2015 IEEE International Parallel and Distributed Processing Symposium Workshop

A Methodology for Co-Location Aware Application Performance Modeling in Multicore Computing

Daniel Dauwe[1], Eric Jonardi[1], Ryan Friese[1],
Sudeep Pasricha[1,2], Anthony A. Maciejewski[1], David A. Bader[3], and Howard Jay Siegel[1,2]
[1]Department of Electrical and Computer Engineering
[2]Department of Computer Science
Colorado State University
Fort Collins, CO, 80523
[3]College of Computing
Georgia Institute of Technology
Atlanta, GA, 30332

***Abstract*—As multicore processor architectures are now prevalent in server nodes of parallel and distributed computing systems, it has become important to characterize the performance of applications run on these architectures. This study investigates the performance degradation an application experiences from memory interference due to other applications co-located on cores of the same multicore processor. We propose a methodology for designing models that are capable of utilizing varying amounts of information relating to an application and its co-located applications to predict the application's execution time performance degradation due to co-location. We evaluate the models using several application co-location scenarios based on real world test data from two scientific benchmark suites on two server class Intel Xeon multicore processors.**

Keywords: performance modeling; resource management; memory interference; application co-location; benchmarking; multicore processors

I. Introduction

There is an inherent trade-off in large scale computer systems between reducing the use of system resources by consolidating applications to as few server processor nodes as possible (to reduce system power) and the performance degradation that occurs in these applications as a result of sharing system resources with other applications. Memory interference caused by multiple applications co-located on a multicore processor has been shown to negatively impact application performance (e.g. [ChD14], [DaF14], [SaS13], [TaM11]). Specifically, sharing of system resources such as DRAM and the last-level cache by co-located applications creates contention and increases the memory intensity of all applications running on the multicore processor [DaF14]. This increase in memory intensity results in a corresponding increase in average memory access time, which ultimately contributes to an increase in the application's overall execution time. This increase in execution time is significant, and in some cases can as much as double or triple the execution time of an application as compared to its baseline execution time [MeS10].

The use of multicore processors that experience performance degradation due to co-location is pervasive throughout many kinds of computing systems, but its effects are most likely to be prevalent in large-scale server systems and high-performance computers. In these kinds of computing systems, highly parallel applications running on multicore processors result in a sharing of resources in a manner that creates memory interference and causes performance degradation. Having a methodology that is capable of predicting how well a system will run in a particular co-location scenario would be very useful for these systems. For example, the information gained from accurate co-location performance degradation could be integrated into intelligent application scheduling, and may lead to system performance improvement by more fully utilizing hardware and thereby increasing opportunities for server consolidation to save power while still maintaining quality of service constraints. **The work presented in this paper provides a methodology that can be used to create co-location aware performance models.**

The methodology for analyzing system performance that is described in this work is general enough to be applicable to any set of applications running on any multicore processor. Once application performance information for a particular combination of multicore processor and target applications has been collected, our methodology uses machine learning techniques to construct performance models characterizing that performance information. Once trained, these models require only a single serial baseline measurement of parameters for each application running alone in the system to make predictions about the performance degradation from

978-1-4673-7684-6/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPSW.2015.38

memory interference that will occur when the application is executing with different types of co-located applications. While it has been shown in [SaS13] that an application's use of memory resources operates in varying phases across its execution, this work shows that going into such a level of detail is not necessary to make accurate predictions.

After describing how our methodology operates, our work validates the theory behind the proposed methodology using real world data collected on two Intel Xeon server-class machines running a collection of scientific application workloads, with some models performing with 98% accuracy. In addition to creating and demonstrating a methodology that is capable of being ported across processor architectures, this work provides insight into what memory use information is most important to know about a set of applications running in a system.

This work makes the following key contributions: (a) we identify factors that can characterize slowdown during application co-location scenarios, and methods to capture those factors; (b) we propose a novel methodology to integrate these factors into multi-granularity and multi-fidelity performance models that can be used to predict application performance under co-location scenarios; (c) we show that a fine level of detail is not always necessary to achieve reasonable prediction accuracy; and (d) we validate the methodology with real-world data obtained from running co-located scientific workloads on contemporary Intel Xeon server-scale multicore processors with up to 12 cores per processor.

The rest of the paper is organized as follows. The next section discusses related work in this area. Our prediction modeling methodology is presented in Section III. Section IV describes details of the testing environment and data collection. Experimental results validating the models are shown in Section V. The paper concludes with a discussion of practical applications that can be taken from the experimental results in Section VI.

II. Related Work

Several works have explored the impact of co-location on application performance in multicore environments. Here we briefly summarize the most relevant prior works in this area.

The authors in [TaM11] give an early examination of how co-locating multiple applications on a single multicore processor affects performance. However, their work focuses on a general examination of the effects that co-location has on the system as a whole, and does not examine the effects on specific applications or create co-location performance models the way that our work does.

The study in [SaS13] gives an excellent review of how the architecture that an application is run on can affect the cache use and memory intensity of that application. The paper however does not attempt to make predictions about performance degradation as we do, but it does show the importance of including memory intensity and cache usage information when characterizing performance degradation in the presence of application co-location.

