J Supercomput (2016) 72:4771–4809
DOI 10.1007/s11227-016-1783-y

HPC node performance and energy modeling with the co-location of applications

Daniel Dauwe[1] · Eric Jonardi[1] · Ryan D. Friese[1] · Sudeep Pasricha[1,2] · Anthony A. Maciejewski[1] · David A. Bader[3] · Howard Jay Siegel[1,2]

Published online: 24 June 2016

Abstract Multicore processors have become an integral part of modern large-scale and high-performance parallel and distributed computing systems. Unfortunately, applications co-located on multicore processors can suffer from decreased performance and increased dynamic energy use as a result of interference in shared resources, such as memory. As this interference is difficult to characterize, assumptions about application execution time and energy usage can be misleading in the presence of co-location. Consequently, it is important to accurately characterize the performance and energy usage of applications that execute in a co-located manner on these architec-

✉ Daniel Dauwe
ddauwe@rams.colostate.edu

Eric Jonardi
eric.jonardi@gmail.com

Ryan D. Friese
ryan.friese@rams.colostate.edu

Sudeep Pasricha
sudeep@colostate.edu

Anthony A. Maciejewski
aam@colostate.edu

David A. Bader
bader@cc.gatech.edu

Howard Jay Siegel
hj@colostate.edu

1 Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO, USA

2 Department of Computer Science, Colorado State University, Fort Collins, CO, USA

3 College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

tures. This work investigates some of the disadvantages of co-location, and presents a methodology for building models capable of utilizing varying amounts of information about a target application and its co-located applications to make predictions about the target application's execution time and the system's energy use under arbitrary co-locations of a wide range of application types. The proposed methodology is validated on three different server class Intel Xeon multicore processors using eleven applications from two scientific benchmark suites. The model's utility for scheduling is also demonstrated in a simulated large-scale high-performance computing environment through the creation of a co-location aware scheduling heuristic. This heuristic demonstrates that scheduling using information generated with the proposed modeling methodology is capable of making significant improvements over a scheduling heuristic that is oblivious to co-location interference.

Keywords Performance modeling · Energy modeling · Resource management · Memory interference · Application co-location · Benchmarking · Multicore processors · Scheduling

1 Introduction

There is an inherent trade-off in large-scale computing systems between reducing the use of system resources by consolidating applications into as few server processor nodes as possible (to reduce system power), and the performance degradation that occurs to these applications as a result of sharing system resources with other applications (e.g., [1,2]). Memory interference caused by multiple applications co-located on a multicore processor has been shown to negatively impact application performance (e.g., [3–7]). Specifically, the sharing of system resources such as DRAM and the last-level cache by co-located applications creates contention and, in many cases, increases the memory intensity of all applications running on the multicore processor [6]. 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 be as much as double or triple the execution time of an application as compared to its baseline execution time, i.e., when unhindered by co-location interference [8].

Multicore processors are pervasive throughout many kinds of computing systems, but the performance degradation effects caused by co-location interference are most likely to be prevalent in large-scale server systems and high-performance computers. This is because in those types of computing systems, executing multiple applications on multicore processors results in high memory interference and, therefore, causes performance degradation [9]. Having a methodology that is capable of predicting how well a system will run in a particular co-location scenario is very useful for such systems. The results of this work show how the information obtained from accurate predictions of co-location performance degradation can be integrated into intelligent application scheduling, and thus lead to system performance improvement due to better utilization of the hardware. Better utilization of the hardware provides increased opportunities to reduce power and save energy through server consolidation, while still

maintaining quality of service constraints in an improved manner over a co-location naïve scheduler. This work provides a methodology that can be used to create co-location aware performance models capable of predicting application execution time and energy use when single-core applications are co-located with other single-core applications on a multicore processor in an HPC node.

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, the methodology uses machine learning techniques to construct performance models characterizing that information. After they are trained, these models require only a single serial baseline measurement of parameters for each application running alone in the system. The models use this serial baseline measurement to make predictions about the performance degradation from memory interference that will occur when the application is executing with different types of co-located applications. While it has been shown in [4] that the degree to which an application's use of memory resources varies among different phases throughout the application's execution, this work illustrates that having such fine grain information is not always necessary to make accurate predictions.

After describing how the methodology operates, the theory behind the proposed methodology is validated using real-world data collected on three Intel Xeon server-class machines that were set to execute a collection of scientific application workloads, with some models providing up to 98 % accuracy. In addition to creating and demonstrating a methodology that is capable of being ported across processor architectures, this work also provides insight into what memory-use information is most important to obtain for a set of applications running in a system to predict the impact of co-location on performance and energy use.

The last portion of this work demonstrates the utility of the proposed modeling methodology through the creation and analysis of a consolidating slack-based scheduling heuristic that utilizes the execution time prediction models generated from the proposed modeling methodology to assist in its application co-location decisions. This "co-location aware" scheduling heuristic is shown in simulated studies to provide a significant performance improvement over a similar consolidating slack-based scheduling heuristic that is naïve to the effects of performance degradation caused by co-location.

Our work makes the following key contributions:

(a) identifies factors that can characterize slowdown during application co-location scenarios;
(b) proposes a novel methodology to integrate these factors into multi-granularity and multi-fidelity performance models that can be used to predict application execution time and energy use under co-location scenarios;
(c) shows that a fine level of detail is not always necessary to achieve reasonable prediction accuracy;
(d) validates 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;

(e) demonstrates through simulation the utility of the modeling methodology through the creation of a consolidating slack-based scheduling heuristic that utilizes models created from the proposed modeling methodology for its scheduling decisions.

The rest of the paper is organized as follows. The following section discusses related work in this area. The modeling methodology is presented in Sect. 3. Section 4 details the testing environment and data collection used for validating the models. Experimental results that validate the models are examined in Section 5. A demonstration of the modeling methodology's utility is shown in Sect. 6. The paper concludes with a summary of the main contributions in Sect. 7.

2 Related work

2.1 Overview

Several works have explored the effect of co-location on application performance and energy use in multicore environments and the use of scheduling heuristics for improving the efficiency of high performance computing. Here, the most relevant prior works in these areas are briefly summarized.

2.2 The effect of application co-location on performance and energy use

The authors in [3] examine 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 as our work does. This work also does not create co-location performance models the way that our work does, nor does it discuss modeling energy use. Our work also discusses a heuristic approach for mapping co-located processes to processor cores, whereas their analysis only examines a single processor node, providing a less precise view than our work where we consider an entire system.

The study in [4] provides an excellent review of how the architecture on which an application is executed can affect the cache use and memory intensity of that application. That work, however, does not attempt to make predictions about performance degradation as we do, but rather it shows the importance of including memory intensity and cache usage information when characterizing performance degradation in the presence of application co-location.

The work in [10] describes the challenges faced by applications sharing resources, and a need for the ability to perform precise predictions of performance degradation. The paper presents its "Bubble-Up" methodology for predicting performance degradation. However, it does not consider the impact of dynamic voltage and frequency scaling on application performance as we do in our work, and their study does not collect experimental data or characterize the memory interference effects of having more than two applications co-located, whereas we examine the performance degradation effects of more than two co-located applications

The authors in [5] present an extension to the "energy roofline" model that explores the effect of memory intensity (from the perspective of arithmetic intensity) on execu-

tion time and power use. The study executes a series of constructed microbenchmarks on twelve machine architectures and provides an analysis of the performance of the systems. While that study collects data about performance degradation from memory interference on a set of real machines, it uses small "microbenchmark" tests on machines, as opposed to the scientific workloads we use. Moreover, their work does not create models to predict execution times or energy use based on memory interference.

Similar to our work, the work in [11] examines creating a portable methodology using machine learning techniques for predicting application performance degradation from shared resources. The authors in that paper also incorporate shared resources beyond the last-level cache. However, incorporating those resources causes the resulting model to be complicated, and their model requires the constant monitoring of 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 a better prediction of 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.

A methodology for online estimation of an application's execution under co-location is presented as the "Application Slowdown Model" in [7]. Similar to our work, their work makes predictions of application performance degradation by monitoring processor performance counters. However, their work does not perform any experiments on how the methodology performs for any actual server class processor as our work does and, therefore, it is potentially limited in its portability or performance on actual systems. Additionally, their work limits their proposed models to analyzing the effect of performance degradation on application execution time and does not attempt to model energy as our work does.

Our work in [6] measures memory interference from application co-location, and its impact on system performance and energy use for a single Intel i7 machine. However, that work does not create models that predict system performance, and the scope is restricted to only a single consumer class machine.

We acknowledge that work exploring the effect of simultaneous multithreading (SMT) on application performance is an open and active area of research. Papers such as [12–14] examine scheduling and resource use of applications executed 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 SMT to remove the possibility of application interference in the L1 cache.

2.3 Scheduling heuristics

Examples of prior work in scheduling for large-scale HPC systems have appeared in [15–18]. In [15], the authors look at the problem of energy-constrained dynamic allocations of tasks in heterogeneous cluster computing environments, in the presence of individual task deadlines. The work in [16] proposes power and thermal-aware scheduling to optimize individual tasks reaching their deadlines. A "utility" metric

is defined in [17] that is used in combination with several energy-aware scheduling heuristics to provide a method of resource allocation that can maximize task performance while operating under a system energy constraint. The work in [18] examines energy-aware static resource allocation of a "bag of tasks" in a heterogeneous computing system. However, all these existing techniques in scheduling do not focus on co-location effects that can significantly impact the validity of scheduling decisions.

In contrast to the above prior works, some researchers have proposed slack-based heuristics to improve system performance. For example, the work in [19–21], and any of the other numerous works that rely on the backfilling technique first described in [22], all rely on slack-based heuristics that perform their calculations based on predictions of application execution time. All these works use application slack to provide better scheduling around uncertainty, but none of them accounts for the effects of performance degradation from co-location. As we demonstrate later, slack-based heuristics that are co-location aware can outperform co-location naïve slack-based heuristics. Consideration of co-location in the slack calculations of these works could be used to improve upon the work presented in these papers.

3 Modeling methodology

3.1 Overview

The proposed modeling methodology uses two types of machine learning techniques, *linear modeling* and *neural networks*, for constructing the predictive models. These techniques have been used in prior related work [10,11], but were limited in attribute selection and scope. For each machine learning technique, several models of varying levels of complexity and utilization of application features were built. The models are designed to make predictions about the execution time or energy use of one single-core application co-located with one or more other single-core applications in a multicore processor. The cores are not multi-tasked.

3.2 Model features

Both the execution time and energy prediction models use up to eight separate features of application execution to predict how the target application performance or energy use is impacted by co-located applications. The eight features were chosen by performing a principal component analysis (PCA) [23] on the data collected from multicore processors considered in this work. PCA allows us to observe which features were most important to include in the models.

