handle load fluctuation of latency sensitive applications. Like Cuanta, Bubble-Up focuses only on memory utilization, ignoring the possibility that applications may interfere with or be bottlenecked on multiple resources at the same time.

Paragon [8] formulates the task of performance degradation prediction of applications running on datacenter platforms as a collaborative filtering problem where the scores of sensitivity and tolerance for interference are learned for every pair of application and source of interference and recorded in two utility matrices. The utility matrices are bootstrapped in an

offline step where a few tens of applications are profiled by running concurrently with all microbenchmarks representing sources of interference. In its online mode, a new application is profiled against two randomly selected microbenchmarks and the scores for the remaining sources of interference are estimated using a regularized SVD method [2, 16] invented by Simon Funk for the Netflix Prize. To make predictions Paragon must run microbenchmarks on the systems for which prediction must be made. In contrast, KONGMING must only be trained on a representative set of execution environments, which may be emulated on a single architecture or even a single system.

Zhao et al [18] predict the performance degradation caused by cross-core interference from aggregated L2 cache misses and memory throughput using a piecewise regression model. They notice that the performance of applications co-located on a multicore processor is significantly dependent on the aggregated pressure on the dominant contention factor, *i.e.*, the shared resource that largely determines the severity of performance degradation. On a system of multiple different types of resources, the dominant contention factor for an application varies in response to the composition of co-located application, hence the piecewise model to improve the accuracy of prediction over the full range of different contention factors. They expect users to specify a set of domain partitioning candidates for model training, conduct model selection to choose the best domain partitioning using a set of randomly selected applications for each platform, then estimate the coefficients of the best model for each application. In contrast, KONGMING automatically employs

a fixed set of representative microbenchmarks to measure arbitrary systems and applications and induce a broad set of execution environments.

VI. CONCLUSION

This paper presents KONGMING, an approach for modeling the behavior of applications on complex architectures that is able to predict the execution times of applications in a variety of system contexts. KONGMING is sensitive to properties of the hardware and software on which an application may execute, as well as contention for a wide range of resources from other applications, including computation, the memory hierarchy, disk as well as file system metadata access. We have demonstrated in this paper that KONGMING can predict how long an application may run on a given system without a priori knowledge of the system's hardware configuration or the applications currently executing on it. Further, for a given system KONGMING can predict how long a pair of applications will run when executed concurrently, accounting for any contention they may have for shared resources.

We have demonstrated the effectiveness of KONGMING on both EC2 virtual instances as well as real systems on a dedicated cluster, showing that KONGMING makes predicts application execution time consistently accurately. Further, we have shown how the accuracy KONGMING relates to the choice of measurement benchmarks and interference threads used in its training, illustrating the fact that a given application's behavior depends on many different resources, which must be considered to make accurate predictions. In our future work we plan to employ KONGMING's predictions as part of a general purpose workload scheduler that can efficiently share resources among complex heterogeneous workloads on various hardware platforms.

REFERENCES

- [1] <http://cassandra.apache.org/>.
- [2] <http://sifter.org/~simon/journal/20061211.html>.
- [3] <http://www.mongodb.org/>.
- [4] http://en.wikipedia.org/wiki/Ordinary_least_squares.
- [5] <http://www.spec.org/cpu2006/>.
- [6] <http://www.project-voldemort.com/voldemort/>.
- [7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.
- [8] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.
- [9] H. Drucker, C. J. C. Burges, L. Kaufman, A. J. Smola, and V. N. Vapnik. Support vector regression machines. In *NIPS*, 1996.
- [10] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):pp. 1189–1232, 2001. ISSN 00905364. URL <http://www.jstor.org/stable/2699986>.
- [11] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *SOCC*, 2011.
- [12] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction 2nd Edition*. Springer, 2009.
- [13] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.
- [14] 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-44*, 2011.
- [15] L. M. Schmitt. Theory of genetic algorithms. *Theoretical Computer Science*, 259(1 - 2):1 – 61, 2001.
- [16] G. Takács, I. Pilászy, B. Németh, and D. Tikk. Scalable collaborative filtering approaches for large recommender systems. *J. Mach. Learn. Res.*, 10:623–656, June 2009. ISSN 1532-4435. URL <http://dl.acm.org/citation.cfm?id=1577069.1577091>.
- [17] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA*, 2013.
- [18] J. Zhao, H. Cui, J. Xue, X. Feng, Y. Yan, and W. Yang. An empirical model for predicting cross-core performance interference on multicore processors. In *PACT*, 2013.