Our work in [DaF14] measures memory interference from application co-location, and its impact on system performance for a single Intel i7 machine. However that work does not create models that predict system performance, and the scope of the work is restricted to only a single consumer class machine.

The work in [MaT11] describes the challenges faced by applications sharing resources, and the need for being able to perform precise predictions of performance degradation. The paper presents its "Bubble-Up" methodology for predicting performance degradation results. However, it does not consider the impact of dynamic voltage and frequency scaling on application performance, and also does not collect experimental data or characterize the memory interference effect of having more than two applications co-located.

The authors in [ChD14] present an extension to the "energy roofline" model that explores the effect of memory intensity (from the perspective of arithmetic intensity) on execution time and power use. The study runs a series of constructed microbenchmarks on twelve machine architectures and provides an analysis of the performance of the systems. While this study collects data about performance degradation from memory interference on a set of real machines, it uses small "microbenchmark" tests on these machines, as opposed to the scientific workloads we use. Moreover their work does not create models to predict performance due to memory interference.

Similar to our work, the work in [DwF12] also looks at creating a portable methodology using machine learning techniques for predicting application performance degradation from shared resources. The authors in this paper also incorporate shared resources beyond the last-level cache. However the addition of incorporating these resources forces the resulting model to be extremely complicated, and their model requires constantly monitoring a large number of processor performance counters, which can cause system-wide slowdown for all running applications. In contrast, our methodology needs to collect performance counter information about each application only a single time, and provides better prediction performance. Additionally, our methodology guarantees a uniform selection of training data over the possible co-location space (allowing for more portability) while the work from these authors selects the vast majority of its training data at random.

The authors acknowledge that work exploring the effect of hyperthreading (SMT) on application performance is an open and active area of research. Papers such as [DeK06] and [PaE00] examine scheduling and resource use of application utilizing SMT. We chose to focus our study on the interference that applications experience at an inter-core granularity, and for this study we have turned off hyperthreading to

remove the possibility of application interference in the L1 cache.

III. Modeling Methodology

A. Overview

Our work uses two types of machine learning techniques, *linear modeling* and *neural networks*, for constructing the predictive models. These techniques have been used in prior work [DwF12], [MaT11] but were limited in attribute selection and scope. For each machine learning technique, we design several models with varying levels of complexity and application features.

B. Model Features

The performance prediction models that we design use up to eight separate features to predict how the target application performance is impacted by co-located applications. The eight features were chosen by performing a principal component analysis (PCA) on the data collected from multicore processors considered in this work. PCA allows all of the features that were gathered to be ranked according to variance of their output, giving an idea of which features were most important to include in the models.

These features are a general set that are present in most multicore processors. We have constructed models that use increasingly complicated combinations of the features. These range from combinations we felt would be most easily available to a resource manager making allocation decisions, to combinations that required more information about the application's baseline information to construct. The eight features are shown in Table I. The table gives the name of the feature in the first column, and the aspect of the processor that it measures in the second column.

The features of Table I can be combined to create models of various complexities. The "target" application in the table is the one for which we are interested in determining slowdown due to co-location. The baseline execution time seen in Table I is the execution time of the target application without any co-location present. The model feature sets listed in Table II represent six possible scenarios, one baseline scenario (model "A") that uses only the *baseExTime* feature for predictions and five other scenarios. For each of the five other scenarios, the resource management system has a certain amount of baseline information about the system, the target application, and the other applications co-located on the system. The progression from one model to the next simulates a realistic process where the resource management system progressively obtains more detailed information about the system and the executing applications.

C. Linear Model

To predict the impact of application performance due to co-location, six linear models were developed using the six feature sets listed in Table II. Each linear model is the sum of the products of the utilized features and the model coefficients determined during training, plus a constant. A general model for predicting co-located execution time using N features would take the form of Equation 1. Linear regression is used to calculate the values for the coefficients using the linear least squares function in the Python package SciPy.

$$co\text{-}located\ execution\ time = \sum_{i=1}^{N}(coefficient_i * feature_i) + constant \quad (1)$$

D. Neural Network Model

From observation of the raw data, there are obvious instances of nonlinearity in a few of the features of our data. This was the primary motivation for attempting to create a prediction model using a neural network that can capture non-linearity effects. Neural networks [Bis06] are a machine learning technique that is commonly used when making predictive models. The approach is inspired by attempting to mimic how the human brain is thought to work by defining a set of "neurons" that are used as nodes for the system. The inputs to these neurons are propagated through the network via a series of functions located at each node in the network. The final output value is determined by the input value's propagation through the network. For our model the input neurons are the features of the data available in each model, and the outputs determine the predicted execution time with performance degradation that the model will experience with co-location. The neural networks used in this work vary in the number of nodes used from ten to twenty depending on the model feature set that is used as inputs to the network. A *scaled conjugate gradient* numerical method was used to determine the co-efficient values at each network node.

E. Model Accuracy

All of the models are evaluated using Mean Percentage Error and Normalized Root Mean Squared Error as two ways of comparing the predicted application execution time for each test ($predicted_j$) to the actual execution time for each test ($actual_j$).