The features selected are a general set that are observable in almost all multicore processors. The models we construct use various combinations of the features. These models use different combinations of features, ranging from those that are most-commonly available to a scheduler to combinations that require detailed information about the application and may be difficult to obtain on some platforms. The eight features that we selected after our PCA analysis are shown in Table 1. The table gives the name of the feature in the first column, and the description of that feature in the

Table 1 Model features

Feature name	Description of feature
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/NI	Sum of co-application last-level cache accesses/instructions
targetCM/CA	Target application last-level cache misses/cache accesses
targetCA/NI	Target application last-level cache accesses/instructions

Table 2 Model feature sets

Set name	Features within set
A	baseExTime
B	baseExTime, numCoApp
C	baseExTime, numCoApp, coAppMem
D	baseExTime, numCoApp, coAppMem, targetMem
E	baseExTime, numCoApp, coAppMem, targetMem, coAppCM/CA, coAppCA/NI
F	baseExTime, numCoApp, coAppMem, targetMem, coAppCM/CA, coAppCA/NI, targetCM/CA, targetCA/NI

second column. The "target" application in the table is the one for which slowdown or increased energy use due to co-location is being predicted. The *baseline execution time* is the execution time of the target application without any co-location present. An application's *memory intensity* is defined to be the ratio of an application's last-level cache misses to the total number of instructions that the application has executed. Memory intensity is discussed further in Sect. 4.

3.3 Generality of the models

The features shown in Table 1 can be combined to create models of various complexities. The model feature sets listed in Table 2 represent six possible models, one baseline model (model "A") that uses only the *baseExTime* feature for predictions, and five other models. For each of the five other models, the resource manger (scheduler) 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 represents a realistic process where the resource manager progressively obtains more detailed information about the computing system and its executing applications.

Designing a methodology that provides several models with various levels of complexity allows a system designer greater freedom to make use of the modeling

methodology to predict application performance and energy use. Providing a set of models with a range of complexities allows for a prediction model with a basic feature set to be used when more detailed application execution information for applications in the system is not available. For example, a system designer may not know or have the ability to measure the performance counter derived information (explained in Sect. 4) required to create more complicated models, but could have access to adequate application information (such as baseExTime and NumCoApp) to use the methodology to design simpler models that still give reasonable prediction accuracy and benefit from co-location awareness. If there is no information about the HPC applications, then we acknowledge that it will be difficult to effectively utilize our methodology.

3.4 Linear modeling technique

To predict the impact of application performance degradation and energy use during execution under co-location, twelve linear models were developed: a set of six models for execution time prediction, and a set of six models for energy use prediction. Each of the six models in each set of models was constructed using the six model feature sets listed in Table 2. Each linear model is the sum of the products of the utilized features (denoted f_i) and the model coefficients determined during training (denoted for the execution time model as ct_i and denoted for the energy use model as ce_i), plus a constant $const$. Linear regression is used to calculate the values for the coefficients. A general linear model for predicting co-located execution time using N features (linear execution time prediction LETP) takes the form of

$$\text{LETP} = \sum_{i=1}^{N} (ct_i * f_i) + \text{const.} \tag{1}$$

The linear prediction of execution time and energy differs only in the objective that the models are trained to predict (execution time or energy use). A general linear model for predicting co-located energy use using N features (linear energy use prediction LEUP) takes the form of

$$\text{LEUP} = \sum_{i=1}^{N} (ce_i * f_i) + \text{const.} \tag{2}$$

3.5 Neural network modeling technique

From our observation of application execution in the presence of co-location, we noted that there are a few instances of nonlinearity in some of the features. This observed nonlinearity provided the motivation for creating a prediction model using a neural network that can capture such nonlinearities.

Neural networks are a class of machine learning techniques that can be used for creating predictive models [24]. The approach attempts to mimic the function of the

human brain by defining several "layers of neurons." Each neuron layer is composed of some number of individual neurons that take the outputs of the previous layer as each of their inputs, with the inputs to the first layer of neurons being the features of the data available in each model (see Tables 1, 2). The final output is the value of the predicted execution time or energy use that the application will experience under co-location.

Each neuron operates by multiplying each of its N inputs, x_i, by N corresponding weight parameters, w_i, summing these results, and finally evaluating the sum with a nonlinear function f. The kth neuron in layer j (denoted y_{jk}) operates according to

$$y_{jk} = f\left(\sum_{i=1}^{N} x_i w_i\right). \tag{3}$$

The nonlinear function f in Eq. 3 is called the activation function of the neuron, and attempts to mimic the biological process that occurs during activation of an actual neuron. Any sigmoidal function will satisfy this activation function, but the hyperbolic tangent function (tanh) was chosen, in particular, for this work because it allows for faster convergence when using gradient methods for training the weight parameters [25]. It is this activation function that allows neural networks to capture nonlinearities when modeling.

The neural network is trained by adjusting the weight values at each neuron to minimize an objective function that measures the squared error between the neural network's set of predicted values of the training data and the actual values of the data. Optimal weight values were determined using a *conjugate gradient* because it provides fast convergence [26]. Two separate neural networks were created for the execution time and energy predictions, respectively. As with the linear models, the neural network execution time prediction models were trained using measured execution time values of the target application, and the neural network energy use models were trained using measured energy use values of the target application.

3.6 Model accuracy

All models are evaluated using Mean Percent Error (MPE) and Normalized Root Mean Squared Error (NRMSE) to offer two different measures for comparing model predicted values to actual values. Error measurements are only made for the target application's execution time or energy use in each test, not for all applications co-located in the system.

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 s to over 1000 s based on the application that is being executed and the state of co-location of the applications in the system). Thus, when finding MPE, a calculation of *relative error* allows the evaluation of prediction accuracy independent of these magnitudes for each of the M sample points of data. In the equation, predicted values are denoted p_j and actual values are denoted by a_j. MPE is defined as:

$$\text{MPE} = 100 * \frac{1}{M} \sum_{j=1}^{M} \left| \frac{p_j - a_j}{a_j} \right|. \tag{4}$$

NRMSE gives an indication of the variance of the predicted values from the actual values. For M sample points, NRMSE provides a ratio of Root Mean Squared Error (an *absolute error*) and the interval of values that the actual data can take (the largest actual data value $a_{\max}$ minus the smallest actual data value $a_{\min}$). Normalized root mean squared error is defined as:

$$\text{NRMSE} = \frac{\sqrt{\frac{\sum_{j=1}^{M} (p_j - a_j)^2}{M}}}{a_{\max} - a_{\min}}. \tag{5}$$

4 Implementation

4.1 Testing environment

This section describes a testing environment that can be used for the modeling methodology's data collection and validation. The testing environment that is described is not only effective and easy to use for collecting the data, but also easy to replicate and can be used on a wide variety of multicore processors.

4.1.1 Operating system

One of the objectives of this research was to design a methodology that can be applied to a wide variety of computing systems. The testing environment was designed to be portable across many multicore processor architectures to allow for simplicity in gathering test data and ease of recreating the testing environment for future users of this work. To ensure accurate data are collected, the testing environment is run from a "lightweight" command line version of the Ubuntu 14.04 operating system [27] installed on a USB drive. This minimizes the effect that the operating system has on application execution. Non-essential OS utilities and kernel daemons were removed so that the applications being monitored suffer as little interference as possible from unpredictable events in the OS. Such an environment mimics a large-scale computing platform meant to execute multiple applications concurrently.

4.1.2 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 [28]. 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 among microarchitectures

the number and types of performance counters that are available to the system are not consistent (e.g., differences described in [29–31]). Given the design goal of having portability for the methodology, interfacing directly with these hardware performance counters is not a feasible option. Therefore, the testing environment makes use of two tools to facilitate interactions with the hardware.

The first tool is "Performance Application Programming Interface" (PAPI) [32], 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 the testing environment utilizes is the HPC toolkit [33]. This suite of tools interfaces with PAPI and makes 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.

4.1.3 Measuring cache use

From [6], it is known that applications that need to access data from memory more often tend to experience a larger amount of performance degradation due to co-location. These performance degradations are incorporated into the prediction models by collecting measurements related to 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:

(a) number of last-level cache misses an application experiences (LLCM) that represents the number of times an application must access main memory;
(b) number of instructions the application executes (NI);
(c) total number of last-level cache accesses the application attempts (TCA).

The model features that are derived from these measurements were listed in Sect. 3. 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 can only represent a sum of the total events of that type that have occurred during the time the performance counters are monitored, in this case the duration of the application's execution.

One notable metric derived from these data is application memory intensity. *Memory intensity* is defined to be a ratio of an application's LLCM to NI. 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 memory access speed rather than computational speed. Memory intensity also gives some idea of how much an application tends to access memory. A highly memory-intensive application is more likely to utilize the shared cache resources more

and, therefore, it will tend to affect, and be more affected by, the effects of memory interference from other applications.

4.1.4 Processor performance states (P-states)

Processor performance states (P-states [28]) 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 threshold. However, these benefits come at the cost of having to throttle the multicore processor speed by decreasing the clock frequency. This effectively always increases the execution time (and thus decreases 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. In this work, this effect is taken into account through knowledge of the baseline execution time of each application at a given P-state. The P-state in which each processor is going to be executed is assumed to be known before execution of the co-located applications. Each P-state has a different baseline execution time and the P-state of the processor is implicitly an input to each of the models by virtue of its value of baseline execution time.

4.1.5 Measuring power and energy use

We used a *Watts Up? PRO* power meter [34] to measure application power and energy use. The *Watts Up? PRO* power meter measures instantaneous power use of a target load at the "wall outlet" level with a sampling rate of once per second. Power values were recorded at the "wall outlet" level for the system as a whole. The resulting data then provided a record of the system's power use over the execution of each application. The data could also be summed to give the value of the total energy used by the system during the application's execution, or averaged to give the average power used by the system during the application's execution.

4.2 Data collection and experimental setup

4.2.1 Benchmark applications

The applications run as testing workloads for the model validation were taken from two scientific benchmark suites. The set of eleven applications considered vary in the types of computations that they perform and are characterized by a wide spread of memory intensity values. Table 3 shows the applications examined in this study. Applications taken from the PARSEC benchmark suite [35] are denoted with (*P*), and applications from the NAS benchmark suite [36] are denoted with (*N*). The table

Table 3 Memory intensity classification

Applications	Classification	Baseline memory intensity (LLCM / 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 II	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 V	6.27×10^{-10}
Swaptions (P)	Class V	4.22×10^{-10}

also shows each 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. The baseline memory intensity values shown for each application in Table 3 are for a single input size of each application. The impact on baseline memory intensity and baseline execution time values of different application inputs is assumed to be known by the system designer prior to execution. Should different application inputs significantly change the baseline memory intensity or baseline execution time characteristics of an application, then each input should be treated as a separate application during training.

As shown in Table 3, these applications have been categorized into five memory intensity classes, denoted "Class I" through "Class V". 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 V 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 allows applications from particular groups to be referred to in a more general manner. These groupings allow for a broader use of the methodology for performance prediction.

Having application class values allows a developer the possibility to be able to use our modeling methodology even if it is not possible to obtain a detailed measurement of an application's memory use. If an estimate of the memory intensity of a particular application type could be made based on its historical use, or if an application developer had prior experience developing similar applications with known characteristics of memory use, then these applications could be broadly categorized into one of these memory intensity classes. Having this general classification of an application, the developer can still gain some insight into the expected performance of the system by running the model substituting average memory intensity values for that application's class in place of its unavailable measured values. A system designer can create more classes than those we consider, to improve resolution and classification granularity,

especially if a much larger set of applications is considered. However, for the sake of brevity in discussion, we restrict the number of classes to five in this work.

It should be noted that the memory intensity values listed in Table 3 are calculated from baseline measurements for one specific system (Xeon E5-2697v2). The memory intensity values do not vary widely among the machines tested, thus the application memory intensity classes are used to represent categories for the Xeon family of multicore processors considered. It is also important to note that the memory intensity values among application classes tend to differ by orders of magnitude. This allows for clearer distinctions to be drawn among application classes.

4.2.2 Multicore processors tested

The specifications of the multicore processors tested during the validation of the methodology are shown in Table 4. All multicore processors used are from the Intel Xeon family of multicore processors, with a varying number of available cores (ranging from four to twelve), L3 (last-level) cache sizes, and frequency ranges. Detailed information about these processors can be found in [29–31].

4.2.3 Model training

Training data were collected from each multicore processor to construct the models discussed in Sect. 3. The training data for all of the machines were collected in the form of execution time, total energy use, and average power values of various co-location combinations using all eleven applications as target applications co-located with a subset of 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. Because our preliminary results show that more memory intensive applications tend to have a greater interference effect on co-located applications, we biased our selection of three of the co-location applications towards more memory intensive applications. The ep application is also included to represent the effects of co-location with a more CPU intensive application. This limitation of four co-location applications was imposed to keep the number of tests that were executed for training tractable.

When measuring application performance, data are 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

Table 4 Multicore processors used for validation

Intel processor	Num. cores	L3 cache (MB)	Frequency range
Xeon E3-1225v3	4	8	800 MHz–3.20 GHz
Xeon E5649	6	12	1.60–2.53 GHz
Xeon E5-2697v2	12	30	1.20–2.70 GHz

Table 5 Training schedule

Multicore processor	Num. cores (n)	Frequencies (GHz) applications	Target applications	Co-location	Num. of co-locations
Intel Xeon E3-1225v3	4	3.20, 2.70, 2.20, 1.80, 1.30, 0.80	All eleven applications	cg, sp, fluidanimate, ep	1, 2, 3
Intel Xeon E5649	6	2.53, 2.40, 2.26, 2.00, 1.73, 1.60	All eleven applications	cg, sp, fluidanimate, ep	1, 2, 3, 4, 5
Intel Xeon E5-2697v2	12	2.70, 2.40, 2.10, 1.80, 1.50, 1.20	All eleven applications	cg, sp, fluidanimate, ep	1, 3, 5, 7, 9, 11

applications. This baseline test provides a basis of comparison for the effect of co-location interference on each application. The training data were collected for each of the eleven target applications by running tests that co-locate each application with multiple copies of each of the four co-location applications mentioned earlier. Multiple homogeneous copies of each of these co-location application types were simultaneously executed with the target application, for each of the number of co-locations denoted in the "number of co-locations" column shown in Table 5. During training data collection, if one of the co-located applications finishes before the target application then the co-located application is immediately rescheduled to run co-located with the target application. This allows the target application to spend minimal time without co-location. Model training and data collection need to occur once per HPC processor type, per application. Each of these sets of tests was performed once for each of the six selected P-states on each multicore processor. The P-state frequencies are shown in Table 5. Each of the columns three to six in the table represent nested loops in the data collection code of Algorithm 1.

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 collected for the third row of Table 5 are for six different frequencies, eleven different target applications, a selection of one of four co-located application types, and one of six choices of the number of copies of the selected co-located application to run with the target application for a total of $6 \times 11 \times 4 \times 6 = 1584$ co-location scenarios.

The "num. of co-locations" column in Table 5 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 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 coefficients for the model. For all the co-location scenarios represented by the three rows in Table 5, we collected experimental data which we used in Sect. 5 to evaluate the accuracy of our prediction methods.

Algorithm 1 Training data collection

```
for each multicore processor do
    for each P-state frequency do
        for each target application do
            for co-located application do
                for each number of co-locations of co-located application do
                    get_exec_time_of_target()
                    get_system_energy_use_during_target_execution()
                    get_average_power_use_during_target_execution()
                end for
            end for
        end for
    end for
end for
```

4.2.4 Model testing

Application testing was performed by partitioning the training data described in Sect. 4.2.3 (the co-location scenarios from Table 5) by repeated random sub-sampling based on the bootstrapping approach first described in [37]. Thirty percent of the data were randomly selected and withheld from the training process of each model. After training, the withheld data were tested using each of the models and measured for accuracy. In this way, each model was tested using data that had not been seen previously during training. These withheld data are referred to as the *testing data*. The partitioning process was repeated twenty times, each time with a new random selection of points being withheld from training. The error values from each of these twenty training and testing partitioned groups were then averaged to determine the overall accuracy of each model.

5 Experimental results

5.1 Overview

This section discusses the performance results of each of the twenty-four models. Altogether there are two sets (one set for execution time predictions, and the other set for energy use predictions) of 12 models (two classes of modeling techniques—linear, and neural network—with six variants each, based on the six model feature sets in Table 2). Each of the feature sets offers a trade-off between prediction accuracy and model sophistication. Figure 1 shows the *execution time* prediction accuracy for the 4-core Intel Xeon E3-1225v3, 6-core Intel Xeon E5649, and 12-core Intel Xeon E5-2697v2 multicore processors. Figure 2 shows the *energy use* prediction accuracy for the same three processors. Model prediction accuracy for both the training and testing data sets presented in both MPE and NRMSE is included for each of the machine learning techniques.

Each data point in Figs. 1 and 2 represents the average training error or average testing error from twenty partitions of the data (as discussed in Sect. 4.2.4) for each particular model. Results for each of the twenty individual partitions are shown as the

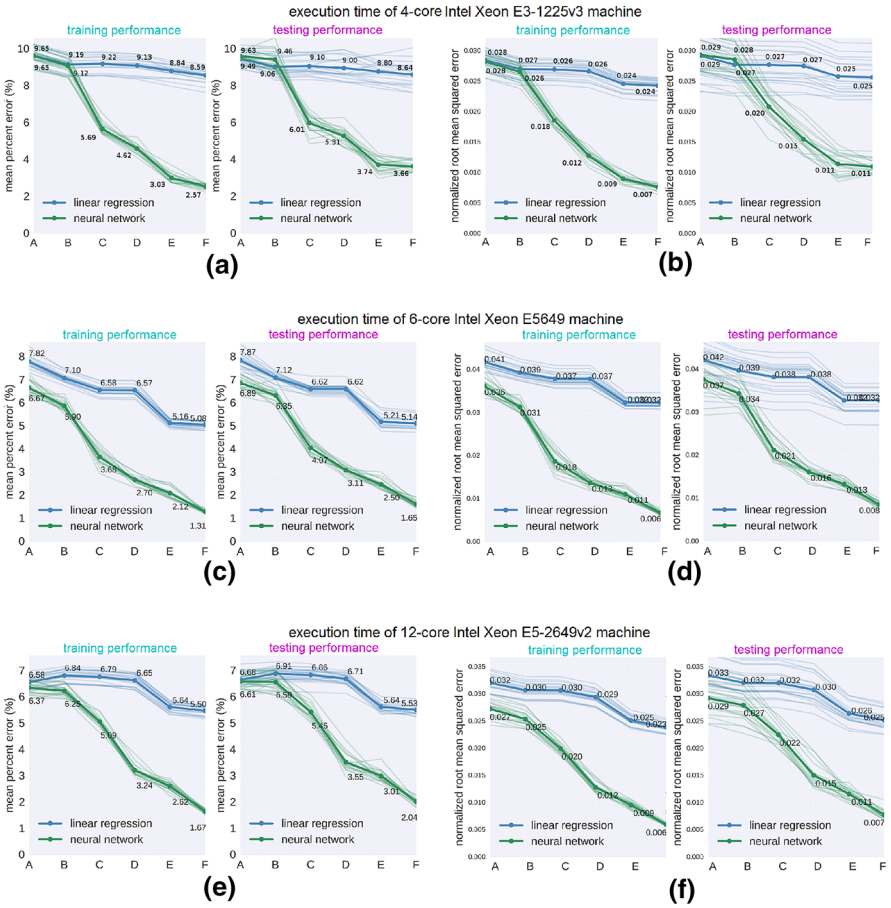

Fig. 1 Execution time prediction model performance per feature set for each Intel Xeon processor. **a**, **c**, **e** Show MPE for the performance of training and testing data sets for model feature sets A through F. **b**, **d**, **f** NRMSE for the performance of training and testing data sets for model feature sets A through F. The figure shows results for each of the machine learning techniques: *linear* (*blue*) and *neural networks* (*green*). Each point on the *figure* represents an average of twenty different partitions of the data into training and testing data. Annotations next to points indicate the value of the point. The *lighter shaded lines* indicate the performance of each model on individual partitions; the *darker shaded lines* indicate the average value across all twenty partitions (color figure online)

lighter shaded lines for each set of models, showing the range of values for each of the darker shaded average lines. The average values represented by each data point in Figs. 1 and 2 provide a comparison of each model's performance across all P-sates on which each application was executed during the data collection outlined in Sect. 4.2.3. As can be seen in the figures, the range of values among each partition that was tested does not vary much (at most 1.5 % for extreme cases). The annotations next to points represent the average value across the twenty partitions for that model.