1) Mean Percent Error: Mean Percent Error (MPE) is defined in Equation 2. The magnitudes of the actual values within the data vary greatly (e.g. when modeling execution time, actual values could range from as little as 150 seconds to over 1000 seconds based on the application that is being executed and the state of co-location of the applications in the system), and MPE allows the evaluation of prediction accuracy independent of these magnitudes for each of the M sample points of data. This error value is taken for just the target application's execution time.

Table I: Model Features

Feature name	aspect of execution measured
baseExTime	baseline execution time of target application at all P-states
numCoApp	number of co-located applications
coAppMem	sum of co-application memory intensities
targetMem	target application memory intensity
coAppCM/CA	sum of co-application last-level cache misses/cache accesses
coAppCA/INS	sum of co-application last-level cache accesses/instructions
targetCM/CA	target application last-level cache misses/cache accesses
targetCA/INS	target application last-level cache accesses/instructions

Table II: Sets of Model Feature Groups

Set name	feature groups within set
A	baseExTime
B	model A + NumCoApp
C	model B + coAppMem
D	model C + targetMem
E	model D + coAppTCM/TCA, coAppTCA/INS
F	model E + targetTCM/TCA, targetTCA/INS

$$MPE = 100 * \frac{1}{M} \sum_{j=1}^{M} |\frac{predicted_j - actual_j}{actual_j}| \quad (2)$$

2) Normalized Root Mean Squared Error: Normalized Root Mean Squared Error (NRMSE) gives an indication of the variance of our predicted values from the actual values. For M sample points, NRMSE provides a ratio of Root Mean Squared Error and the interval of values that the actual data can take ($actual_{max} - actual_{min}$). Normalized root mean squared error is defined in Equation 3.

$$NRMSE = \frac{100}{M} * \frac{\sqrt{\sum_{j=1}^{M} (\frac{predicted_j - actual_j}{actual_j})^2}}{actual_{max} - actual_{min}} \quad (3)$$

IV. Implementation Overview

A. Testing Environment

1) Operating System: One of the objectives of this research was to design a methodology that could be applied to a wide variety of computing systems. Our testing environment was designed to be portable across many multicore processor architectures to allow for simplicity of gathering test data and ease of recreating the testing environment for future users of this work. To ensure accurate data is collected, the testing environment is run from a "lightweight" command line version of the Ubuntu 14.04 operating system [Can12] installed on a USB drive. This is done to minimize the effect that the operating system has on application execution. Unessential OS utilities and kernel daemons were removed so that the applications being monitored suffer as little interference from unpredicted events in the OS as possible. Such an environment mimics a large-scale computing platform meant to execute multiple parallel jobs.

2) Processor Performance Counters: Modern multicore processors provide the ability for developers to monitor hardware events that occur inside a multicore processor during the execution of an application. Through the use of specialized "performance counters" present in the processor it is possible to track the number of occurrences of certain events that take place, such as the number of instructions executed or last-level cache misses. These performance counters are architecture dependent, and due to differences between microarchitectures the number and types of performance counters that are available to the system are not consistent (e.g. differences across [Int14a], [Int14b], and [Int14c]). Given the design goal of having portability for our methodology, interfacing directly with these hardware performance counters is not a valid option. Therefore, the testing environment makes use of two tools to facilitate interactions with the hardware.

The first tool is the "Performance Application Programming Interface" (PAPI) [Pap14], an API that was made specifically to provide portability when accessing performance counters across different architectures. PAPI has created a general list of more than 100 standard performance counter "presets" that are likely to be present in a modern processor. PAPI has made it more accessible to interface with these counters across architectures.

The second tool our testing environment utilizes is the HPC toolkit [Htk14]. This suite of applications interface with PAPI and make it easier to monitor and collect information from multiple performance counters in the system. Specifically, HPC toolkit's "hpcrun-flat" application profiler is used to collect performance counter information because it is able to run with very low overhead.

3) Measuring Cache Use: From [DaF14], it is known that applications that need to access data from memory more often experience a larger amount of performance degradation due to co-location. We incorporate these perfor-

mance degradations into our prediction models by collecting measurements of these effects. We have found that three hardware performance counter measurements can be used to collect the information necessary for deriving the metrics used in our methodology's models:

- number of last-level cache misses an application experiences (LLC) representing the number of times an application must go to main memory
- number of instructions the application executes (NI)
- total number of last-level cache accesses the application attempts (TCA)

Measured features derived from these measurements were listed in Section III. It should be noted that "last-level" cache misses and accesses are dependent on architecture, and can refer to either the L2 or L3 cache depending on the multicore processor that is being used. It is also important to note that when collecting test results for the execution of applications the values measured in these performance counters suffer a loss of temporal information, so they can only represent an average value across time.

One notable metric derived from this data is application memory intensity. Memory intensity is defined to be a ratio of an application's last-level cache misses to the number of instructions executed by that application. This metric gives an idea of the rate at which an application needs to go to main memory to fetch data. It is useful because it shows whether an application's execution will be more likely to be memory bound relative to another application, meaning that its performance depends more on its memory access speed rather than its computational speed. Memory intensity also gives some idea of how much an application tends to access memory. A highly memory intensive application will utilize the shared cache resource more, and therefore it will tend to affect, and be more affected, by the memory interference from other applications.

4) Processor Performance States (P-states): Processor performance states (P-states) are a set of discrete voltage and frequency values in which a multicore processor can operate. P-states utilize dynamic voltage and frequency scaling (DVFS), supported in all contemporary multicore processors. DVFS techniques can reduce the dynamic operating power of a multicore processor to consume less power or to temporarily reduce the operating temperature due to the multicore processor having exceeded a thermal cut-off. However, these benefits come at the cost of having to throttle the multicore processor speed by decreasing the clock frequency. This generally increases the execution time (decreasing system performance) of any application running on the multicore processor. The range and number of P-state frequencies that are available in a system are highly dependent on the architecture of the multicore processor. Processor P-states are likely to change in high performance computing systems based on the system's need to reduce power or temperature. This work focuses only on how changing P-states affects application execution time. This effect is taken into account through knowledge of the baseline execution time of each application at a given P-state.

B. Data Collection and Experimental Setup

1) Benchmark Applications: The applications run as testing workloads for our model validation were taken from two scientific benchmark suites. The set of eleven applications that we considered varied in the types of tasks that they perform and are characterized by a wide spread of memory intensity values. Table III shows the applications, those taken from the PARSEC benchmark suite [Par14] are denoted with *(P)*, and those from the NAS benchmark suite [Nas14] are denoted with *(N)*. The table also shows the application's associated baseline memory intensity values, where baseline memory intensity values are measured when the applications are executed on a multicore processor by themselves without interference caused by co-location.

As shown in Table III, these applications have been categorized into four memory intensity classes denoted as "Class I" through "Class IV." Class I applications are the most memory intensive applications (meaning that they have the highest number of last-level cache misses per number of instructions executed and are more memory bound), while class IV applications are the least memory intensive (meaning that they experience fewer last-level cache misses per number of instructions executed, and their execution is more CPU bound). Categorizing the applications into groups in this way allows applications from particular groups to be referred to more generally. These groupings allow for more broad use of this methodology for performance prediction. Should a system developer not have detailed memory intensity information about the applications running in the system, but still has a general idea of how memory intensive the applications might be, then having application class values will allow the developer to still be able to use the model. The developer can still gain some insight as to the expected performance of the system by running the model with average values for that application's class.

It should be noted that the memory intensity values listed in Table III are from baseline measurements for one specific system. We found that the memory intensity values do not vary widely between the machines we tested, thus we used the memory intensity classes to accurately represent class categories for the Xeon family of multicore processors we consider. It is also important to note that the memory intensity values between application classes tend to differ by orders of magnitude. This allows for clearer distinctions to be drawn between application classes.

2) Multicore Processors Tested: The specifications of the multicore processors tested during the validation of our methodology are shown in Table IV. All multicore processors used are from the Xeon family of multicore processors, with a varying number of available cores (ranging from six

Table III: Memory Intensity Classification

Applications	classification	baseline memory intensity (LLC / NI)
canneal (P)	Class I	1.84×10^{-2}
cg (N)	Class I	1.56×10^{-2}
ua (N)	Class II	1.63×10^{-3}
sp (N)	Class II	1.50×10^{-3}
lu (N)	Class II	1.11×10^{-3}
fluidanimate (P)	Class III	8.60×10^{-4}
freqmine (P)	Class III	3.47×10^{-5}
blackscholes (P)	Class III	1.88×10^{-5}
bodytrack (P)	Class IV	8.69×10^{-7}
ep (N)	Class IV	6.27×10^{-10}
swaptions (P)	Class IV	4.22×10^{-10}

to twelve), L3 (last-level) cache size, and frequency ranges. More detailed information about these processors can be found in [Int14b] and [Int14c].

Table IV: Multicore Processors Used for Validation

Intel processor	num. cores	L3 cache	frequency range
Xeon E5649	6	12MB	1.60-2.53 GHz
Xeon E5-2697v2	12	30MB	1.20-2.70 GHz

3) Training Setup: "Training data" was collected from each multicore processor to construct the models discussed in Section III. The training data for all of the machines was collected in the form of execution time values of various co-locations using all eleven applications as target applications co-located with a subset of only four of the applications available in the testing environment. Specifically, *cg*, *sp*, *fluidanimate*, and *ep* were used as the applications that were co-located with each "target" application to provide the training data with a selection of applications that are representative of each of the four classes of memory intensity. We limited our use of co-location applications for training data to four applications to keep the number of tests that we could run for training tractable.

When measuring application performance, data is collected for only a single "target" application during any given co-location test. Initial baseline tests were run that measured each application's execution without co-location across six P-state frequencies to determine how each application performed without interference from other applications. This baseline test provides a basis of comparison for the effect of interference on application performance degradation. These four applications were then scheduled co-located with each other in a manner that allowed the machine being tested to have a sparsely distributed set of co-location tests that were evenly spread across the number of total possible co-location combinations available to the machine for use as training data.