In addition to results for each feature set of both machine learning techniques, Fig. 2 showing energy prediction model results also shows how effective it would be

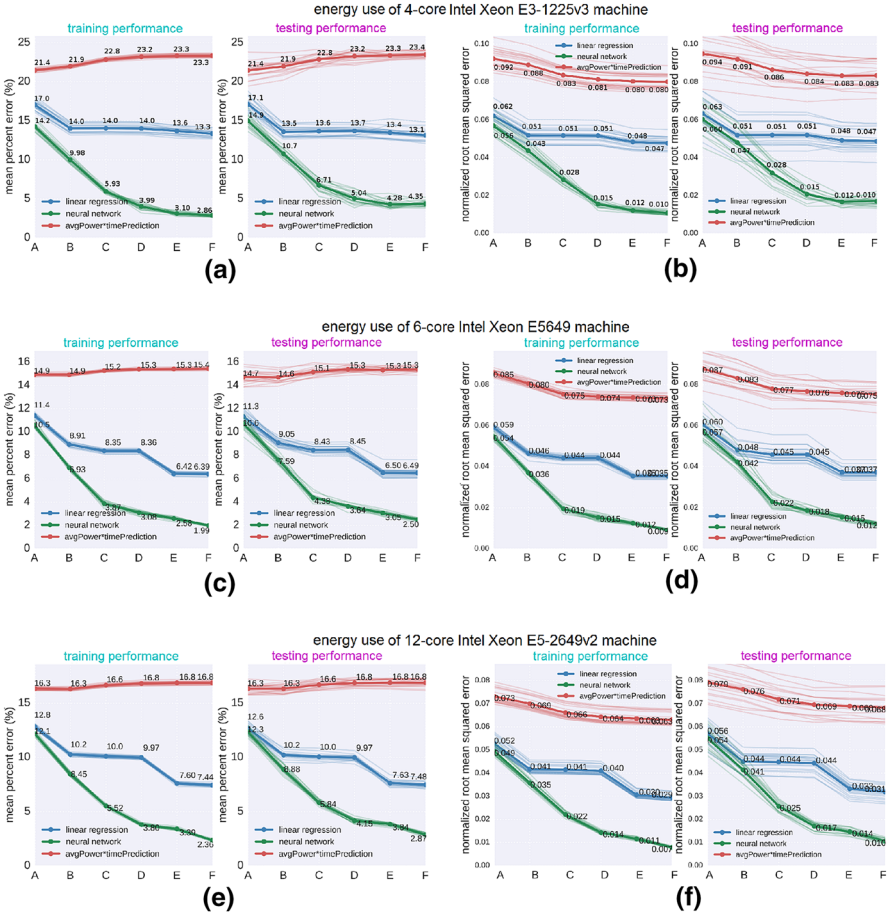

Fig. 2 Energy prediction model performance per feature set for each Intel Xeon processor. **a**, **c**, **e** MPE for the performance of training and testing data sets for model feature sets A through F. **b**, **d**, **f** Show NRMSE for the performance of training and testing data sets for model feature sets A through F. The *figure* shows results for each of the machine learning techniques: *linear* (*blue*) and *neural networks* (*green*), as well as a comparison to the results that are obtained by simply multiplying the corresponding feature set of the neural network execution time prediction for each processor (the results shown in Fig. 1) to each test's measured baseline average power value (*red*).Each point on the *figure* represents an average of twenty different partitions of the data into training and testing data. Annotations next to points indicate the value of the point. The *lighter shaded lines* indicate the performance of each model on individual partitions; the *darker shaded lines* indicate the average value across all twenty partitions (color figure online)

to make power-based predictions about energy use by multiplying an application's measured average power use with its predicted execution time under co-location (i.e., "$avgPower * timePrediction$"). This last set of power-based energy prediction models is presented to serve as a basis of comparison for the models generated by the methodology, and does not serve as part of the overall modeling methodology.

The value representing an application's average power use ($avgPower$) is *baseline average power*, a measured value collected for each application type and for each processor P-state. This value represents the average amount of power (specified in

watts) that the application uses throughout its execution. Similar to the model features listed in Table 1, baseline average power use is a value that is measured from an application's execution without co-location, and is obtained during the collection of application energy data specified in Sect. 4.2.3.

The predicted execution time value ($timePrediction$) used for the calculation of each "$avgPower * timePrediction$" energy prediction model is generated by the corresponding neural network execution time prediction model from the methodology for that particular feature set for that particular processor. For example, if an energy use prediction for an application running on the 4-core Intel Xeon E3-1225v3 processor is made using the "$avgPower * timePrediction$" model feature set "D", then the execution time prediction for the calculation would be made using the neural network model feature set "D" execution time prediction model for the 4-core Intel Xeon E3-1225v3 processor (the neural network model results that are shown in Fig. 1). The neural network model was used for execution time predictions because it provides better accuracy than the linear model, as discussed below.

5.2 Results of linear modeling

5.2.1 Overview

As the linear model feature sets become more advanced (i.e., as the models progress from using feature set A to feature set F), both the training and testing errors generally decrease for both the execution time and energy use sets of prediction models for all three processor types, indicating that the models become increasingly accurate when using increasingly advanced feature sets. It can be further seen from the lighter shaded lines showing the performance of individual partitions for both the execution time and energy prediction models of all three processors that as the model feature sets become more advanced the variance seen among individual partition results tends to decrease.

The complexity of the sample space makes it challenging for the linear models to perform well and improve significantly beyond the accuracy of the baseline model (relative to the improvement demonstrated by the neural network models). As mentioned earlier, non-linearity in the data makes predictions with these linear models less accurate.

5.2.2 Linear execution time modeling

For all three multicore processors tested, the more advanced linear models provide only a modest improvement in execution time prediction accuracy over the baseline linear model (model A) when feature information is added to the models. Linear execution time model results are shown in Fig. 1. The 4-core processor results (Fig. 1a, b) have a training and testing error that reduces by about 1 % MPE and 0.004 NRMSE from model A to model F. The 6-core processor (Fig. 1c, d) shows a reduction of about 2.74 % for both training and testing MPE from model A to model F. The 12-core processor (Fig. 1e, f) shows only about a 1 % MPE improvement from the addition of more model features from model A to model F. The linear NRMSE variance results

for the 6-core and 12-core processors follow very similar trends, reducing by about 0.008–0.009 NRMSE for both processor's training and testing results from model A to model F.

5.2.3 Linear energy use modeling

In general, the prediction accuracy for the energy use models tends to be lower than the execution time models, indicated by the models having higher MPE and NRMSE values. However, they also show a greater increase in accuracy as the models become more advanced. Linear energy use model results are shown in Fig. 2. The 4-core processor results (Fig. 2a, b) have training and testing error that reduces by about 3.7 and 4 % MPE, respectively, and a 0.015 NRMSE decrease in training error and 0.016 NRMSE decrease in testing error from model A to model F. The 6-core processor (Fig. 2c, d) shows a reduction of about 5.01 % MPE and 0.024 NRMSE for training data and 4.81 % MPE and 0.023 NRMSE for testing data from model A to model F. The 12-core processor results (Fig. 2e, f) shows a 5.36 % MPE and 0.023 NRMSE improvement for training data and a 5.12 % MPE and 0.025 NRMSE improvement for testing data from model A to model F.

It should be noted that even though the linear models for energy use are not able to outperform the neural network models, they still significantly outperform the energy calculation made by the "average power multiplied by co-located execution time." This is explained in detail in Sect. 5.4.

5.3 Results of neural network modeling

5.3.1 Overview

To allow for a fair comparison, the linear and neural network models use the same training and testing data partitions. The first observation to note regarding the neural network models is their clear improvement in prediction accuracy over the linear models, except for the single case of the execution time prediction model B for the 4-core Intel Xeon E3-1225v3 machine shown in Fig. 1a, b.

The neural network models exhibit similar results to the linear models of the lighter shaded lines converging with the more advanced feature sets, indicating increased model accuracy and decreased prediction variability between results. In addition, with more advanced feature sets, the neural network model results in a closer grouping of the lighter shaded lines than with the linear models. This implies that the individual partition results of the neural network models show less deviation from their mean than the predictions made by the linear models with the same partitions of data, indicating that the neural networks typically provide more consistent predictions than the linear models.

5.3.2 Neural network execution time modeling

Predictably, the complex neural network models, which utilize the most information, perform the best at predicting application execution time.

Neural network execution time prediction model results are shown in Fig. 1. The 4-core processor results (Fig. 1a, b) have training and testing error that reduces by about 7.08 % MPE for training and 5.97 % MPE for testing, and 0.021 NRMSE for training and 0.018 NRMSE for testing from model A to model F. The 6-core processor (Fig. 1c, d) demonstrates a reduction of about 5.36 % for training MPE and 5.24 % for testing MPE, and 0.030 NRMSE for training and 0.029 NRMSE for testing from model A to model F. The 12-core processor (Fig. 1e, f) shows about a 4.7 % MPE improvement and a 0.021 NRMSE improvement for training error, a 4.57 % MPE improvement and a 0.022 NRMSE improvement for testing error from the addition of more model features from model A to model F.

5.3.3 Neural network energy use modeling

The general prediction accuracy for the energy use models tends to have higher error values than the execution time models. Neural network energy use model results are shown in Fig. 2. The 4-core processor results (Fig. 2a, b) have training and testing error that reduces by about 11.34 and 10.55 % MPE, respectively, and a 0.046 NRMSE decrease in training error and 0.050 NRMSE decrease in testing error from model A to model F. The 6-core processor (Fig. 2c, d) shows an MPE reduction of about 8.51 % MPE and 0.055 NRMSE for training data and 8.1 % MPE and 0.045 NRMSE for testing data from model A to model F. The 12-core processor results (Fig. 2e, f) shows a 9.74 % MPE and 0.046 NRMSE improvement for training data and a 9.43 % MPE and 0.044 NRMSE improvement for testing data from model A to model F. It should be noted that the neural network energy prediction models for all three processors significantly outperform the energy calculation made by "average power multiplied by co-located execution time."

5.4 Calculating system energy use from time predictions and average power

The "$avgPower * timePrediction$" results shown in Fig. 2 (indicated by the red line in each portion of the figure) demonstrate the benefits associated with using our proposed energy prediction models (indicated by the blue and green lines in the figure) as opposed to the more approximate calculation of energy, i.e., using measured average power multiplied by the execution time prediction. The problem with these simpler calculations of energy use lies in the necessity of using an application's baseline average power. Even as the execution time prediction in the calculation becomes more accurate (moving from an execution time model using feature set "A" to one using feature set "F"), the resulting energy use prediction shows little to no improvement. An application's baseline average power use turns out to not be a sufficiently accurate measurement for making energy predictions calculated from power and execution time. Even if it were possible to make perfectly accurate execution time predictions, the "$avgPower * timePrediction$" models could not improve in their energy use predictions, demonstrating the necessity of using a more sophisticated modeling methodology, such as the one we propose, for making predictions about application energy use under the influence of application co-location.