In particular, training data was collected for each of the eleven target applications by running tests co-locating each application with multiple copies of each of the four co-location applications mentioned earlier. These four co-location applications represent applications from each of the four memory intensity classes. Multiple copies of each of these co-location applications were run co-located together for each of the number of co-locations denoted in the "number of co-locations" column shown in Table V. Each of these sets of tests where then run once for each of the six selected P-states for each multicore processor. The P-state frequencies are shown in Table V. It should be clarified that each of the columns three to six in the table represent nested loops in the data collection code. Thus each attribute that is included as parameters to the data collection increases the size of the co-location space substantially. For example, the data collection was generally conducted as:

```
for each multicore processor:
   for each frequency:
      for each target application:
         for each co-located application:
            for each co-locations:
               get_exec_time_of_target()
```

The "num. of co-locations" column in Table V shows the number of additional applications that were homogeneously co-located with the target application (i.e. all co-located applications are of the same type). The applications ranged on each multicore processor from only a single co-located application occupying one additional core, to co-located applications running on all of the multicore processor's available cores (i.e., one target application plus $n-1$ co-located applications for a multicore processor that has n cores).

Setting up the training data in this way is an attempt to sample the set of all possible co-locations for a given machine in a uniform way that minimizes the amount of training data that is needed to calculate co-efficients for our model. At the same time, the training data is carefully selected to providing a model that is flexible. The data that is collected for training the models is designed to be able to both predict between the training data's gaps in the sample space, and extend beyond the set of four co-location applications available to the training data and be able to make predictions about applications that it has not seen previously.

4) Model Testing: Application testing was done by partitioning the training data described in Section IV-B3 through repeated random sub-sampling validation based on the bootstrapping approach first described in [EfT94]. Thirty percent of the data was randomly selected and withheld from the training process of each model. After training, the withheld data was run through each of the models and measured for accuracy. In this way, each model was tested using data that had not been seen previously during testing. The partitioning process was repeated one-hundred times, each time with a new random selection of points being withheld from training. The error values from each of these one-hundred training and

Table V: Training Schedule

Multicore processor	num. CPUs (n)	target applications	co-location applications	frequencies (GHz)	num. of co-locations
Intel Xeon E5649	6	all eleven applications	cg,sp,fluidanimate,ep	2.53, 2.40, 2.26, 2.00, 1.73, 1.60	1, 2, 3, 4, 5
Intel Xeon E5-2697v2	12	all eleven applications	cg,sp,fluidanimate,ep	2.70, 2.40, 2.10, 1.80, 1.50, 1.20	1, 3, 5, 7, 9, 11

Table VI: Effects of Memory Interference on *Canneal*

num. co-located applications	execution time (s)	normalized execution time	linear model F (MPE)	neural network model F (MPE)
0	220	1.00	-	-
1	229	1.04	11.0%	2.5%
3	239	1.09	9.4%	2.2%
5	253	1.15	7.3%	1.8%
7	266	1.21	4.5%	0.4%
9	277	1.26	4.1%	5.0%
11	292	1.33	1.7%	11.4%

testing partitioning groups was then averaged to determine the overall accuracy of each model.

V. Experimental Results

A. Overview

This section details the performance results of each of the 12 models we propose (two classes of modeling techniques - linear, and neural network - with six variants each, based on the six model feature groups in Table II). Each of the feature groups offers a trade-off between prediction accuracy and model complexity. Figures 1- 4 show the prediction accuracy for the training data set and testing data set on the 6-core Intel Xeon E5649 and the 12-core Intel Xeon E5-2697v2 multicore processors, respectively. Model prediction accuracy represented in both MPE and NRMSE is included for each of the machine learning techniques.

Each data point in the figures represents the average training error and average testing error from one-hundred partitions of the data for a particular model. In the case of each model, the error for each partition that was tested did not vary much (at most a quarter of a percent), meaning that each separate model shown has a tight confidence interval for its prediction accuracy.

B. The Effects of Memory Interference

Table VI provides an example of the performance degradation the *canneal* application experiences from memory interference due to increasing numbers of the *cg* application co-located on cores of a 12-core Intel Xeon E5-2697v2 multicore processor. It can clearly be seen in the table how the application's performance (the execution time of *canneal*) is negatively impacted over *canneal's* baseline execution time of 220 seconds as the number of co-located applications increases. The increase in execution time is shown in the "normalized execution time" column of the table as the co-located execution time normalized by the baseline execution time of the *canneal* application. In this case, increasing application execution time by as much as 33%. The table also shows the accuracy of the execution time predictions for linear and neural network models (using the MPE metric) making execution time predictions using feature set "F."

C. Results of Linear Modeling

For both multicore processors tested, the more advanced linear models provide only a modest improvement over the baseline linear model (model A) when feature information is added to the models. For the 6-core processor (Figure 1) the linear model has an MPE training error and testing error of 8% for its baseline models. With the addition of new features the 6-core processor is able to reduce its error down to an MPE of 6.5% for the linear model. The 12-core processor does not show any improvement from the addition of more model features (Figure 2). The linear NRMSE variance results for the 6-core and 12-core processors (Figure 3 and Figure 4) follow very similar trends to the MPE model except that the variance in predictions reduces from 4.25% to 3.75%.

The complexity of the sample space makes it challenging for the linear models to perform well beyond the baseline model, and in each case the only cache use information that proves to be helpful is information about the co-located applications. Given the fact that the model features do not help to train these models very well, it is not surprising that performance of the testing data very closely matches that of the training data. As mentioned earlier, non-linearity in the data make predictions with these linear models less accurate. The 12-core processor results in particular suffer from having to predict performance degradation for a much greater number of possible co-locations.

D. Results of Neural Network Modeling

The neural network models provide a clear improvement and very accurate predictions over the linear models (Figures 1-4). It can clearly be seen in the neural network models how the addition of application cache use helps to improve the predictions of each model. Predictably, the most complex neural network models, which utilize the most information, perform the best (operating with only a 2% MPE error on the

testing data for both multicore processors), however it appears as though the most important features are the features measuring the cache use information of the applications that are co-located with the target (coAppMem, coAppCM/CA, and coAppCA/INS from Table I).

As with the linear models, the NRMSE results show that the variance between the predictions and actual values decrease with generally the same trends as the MPE graphs except that for both machines the inclusion of the co-located application's cache access related metrics in model E reduced the NRMSE greater than it reduced the MPE.

E. Model Accuracy

Figure 5(a) shows a detailed view of each application's execution time distribution on a 6-core Intel Xeon E5649 machine. The points inside each application's distribution mark the specific execution time values measured for each test run for each application. Space limitations prevent showing a detailed view of the performance of all models, however Figure 5(b) shows a detailed view of the performance of the neural network model using feature set F (i.e. the most accurate model) on the execution time data shown in Figure 5(a). Distributions of the percent error between the model's execution time predictions and the application's actual execution time are shown for each application. The lines across each distribution represents the distribution's median (dashed line) and upper and lower quartiles (dotted lines). The figure shows that the accuracy of the model's predictions are generally accurate (there error is close to zero), that the majority of the model's predictions are $\pm 2\%$ from the actual execution time values, and that nearly all of the predictions are within 5% of the actual execution time values.

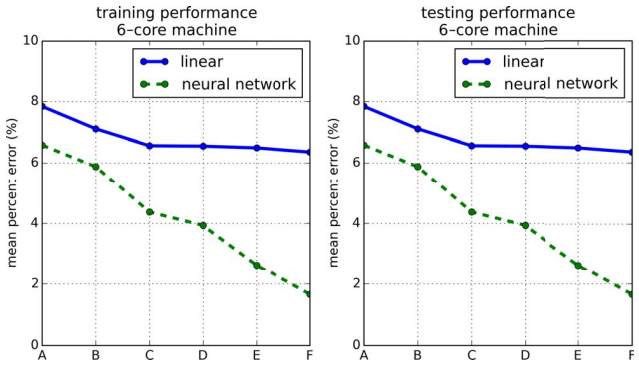

Figure 1: Model performance per feature set for the 6 core Intel Xeon E5649. (a) MPE for the training data set. (b) MPE for the testing data set. The figure shows results for each of the machine learning techniques: *linear* (solid blue) and *neural networks* (dashed green).

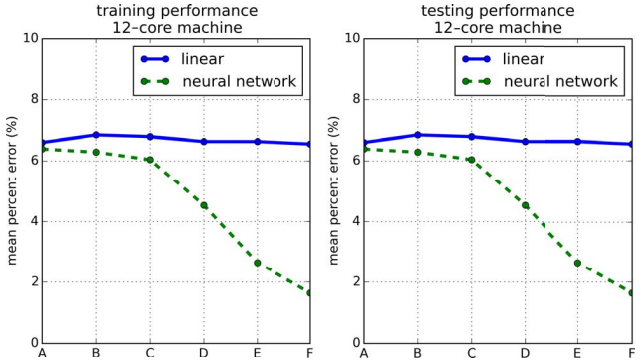

Figure 2: Model performance per feature set for the 12 core Intel Xeon E5-2697v2. (a) MPE for the training data set. (b) MPE for the testing data set. The figure shows results for each of the machine learning techniques: *linear* (solid blue) and *neural networks* (dashed green).

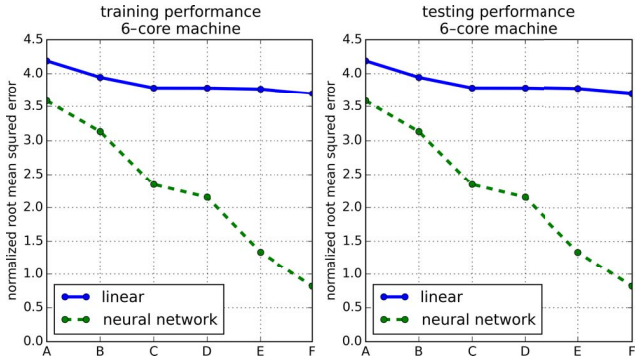

Figure 3: Model performance per feature set for the 6 core Intel Xeon E5649. (a) NRMSE for the training data set. (b) NRMSE for the testing data set. The figure shows results for each of the machine learning techniques: *linear* (solid blue) and *neural networks* (dashed green).