Unfortunately an application's baseline average power is the only fair power measurement that can be used for this energy calculation without making the resulting model at least as complex as using models from our proposed methodology. Any power measurement that could be used to more accurately predict energy use would require either average power measurements directly measuring the effects of co-location on an application's average power, or power measurements that would require a more sophisticated modeling strategy to be used effectively for energy calculation. In either case, trying to more accurately predict energy use when using power and execution time measurements would not be effective without the creation of a more sophisticated model.

Another interesting effect caused by the inaccuracy of the average power by time calculation is observed when the execution time model increases in accuracy (moving left to right from model A to model F) for each processor type shown in Fig. 2. Both training and testing MPE actually *increase* when using the "$avgPower * timePrediction$" energy use calculation. This is because the execution time prediction models used for the $timePrediction$ value are making more accurate execution time predictions, and thus causing the energy use calculations to make more precise predictions of an inaccurate energy use value. That is, the model is making more accurate predictions of "$avgPower * timePrediction$," but because the use of baseline average power produces results that are inherently inaccurate; the MPE increases as the execution time predictions incorporate more features.

The reason that all three processors' NRMSE values of "$avgPower * timePrediction$" decrease as the predicted execution time value of the model becomes more accurate is because the calculation of NRMSE starts as a calculation of an absolute error (root mean squared error) that is then normalized to the range of the *aggregate whole* of the actual data (as shown in Eq. 5), whereas MPE is a relative error that is normalized to the actual energy use value at each data point. Because of this, small changes in these larger energy use predictions produce larger effects in NRMSE than it does in MPE where these larger predictions are each normalized by larger data values.

A more detailed analysis of these model results shows that the predictions made with the "$avgPower * timePrediction$" models tend to have the most variation in prediction accuracy for co-locations that produce higher values of energy use. Consequently, as the execution time predictions improve from models using feature set "A" to models using feature set "F," the energy use data that have higher values experience the most improvement in its prediction accuracy. The model predictions are inherently inaccurate because of their reliance on baseline average power. Because the predictions of high valued data were the most inaccurate, the improvement in these predictions was the most significant.

5.5 Model accuracy

The illustrations in Figs. 3 and 4 provide a more detailed view of the accuracy of the predictions made by the most accurate execution time prediction model and the most accurate energy use model created for the 6-core Intel Xeon E5649 machine. Each of the icons in Figs. 3a and 4a, respectively, represents the distribution of each

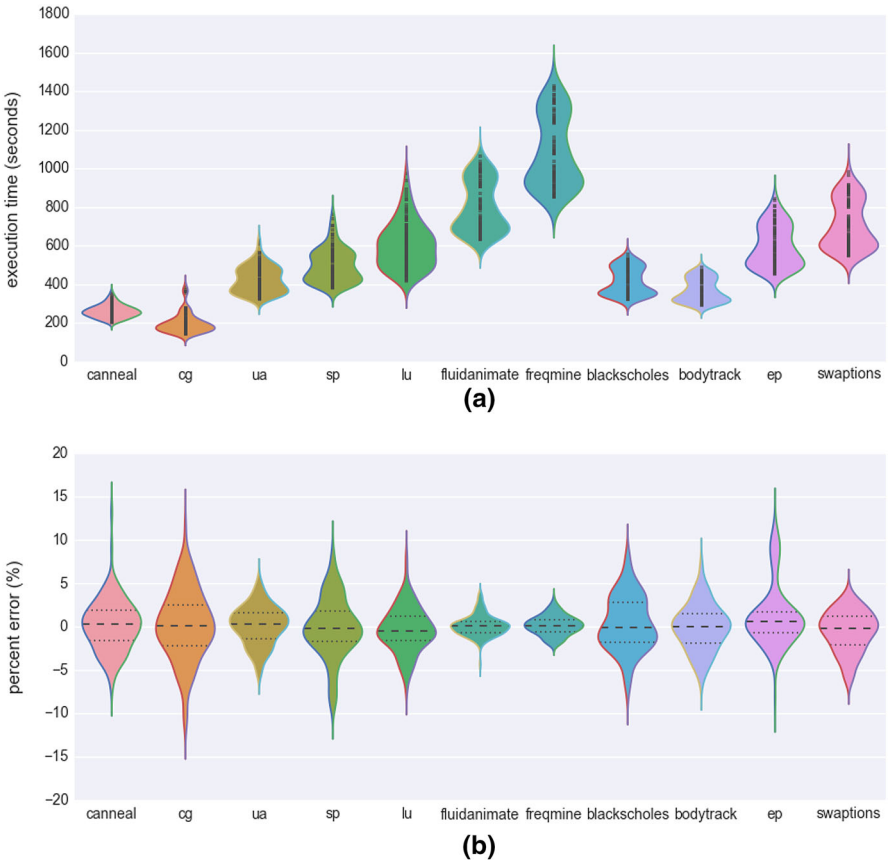

Fig. 3 **a** Distributions of each application's *execution time*. **b** The accuracy of the neural network model using feature set F predicting execution time for each application, on the 6-core Intel Xeon E5649 machine

application's measured execution time and energy use data. The icons in Figs. 3b and 4b, respectively, represent the distribution of error for the predictions of each application's measured execution time and energy use data. The larger width across each icon indicates a higher density of data points.

Figure 3a shows a detailed view of each application's execution time distribution on a 6-core Intel Xeon E5649 machine. The 120 points inside each application's distribution mark the specific execution time values measured for each test run of the application. Figure 3b shows a detailed view of the performance of the neural network model using feature set F (the most accurate model) on the execution time data shown in Fig. 3a. Distributions of the percent error between the model's execution time predictions and the application's actual execution times are shown for each application. The lines across each distribution represent the distribution's median (dashed line) and upper and lower quartiles (dotted lines). Figure 3 demonstrates that the model's predictions are typically accurate (their error is close to zero), that about half of the model's predictions are $\pm 2\,\%$ from the actual execution time values, and that nearly all the predictions are within $5\,\%$ of the actual execution time values.

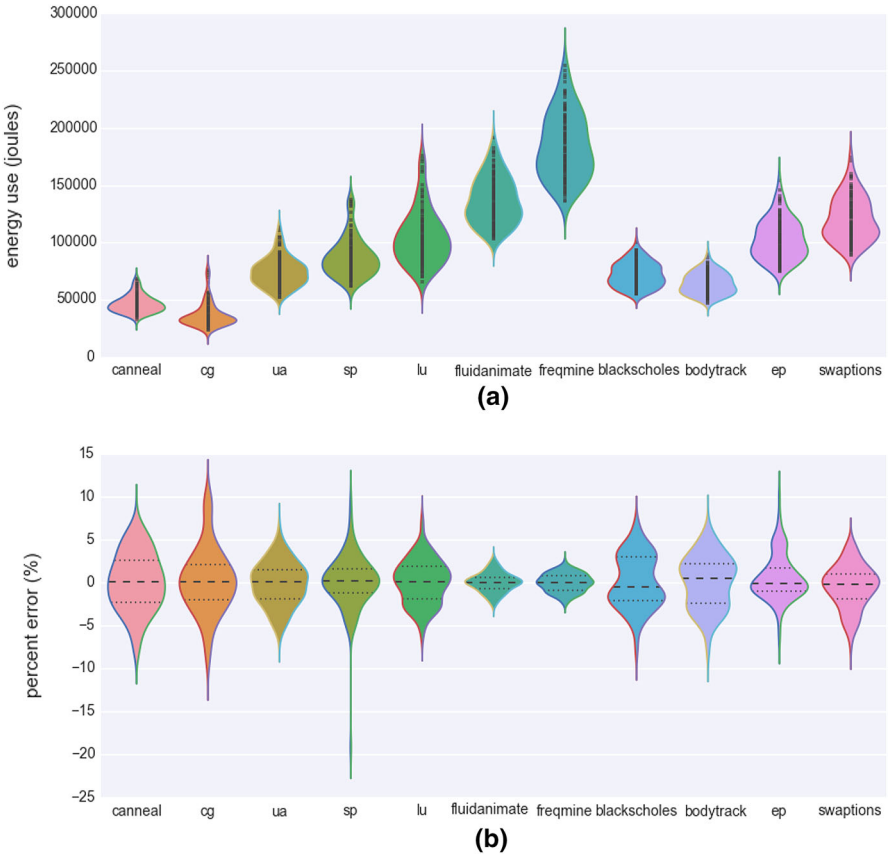

Fig. 4 **a** Distributions of each application's *energy use*. **b** The accuracy of the neural network model using feature set F predicting energy use for each application, on the 6-core Intel Xeon E5649 machine

The energy use prediction model results of the data shown in Fig. 4 are very similar to those of the execution time prediction model from Fig. 3. Each of the distributions shown in Fig. 4a shows a detailed view of each application's energy use distribution on a 6-core Intel Xeon E5649 machine, with the 120 points in each distribution marking specific energy use values of each respective application. Figure 4b shows a detailed view of the performance of the neural network model using feature set F (the most accurate model) on the energy use data shown in Fig. 4a. Distributions of the percent error between the model's predictions of energy use and the actual energy use are shown for each application. The lines across each distribution represent the distribution's median (dashed line) and upper and lower quartiles (dotted lines). Again, the figure shows a low error in model prediction accuracy for the majority of the data points. About half of the model's predictions are approximately ± 2.5 % from the actual energy use values, and nearly all the predictions are within about 7 % of the actual values.

Not surprisingly, by comparing the distributions of energy use shown in Fig. 4a to the distributions of execution time shown in Fig. 3a, it can be seen that the distributions are

fairly correlated. However, as discussed in Sect. 5.4, this correlation does not result in accurate energy use predictions when simply multiplying execution time and baseline power.

6 Prediction model utility

6.1 Overview

Predictions made about HPC system node performance are only useful if they can be shown to provide some benefit to the system. This section specifically demonstrates an example of the value of our modeling methodology's execution time prediction models when applied to scheduling compute node resources in a simulated 500-node homogeneous HPC system composed of system nodes with 4-core Intel Xeon E3-1225v3 processors. It is shown that, due to the impact of co-location interference on execution performance, scheduling with our proposed methodology can provide significant improvements in overall system performance.

6.2 Simulator

We designed a simulator to provide a comparison between a scheduling heuristic that is naïve to the effects of memory interference from co-location, and a heuristic that is aware of memory interference by utilizing the proposed modeling methodology. The simulator is event based, with the ability to simulate execution of multiple applications on large-scale HPC platforms. While the simulator is capable of simulating large-scale HPC platforms comprising any multicore processor type that application performance data have been collected for, we limit its use in this study to the 4-core Intel Xeon E3-1225v3 processor.

The simulator is also capable of modeling the effects of memory interference due to co-location on individual nodes through the use of "memory interference events." At a memory interference event, applications experience a delay in their execution, resulting in an increase in their execution times that is dependent on their specific co-location scenario (i.e., which application types are running on the other cores of the processor). The amount of execution time increase experienced by a particular application for a given co-location scenario was determined empirically from the data collection on the 4-core Intel Xeon E3-1225v3 processor, as discussed later in Sect. 6.3.

6.3 Data collection

To ensure that the simulated HPC environment would be able to simulate memory interference from co-location as accurately as possible, additional data collection beyond what was described in Sect. 4.2 was needed. That is, an exhaustive set of data for all possible co-location combinations of the applications used for simulation was collected on a real machine for these experiments to ensure accurate simulation-based

analysis. This complete set of data ensured a more accurate estimation of the real-world execution time for all the applications co-located on a multicore processor in the system. Because it would be prohibitively time consuming to collect application execution time information for all possible application co-locations for all the eleven applications used for validating the methodology (the applications listed in Table 3), the work for this simulated study considers only a subset of these applications (*cg, sp, fluidanimate,* and *ep*, described in Sect. 4.2.1).

All data collection was performed using the same testing environment described in Sect. 4.1. All data were collected at the lowest numbered P-state (highest performance P-state) of the 4-core Intel Xeon E3-1225v3 machine. The data used for the simulator were collected as a series of nested loops specifying the applications to be run on each core, as outlined in Algorithm 2.

The three processor cores that execute the co-located applications need to include an "IDLE application" in addition to the four application types cg, sp, fluidanimate, and ep to allow for application scheduling situations where no applications are scheduled on a particular core. This gave a total of $4 \times 5 \times 5 \times 5 = 500$ co-location scenarios for which data were collected. The exhaustive data collection described above was performed ten times. An average of each data point out of the ten sets of collected data was used for the simulation experiments.

Algorithm 2 Simulator data collection

```
for each multicore "targetApp" in {cg, sp, fluidanimate, ep} do
    for each multicore "coApp1" in {IDLE, cg, sp, fluidanimate, ep} do
        for each multicore "coApp2" in {IDLE, cg, sp, fluidanimate, ep} do
            for each multicore "coApp3" in {IDLE, cg, sp, fluidanimate, ep} do
                get_exec_time_of_target()
            end for
        end for
    end for
end for
```

6.4 Task simulation

Simulated instances of applications are referred to as *tasks*. Tasks can be any of the four application types, cg, sp, fluidanimate, and ep, and upon arrival, each task's type is selected according to a uniform random distribution. This makes the number of each application type arriving into the system for scheduling equally likely. Assumptions made about the simulated behavior of systems with task deadlines are based on our active collaborations with DoD and ORNL where we are researching scheduling strategies for their HPC platforms. We have found that in almost all oversubscribed scenarios, dropping tasks is acceptable and task deadlines are always defined based on the arrival time of a task [38,39].

For a given task i, the arrival time of that task into the system is defined as T_{A_i}, and is determined using a Poisson process. The first task to arrive into the system arrives at time zero ($T_{A_0} = 0$), and all subsequent tasks arrive according to the previous task's arrival time ($T_{A_{i-1}}$) plus an exponential random variable $\mathcal{T}_i \sim Exp(\lambda)$ with an

expected arrival rate of $E[\mathcal{T}_i] = \frac{1}{\lambda}$. Tasks arriving in this manner allow for flexibility in adjusting the subscription level of the system by only having to modify a single parameter λ. Task arrival time is given by

$$T_{A_i} = T_{A_{i-1}} + \mathcal{T}_i. \tag{6}$$

Tasks that do not complete execution by their deadlines are removed from the system. The deadline of any given task i is denoted T_{D_i}, and is generated using the task's arrival time T_{A_i} and *baseline execution time*. Baseline Execution Time is defined in Sect. 3.3 and denoted here as the variable ET_B. Task deadlines take a uniform random value over an interval $[a, b]$ giving $T_{D_i} = \mathcal{U}_i(a, b)$, with a defined as task arrival time plus a parameter β multiplied by baseline execution time, i.e.,

$$a = T_{A_i} + \beta ET_B. \tag{7}$$

The end of the interval of the uniform random variable, b, is defined as task arrival time plus a parameter γ multiplied by baseline execution time, i.e.,

$$b = T_{A_i} + \gamma ET_B. \tag{8}$$

For all experiments shown, $\beta = 1.2$ and $\gamma = 2.0$, thus making a task's deadline equal to the task's arrival time plus a random value between $1.2 * ET_B$ and $2.0 * ET_B$. The task deadline for this work is

$$T_{D_i} = T_{A_i} + \mathcal{U}(1.2, 2.0) * ET_B. \tag{9}$$

6.5 Scheduling heuristics

6.5.1 Overview

The goal of our scheduling heuristics is to maximize the number of tasks that complete by their deadlines. We simulate the behavior of two consolidating slack-based scheduling heuristics, one that is naïve to the effects of memory interference from application co-location (co-location naïve) and another that is aware of the effects of memory interference due to application co-location (co-location aware). These heuristics are utilized during simulated scheduling events that map tasks to processor cores. Scheduling events occur every time tasks arrive to the system, or processor cores become free. Consolidation-based schedulers attempt to maximize the number of cores that are executing tasks in each multicore processor node before powering up additional system nodes. Consolidation-based scheduling has been shown to provide benefits for HPC systems by minimizing the number of processor nodes that need to be active during a system's execution and, therefore, reducing the power needs and potentially increasing the energy efficiency of the system (e.g., [1,2]). We use a consolidation approach for our sample use of co-location in scheduling, and are aware

that there are trade-offs between the consolidation approach and an approach where tasks are distributed throughout the system.

Both the co-location naïve and co-location aware heuristics use a measure of a task's slack for making scheduling decisions. Task slack is a prediction of the time a task has remaining between when it is expected to finish and its deadline. The task slack calculation is used to ensure that a task will have enough time to completely execute under a given co-location scenario, and this calculation differs between a co-location naïve heuristic and a co-location aware heuristic. The task slack is defined in detail in Sect. 6.5.2. It is the goal of all scheduling heuristics used for these experiments to create schedules that maximize the number of tasks that meet their deadlines, while consolidating tasks as much as possible.

6.5.2 Task slack

Calculating task slack provides an estimate of the amount of time that is available between when a task is expected to be completed and its deadline. Because predictions of task execution time are necessary for calculations of task slack, it is to be expected that a better prediction of task execution time would produce a better prediction of task slack and, consequently, allow any slack-based scheduling heuristics dependent on task execution times to produce better task schedules.

For some task i in the simulated system, co-location naïve slack, S_{CN_i}, is calculated as the task's deadline (T_{D_i} defined in Sect. 6.4) minus the task's co-location naïve predicted time of completion, T_{CN_i}, calculated as the task's *baseline execution time* (defined in Sect. 3.3) plus the simulated current time (CT). Thus, the co-location naïve slack equation is

$$S_{CN_i} = T_{D_i} - T_{CN_i}. \tag{10}$$

The major difference between the equations for calculating co-location naïve and co-location aware task slack is that, instead of using the value of the task's baseline execution time for predicting a task's execution time, the co-location aware slack calculation uses a prediction of the co-location aware time at which a task will be completed, denoted T_{CA_i}. Before a task begins executing and has been assigned to a processor node, the initial value of T_{CA_i} is equal to the current time of the simulator (CT) plus the task's baseline execution time value. After the task's initial completion time has been calculated, the co-location aware slack for each task i, denoted S_{CA_i}, is calculated as the task's deadline (T_{D_i}) minus the task's co-location aware completion time prediction (T_{CA_i}), i.e.,

$$S_{CA_i} = T_{D_i} - T_{CA_i}. \tag{11}$$

However, after the scheduling heuristic has assigned the task to a processor core, the time at which the task will be completed changes due to its co-location with other applications, and the value of T_{CA_i} must be recalculated. The prediction of a target task's completion time after being co-located with other tasks is challenging because

(a) the interference, a target task will experience during its execution, will change over time as either the tasks that it is co-located with finish executing, or as new tasks arriving into the system are co-located with the target task during its execution;
(b) the target task itself will cause interference with the other tasks it is co-located with, increasing their execution time, and thereby changing the duration of each task's interference effects on the target task, making the execution time prediction harder.

The first problem can be solved by having the scheduling heuristic recalculate a task's slack value at every scheduling event (when tasks arrive into the system, or processor cores become free), thus ensuring that the slack value is up-to-date whenever the value needs to be used for scheduling. The second problem of finding an accurate prediction (as far as an underlying execution time prediction model will allow) for all tasks co-located on a multicore processor node despite there being a continuous complex interaction among them is solved through the iterative approximation algorithm shown in the pseudo-code in Algorithm3.

Algorithm 3 takes a set of tasks co-located on a processor, denoted **T**, and calculates the predicted completion time of all tasks in the set. Each task in the set of co-located tasks presented to the algorithm may be in various states of progress through their execution, and will have previous predictions of their completion times recorded from earlier in their execution that will be used in the algorithm.

Algorithm 3 Predicted time to task completion under current co-location (T_{CA_i})

inputs: a set **T** of the tasks co-located on the same processor node
outputs: a set **T**′ of tasks with predicted completion times
let $IS = CT$
while the size of **T** > 0 **do**
  let IE be the earliest task completion time value in **T**
  for each task t in **T** **do**
    calculate PUI of t under co-location **T**
    $T_{CA_i} = T_{CA_i} + PUI * (IE - IS)$
  end for
  for each task t in **T** **do**
    if $IE + 1 \geq T_{CA_i}$ **then**
      add t to **T**′
      remove t from **T**
    end if
  end for
  $IS = IE$
end while
return **T**′

The algorithm starts by recording the simulator's current time in the variable IS that stores the start time of the current co-location interval. The co-location interval is defined as the largest interval of time that the co-located tasks in the processor are guaranteed not to change. For example, if two tasks, A and B, are co-located on a processor, then each of these tasks will have a previously predicted completion time calculated earlier in the simulation. Let the predicted completion time of task A be $CT + 5$ and that of task B be $CT + 10$. Then for this scenario, the co-location interval

is from CT to $CT+5$ which is the interval during which these tasks are guaranteed to be executing co-located together. As another scenario, consider a task C that arrives and is co-located with tasks A and B with task C having a completion time of $CT+4$. In this scenario, the co-location interval will be from CT to $CT+4$.