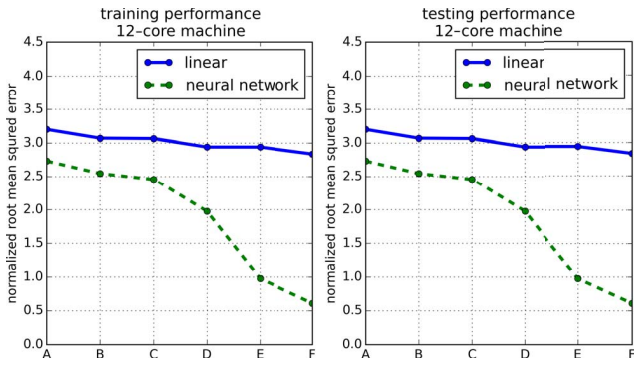

Figure 4: Model performance per feature set for the 12 core Intel Xeon E5-2697v2. (a) NRMSE for the training data set. (b) NRMSE for the testing data set. The figure shows results for each of the machine learning techniques: *linear* (blue) and *neural networks* (green).

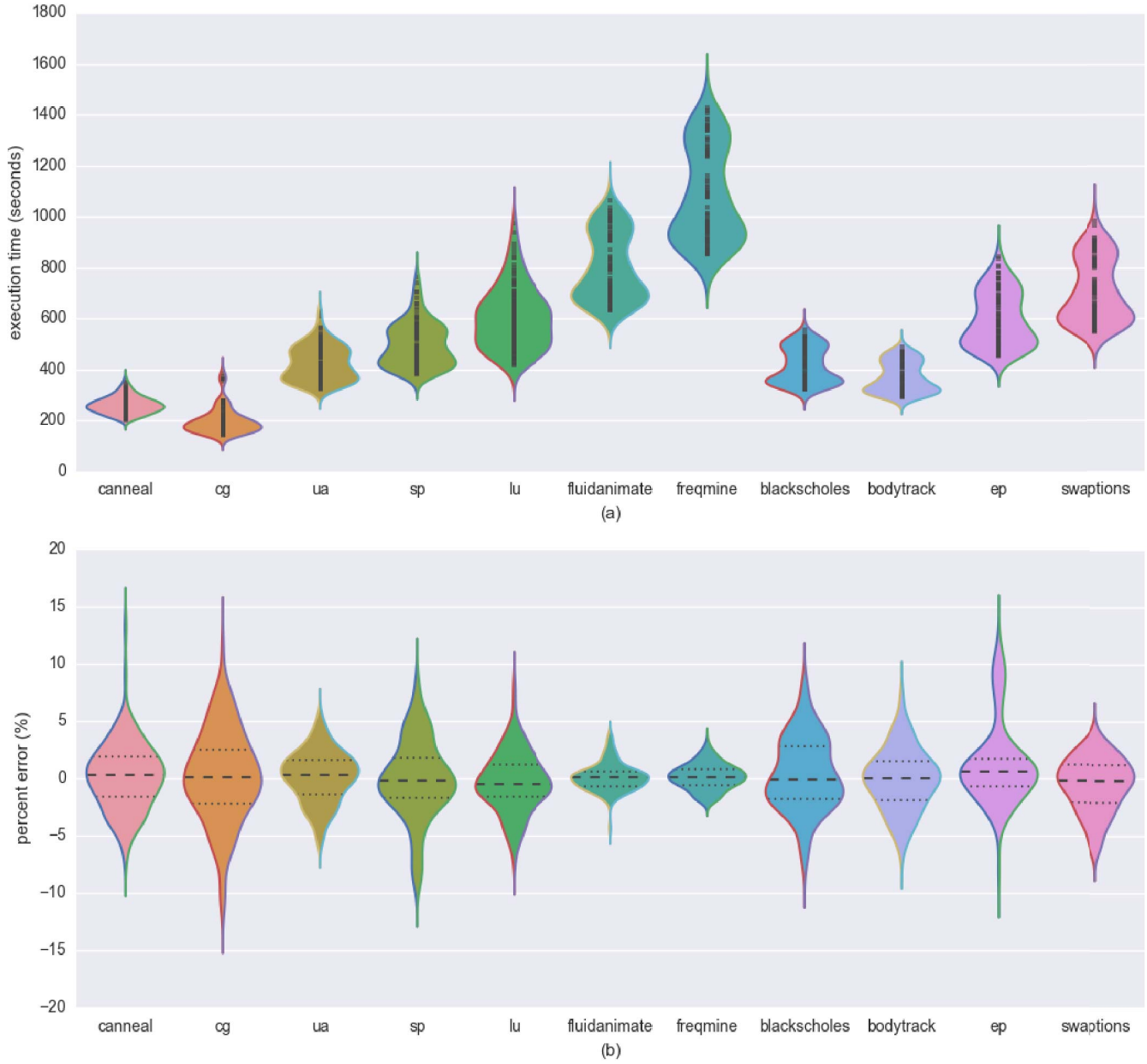

Figure 5: Distributions of each application's execution time (a), and the accuracy of the neural network model using feature set F on each application (b), for the 6 core Intel Xeon E5649 machine.