The calculation of predicted unit interference PUI is where our methodology's execution time prediction models are utilized. The predicted value of a task's execution time under co-location is denoted ET_C. After the execution time prediction has been made by the prediction model, the task's baseline execution time (ET_B) is subtracted from this value and then divided by the task's baseline execution time. This gives the increase in execution time a task experiences during a one second unit of execution for a particular co-location scenario. Thus, PUI is

$$\text{PUI} = \frac{ET_C - ET_B}{ET_B}. \tag{12}$$

The general procedure for calculating each task's completion time occurs in the *while* loop starting on line 4. An iteration of this *while* loop starts by finding the end of the current co-location interval, IE (line 5). Next, the *for* loop on line 6 updates the completion times of each task in **T** by first calculating the task's PUI according to Eq. 12, then multiplying this value by the value of the current co-location interval ($IE - IS$), and finally adding it to the task's last predicted completion time (lines 6–8). After each task's completion time has been updated for the current iteration of the *while* loop, the *for* loop on line 10 removes tasks from **T** with predicted completion times within 1 s after the end time of the co-location interval (necessary for establishing which tasks will be in the next co-location interval), and puts them in the set $\mathbf{T}'$. The algorithm then updates the co-location interval start time of the next iteration of the *while* loop to be the current interval's end time (line 16). The algorithm continues iterating through the *while* loop until all tasks have been removed from **T**, indicating that completion time predictions have been made for all tasks co-located on the node, and returns the set $\mathbf{T}'$ of tasks with updated completion times. After the predictions of completion time under the current co-location scenario have been completed for each task in the processor node, the co-location aware slack for each task i is calculated as defined earlier in Eq. 11.

The neural network model F for the 4-core Intel Xeon E5-1225v3 system (the model used for execution time predictions in this simulated study) is trained just once, using the entire set of data collected as specified in Sect. 4.2.3. The accuracy of this execution time prediction model has a training performance MPE of 2.50 % when trained with the full set of training data.

6.5.3 Co-location naïve scheduling heuristic

The co-location naïve scheduling heuristic is shown in Algorithm 4, and takes as input the set of "unmapped tasks" **U** that are a set of tasks that have arrived into the system but have not yet been scheduled to a processor core, and the list **N** of processor nodes with available cores. The algorithm returns a mapping of a subset of the tasks in **U** to a subset of the nodes in **N**.

Algorithm 4 starts by sorting tasks in **U** from least slack to most slack according to their slack values calculated with Eq. 10, then stores these values in the list $\mathbf{U}'$, and initializes the boolean variable NA, which indicates if a node is available, to false (line 3–4). Next, the algorithm enters the outer *for* loop that iterates over each task u_i in $\mathbf{U}'$ (line 5). Because the list is sorted according to slack, the tasks that are closest to their deadlines and most in need of scheduling get considered for scheduling first. At the beginning of each iteration of the outer *for* loop, the set of nodes **N** is sorted from the node with the least number of available cores to the node with the most number of available cores, and stored in $\mathbf{N}'$. It is this list $\mathbf{N}'$ that is subsequently iterated over in the inner *for* loop (lines 6–7). Allowing the nodes to be iterated over in this way encourages consolidation. In line 8, the co-location naïve slack S_{CN_i} is calculated for task u_i if it were to be scheduled on node n_j. If the calculated slack prediction is greater than zero, then NA is set to true, the available node n_j is recorded to the variable that stores the node with available slack $NWAS$, and the algorithm breaks from the inner loop, otherwise it checks the next processor node (lines 9–12). Once the inner *for* loop is complete, if there was a node available, then task u_i is scheduled onto the node recorded in $NWAS$, otherwise the algorithm moves on the next unmapped task (lines 15–16). After the outer *for* loop has iterated through all the unmapped tasks, the algorithm is complete.

Algorithm 4 Co-location **naïve** consolidating slack-based scheduling heuristic

```
1: inputs: unmapped tasks U, nodes with available cores N
2: outputs: a mapping of a subset of U to a subset of N
3: let U' be a sorted list of tasks U from least slack to most slack
4: initialize NA = False
5: for each task u_i in U' with i = 1 to the size of U' do
6:     let N' be a sorted list of nodes N from least to most number of available cores
7:     for each node n_j in N' with j = 1 to the size of N' do
8:         calculate co-location naïve slack, S_CNi, for task u_i on node n_j
9:         if S_CNi > 0 then // u_i can be completed on node n_j
10:            NA = True
11:            NWAS = n_j
12:            break
13:        end if
14:    end for
15:    if NA then // it is predicted that task u_i can be completed
16:        assign u_i to execute on node NWAS
17:    end if
18: end for
```

6.5.4 Co-location aware scheduling heuristic

The co-location aware scheduling heuristic operates similarly to the co-location naïve scheduling heuristic shown in Algorithm 4. The only difference between the co-location aware heuristic and the co-location naïve heuristic is the scope and calculation of task slack (lines 8–9 of Algorithm 4). The co-location aware heuristic uses the slack calculation of Eq. 11 for its task slack predictions, as opposed to the co-location naïve

heuristic's use of Eq. 10. In addition, the co-location aware scheduling heuristic is aware of the other tasks already executing on the node n_j and calculates co-location aware slack, S_{CA_i}, for all the tasks on the node, not just the unmapped task u_i. When the algorithm checks the slack for all tasks in the node, it is not only checking to see if the unmapped task u_i will be able to finish executing when subjected to node n_j's co-location, but also checking to make sure that placing u_i on the node will not produce so much interference that it could make the co-located applications already executing on node n_j miss their deadlines. For the co-location aware scheduling heuristic, the conditional statement checking the boolean value of NA would look similar to line 15 of Algorithm 4 of the co-location naïve heuristic. However, a value of $NA = \textbf{true}$ for the co-location aware heuristic indicates that not only can the task u_i be completed, but also the task u_i can be mapped to the node without causing any missed deadlines.

6.5.5 Perfect prediction scheduling heuristic

A third scheduling heuristic that we consider in our study demonstrates co-location aware scheduling with perfect prediction (perfect pred) of application interference. As the name suggests, the "perfect prediction" scheduler shows how the co-location aware scheduling heuristic would behave if the modeling methodology was capable of perfectly predicting an application's execution time under any co-location situation. The "perfect prediction" scheduler makes its scheduling decisions using the same scheduling procedure as the co-location aware scheduling heuristic outlined above in Sect. 6.5.4, except that instead of using our proposed co-location interference modeling methodology for its prediction of a task's increased execution time under co-location, its execution on a simulator allows it to predict the future execution time values under co-location obtained after profiling on a real system. Therefore, a task's execution time and slack under co-location are "predicted" perfectly accurately for the purposes of scheduling.

It is not possible for such a scheduler to exist outside of a controlled simulation, but it is shown here for the purposes of providing an upper bound on what is achievable for any consolidation-based co-location aware heuristic. The difference in performance between the co-location aware scheduling heuristic using "perfect prediction" and the co-location aware heuristic using the proposed modeling methodology for its predictions gives an indication of how often tasks miss their deadlines due to inaccurate predictions, instead of environmental factors of the computing system.

6.6 System measurements

6.6.1 Overview

We use four different measures for comparing the heuristics in the system: system performance, core utilization, node utilization, and core utilization of active nodes (each defined in the following sections). Each measure gives a different view of how the heuristics behave during system simulation, and allows for a detailed analysis of the system.

6.6.2 System performance measure

After a simulation has completed, each task will be in one of three states.

(a) *completed* The task was scheduled to a processor core and successfully completed at or before its deadline.
(b) *missed deadline* The task was scheduled to a processor core, but was unable to be completed before its deadline. This outcome occurs when a scheduling heuristic makes a mistake in its assumption about a task's execution time (for instance, the task experiences longer execution times due to co-location interference). When a task misses its deadline, it is removed from its processor core at its deadline time.
(c) *unassigned* When the number of arrived tasks in the system exceeds the number of available cores (either physically available cores, or cores that a scheduling heuristic determines can be used without causing missed deadlines), then the task is put in a queue of arrived tasks waiting to be scheduled. If the task is unable to be scheduled before its deadline, then at the time of its deadline the task is removed from the system and labeled as "unassigned."

The goal of this scheduler is to complete as many tasks by their deadlines as possible, thus reducing the number of tasks that miss their deadlines or are not executed at all. The overall system performance using a particular scheduling heuristic is determined by measuring the number of tasks that meet their deadlines at the end of system simulation.

6.6.3 Core utilization measure

A system core is considered active if it is executing a task. The core utilization CU at a particular instant of the system's simulation is calculated from the ratio of the system's active cores (AC) to the system's total cores (TC), i.e.,

$$\mathrm{CU} = \frac{\mathrm{AC}}{\mathrm{TC}}. \tag{13}$$

6.6.4 Node utilization measure

A system node is considered active if it has at least one active core. The system's node utilization NU is a measure of the percentage of active nodes in the system (nodes with at least one task is running on the node). Node utilization is calculated simply as a ratio of the number of nodes with at least one active core (active nodes denoted AN) to the total number of nodes in the system (denoted TN), i.e.,

$$\mathrm{NU} = \frac{\mathrm{AN}}{\mathrm{TN}}. \tag{14}$$

6.6.5 Core utilization of active nodes measure

The core utilization of active nodes (CUAN) is a system measure that examines how consolidated tasks are in the system by examining how "full" active nodes are in the

system. If the tasks in the system are consolidated, then the value of CUAN is high, and if the tasks in the system are spread out across processor nodes, then the value of CUAN is low. The CUAN metric is calculated as the sum over all active nodes in the system of the number of active cores in each active node ACIN_i divided by the total number of cores in that active node TCIN_i. This sum is then divided by the total number of active nodes (AN), i.e.,

$$\text{CUAN} = \frac{\sum_{i=1}^{\text{AN}} \frac{\text{ACIN}_i}{\text{TCIN}_i}}{\text{AN}}. \tag{15}$$

Because the simulated HPC system used in this study is homogeneous with each node consisting of a 4-core processor, $TCIN_i$ will always be equal to four in this system.

6.7 Experimental setup

Simulations were performed at two different task subscription levels. The total number of tasks that arrive at the simulated system for each subscription level remains a constant value of 8000 tasks in each simulation. This way, the only difference between simulations in one subscription level and another is the $\frac{1}{\lambda}$ values associated with the expected arrival times of the tasks (described in Sect. 6.4). The subscriptions levels are defined as oversubscribed with $\frac{1}{\lambda} = 0.1$ and undersubscribed with $\frac{1}{\lambda} = 0.25$. The oversubscribed system has tasks arriving to the system at a higher rate than the system can execute them.

The subscription levels provide a good range of situations in which an HPC system could be operating. The resulting performance of each of the scheduling heuristics for each of these subscriptions levels provides a good assessment of when the system benefits from a co-location aware scheduling heuristic.

After the arrival pattern and deadlines of the tasks are determined according to Sect. 6.4 for each of the subscription levels, the 500 node system is simulated with each of the three scheduling heuristics described in Sect. 6.5 (co-location naïve, co-location aware, and "perfect prediction") for a total of six simulated system scenarios.