VI. Conclusions

In this paper, we propose a modeling methodology that allows for predictions to be made about the performance degradation that occurs from multiple applications running co-located on a multicore processor. The methodology is general enough to be applied to any multicore processor or set of applications. To validate the methodology we demonstrate its effectiveness by applying it to two server class Intel Xeon multicore processors with up to 12 cores, running real data workloads from two scientific benchmark suites.

Specifically, this work looked at how machine learning techniques can be used to make predictions about application performance degradation due to contention in shared cache and main memory resources when multiple applications were run co-located on the same multicore processor. While simpler linear models do not seem to provide much benefit from the addition of application cache use information,

the results from Figure 1 and Figure 2 show that neural networks can provide quite accurate predictions of application performance. For the neural network even the addition of only some of the application features is capable of producing fairly accurate performance predictions. Using all the features the neural network has only a very small MPE of 2% and an NRMSE of around 1%. Considering that performance degradation due to co-location can extend an application's execution time quite significantly, even a model limited to only knowledge of application memory intensity may be able to provide good enough predictions for performance.

Applying this methodology to create models for larger scale systems where the much greater number of cores could cause significantly larger performance degradation due to co-location would enable the design of a new class of interference-aware intelligent scheduling mechanisms that could provide substantial performance improvement in those systems. In addition to possibilities for optimizing application scheduling, a system for predicting application performance, such as the method shown here, offers the possibility for use in other areas such as energy modeling. Our next steps are to include monitoring of application power use into the testing environment. The energy use of a system is heavily dependent on the time that the system spends executing applications. Having this methodology that is capable of predicting an application's execution time when presented with the uncertainty of memory interference from co-location allows this work to lend itself very well to being able to also include the ability to estimate the energy used by the system during execution of a particular application, as well as the increase in energy use that is caused by memory interference. Finally, it would also be interesting to extend this work by examining application performance on families of machines outside of Intel's Xeon architecture.

VII. Acknowledgments

This work was supported by the National Science Foundation (NSF) under grant numbers CNS-0905339, CCF-1252500, CCF-1302693, ACI-1339745, and an NSF Graduate Research Fellowship. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. The authors thank Hewlett Packard (HP) of Fort Collins for providing us some of the machines used for testing.

References

[Bis06] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer New York, 2006, Vol. 1.

[Can12] Canonical. (2012) Ubuntu 14 release notes, https://wiki.ubuntu.com/TrustyTahr/ReleaseNotes.

[ChD14] J. Choi, M. Dukhan, X. Liu, and R. Vuduc, "Algorithmic time, energy, and power on candidate HPC compute building blocks." *IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 447–457.

[DaF14] D. Dauwe, R. Friese, S. Pasricha, A. A. Maciejewski, G. A. Koenig, and H. J. Siegel, "Modeling the effects on power and performance from memory interference of co-located applications in multicore systems," *The 2014 International Conference on Parallel and Distributed Processing Techniques and Applications(PDPTA-2014)*, Jul. 2014, pp. 3–9.

[DeK06] M. De Vuyst, R. Kumar, and D. M. Tullsen, "Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors," *International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, IEEE, 2006, pp. 10–20.

[DwF12] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei, "A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads," *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[EfT94] B. Efron and R. J. Tibshirani, *An Introduction to The Bootstrap*, CRC press, 1994.

[Htk14] (accessed Mar. 2014) HPCToolkit, http://hpctoolkit.org/.

[Int14a] "Intel 64 and ia-32 architectures software developer's manual volume 3b: System programming guide, part 2," Tech. Rep., Feb. 2014, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-.manual.pdf.

[Int14b] (accessed Oct. 2014) Intel xeon e5-2697v2 processor, http://ark.intel.com/products/75283/.

[Int14c] (accessed Oct. 2014) Intel xeon e5649 processor, http://ark.intel.com/products/52581/.

[MaT11] 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," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2011, pp. 248–259.

[MeS10] A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious scheduling for energy efficiency on multicore processors," *5th European Conference on Computer systems*, ACM, 2010, pp. 153–166.

[Nas14] (accessed Oct. 2014) NAS parallel benchmarks, http://www.nas.nasa.gov/publications/npb.html.

[PaE00] S. Parekh, S. Eggers, H. Levy, and J. Lo, "Thread-sensitive scheduling for smt processors," *Technical report, Department of Computer Science and Engineering, University of Washington*, 2000.

[Pap14] (accessed Mar. 2014) Performance application programming interface, http://icl.cs.utk.edu/papi/.

[Par14] (accessed Oct. 2014) PARSEC benchmark suite, http://parsec.cs.princeton.edu/.

[SaS13] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer, "Modeling performance variation due to cache sharing," *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*, May. 2013, pp. 155–166.

[TaM11] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," *38th Annual International Symposium on Computer Architecture (ISCA'11)*, Jun. 2011, pp. 283–294.