6.8 Experimental results

Simulation results for demonstrating the utility of the modeling methodology are shown in Figs. 5 and 6 for each of the subscription levels discussed in the experimental setup from Sect. 6.7. The most important result from the simulation is its demonstration of the utility of our co-location interference modeling methodology. For both subscription levels, the heuristic performance results shown in Figs. 5a and 6a indicate that the co-location aware heuristic is able to complete more tasks than the co-location naïve heuristic by avoiding missing task deadlines, while at the same time, performing competitively with the perfect prediction heuristic.

The performance of the "perfect prediction" heuristic in Figs. 5a and 6a further demonstrates how many of the tasks fail to be completed because of a missed deadline (due to an inaccurate execution time prediction) or fail to be completed because

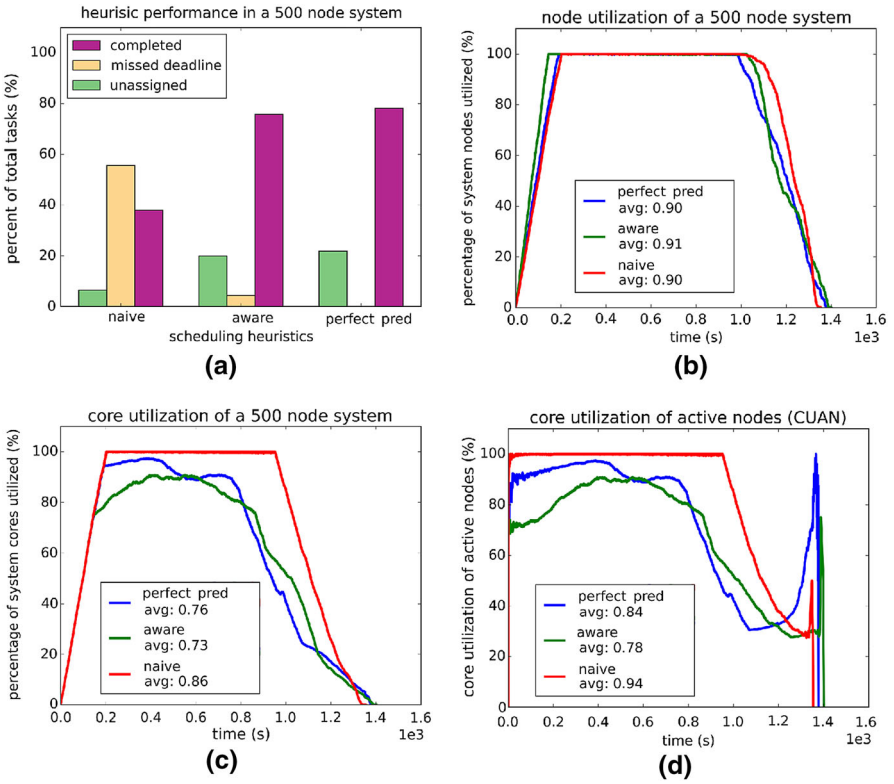

Fig. 5 Simulation results of an *oversubscribed* 500 node homogeneous system comprised 4-core Intel Xeon E3-1225v3 processors. **a** Scheduling heuristic performance. **b** Node utilization of the simulated system. **c** Core utilization of the simulated system. **d** Core utilization of active nodes (CUAN) of the simulated system. In (*a*), the *purple bar* shows the percentage of total tasks completed, the *brown bar* shows the percentage of total tasks that missed their deadlines, and the *green bar* shows the percentage of tasks that were left unassigned. In (**b**, **c**, **d**) the *red line* indicates the naïve heuristic utilizations, the *green line* indicates the co-location aware heuristic utilizations, and the *blue line* indicates the perfect prediction utilizations (color figure online)

they are not able to be scheduled in the system. As expected, the co-location aware scheduling heuristic with "perfect prediction" does not have a single task that misses its deadline for either of the two task subscription levels. However, even for the "perfect prediction" scheduling heuristic, tasks that are executed in an oversubscribed system (shown in Fig. 5a) are sometimes unassigned, indicating that it would never be possible to complete more than approximately 78 % of the tasks that arrive to the system. Given this upper bound on performance, our co-location aware scheduling heuristic performs very well. In both subscription level scenarios where we use our co-location aware scheduling heuristic, the number of completed tasks for our heuristic comes within 3% of the heuristic with "perfect prediction."

It can be observed from the results for the three utilization metrics of each heuristic shown in Figs. 5 and 6 that there is a trade-off for each heuristic between the subscription level of the tasks in the simulated system and each heuristic's ability to

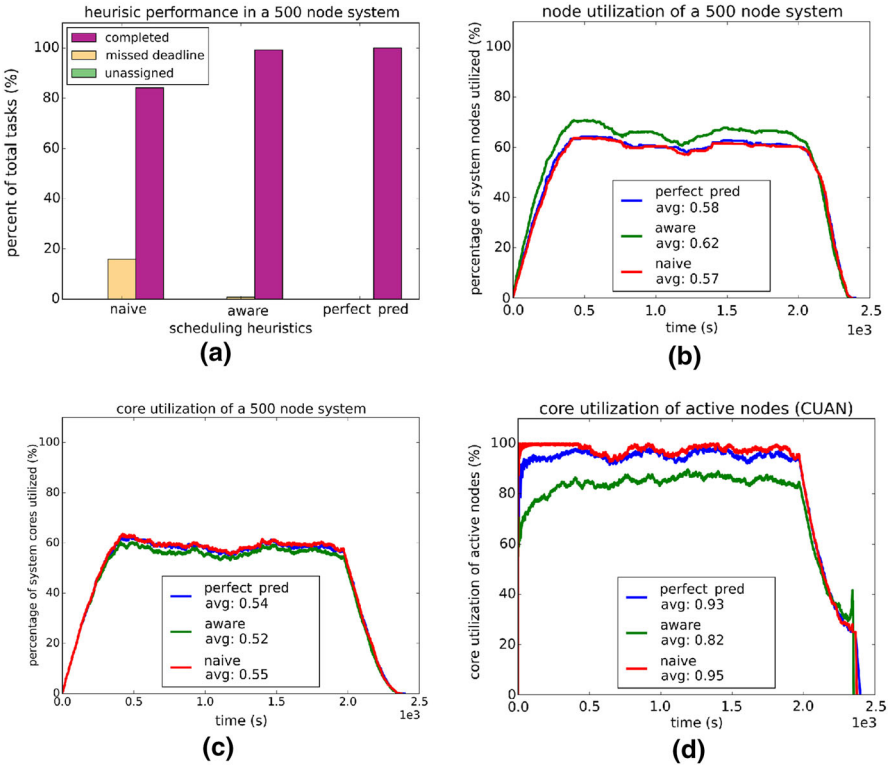

Fig. 6 Simulation results of an *undersubscribed* 500 node homogeneous system comprised 4-core Intel Xeon E3-1225v3 processors. **a** Scheduling heuristic performance. **b** Node utilization of the simulated system. **c** Core utilization of the simulated system. **d** Core utilization of active nodes (CUAN) of the simulated system. In **(a)**, the *purple bar* shows the percentage of total tasks completed, the *brown bar* shows the percentage of total tasks that missed their deadlines, and the *green bar* shows the percentage of tasks that were left unassigned. In **(b, c, d)** the *red line* indicates the naïve heuristic utilizations, the *green line* indicates the co-location aware heuristic utilizations, and the *blue line* indicates the perfect prediction utilizations (color figure online)

consolidate tasks in the system. For both subscription levels, the co-location aware heuristic provides worse consolidation than the co-location naïve heuristic. This is to be expected because the co-location aware heuristic specifically leaves processor cores idle if scheduling tasks to those cores would cause other tasks in the system to miss their deadlines. By examining the oversubscribed system in Fig. 5b, c, d, it can be observed that the co-location naïve heuristic aggressively consolidates tasks in the system because the node utilization, core utilization, and core utilization of active nodes (CUAN) are all close to 100 % for the majority of the system simulation. In contrast, the co-location aware heuristic still takes advantage of all the nodes in the system (node utilization is still at 100 %), but is more selective about its use of cores and, therefore, has core utilization, and CUAN values that peak at about 90 % and do not remain constant throughout the simulation. Similar conclusions can be drawn from the results for the undersubscribed system shown in Fig. 6. In this system, the

co-location naïve heuristic still has the highest levels of core utilization and CUAN, but has a *lower* node utilization, indicating that it is more aggressively consolidating tasks in the system due to its inability to foresee the potential problems that arise when tasks are co-located on the same multicore processor.

7 Conclusions

We proposed a modeling methodology that predicts application execution time and energy use when under co-location interference effects caused by resource sharing among cores in a multicore processor. The methodology is general enough to be applied to any multicore processor and set of applications. To validate our methodology, its effectiveness was demonstrated by applying it to three server class Intel Xeon multicore processors with up to 12 cores, executing real data workloads from two scientific benchmark suites. After validation, the utility that such prediction models can provide was demonstrated by creating a scheduling heuristic that takes advantage of the proposed modeling methodology for its scheduling decisions.

Specifically, this work used machine learning techniques to make predictions about application performance degradation due to contention in shared cache and main memory resources when multiple applications were co-located on the same multicore processor. While simpler linear models do not provide a significant increase in prediction accuracy from the addition of application memory use information, the results from Figs. 1 and 2 show that neural networks can provide very accurate predictions of application execution time and energy use. For the neural network model, when using only a subset of the application features, it is still able to produce fairly accurate performance predictions. Using all the features, the neural network achieved a very small MPE of 2 % and an NRMSE of around 0.01 on all processors tested. Considering that performance degradation due to co-location can extend an application's execution time quite significantly, even a model with access to only a limited set of model features may be able to provide good enough predictions to improve the performance of schedulers in HPC systems.

Applying our methodology to create models for a large-scale simulated system enabled us to examine the benefit of a co-location aware scheduling heuristic that can provide substantial performance improvement in a simulated homogeneous 500 node system. Although the experiments in Sect. 6 were only performed for a homogeneous system of 4-core nodes, it is reasonable to expect that the benefits demonstrated can be replicated in either a homogeneous or heterogeneous system that includes processor nodes with more cores. Not only is the interference from co-location that applications experience likely to be greater in machines with more cores, but also the results shown in Fig. 1 indicate that for the more advanced feature sets the prediction models tend to perform even better for the 6-core and 12-core systems than they do for the 4-core system used for those experiments.

Acknowledgments The authors thank Mark Oxley for his valuable comments on this research. 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. Pacific Northwest National Laboratory is operated by Batelle for the U.S. Department of Energy under contract DE-AC0576RL01830. A preliminary version of portions of this work appeared in [40]. The additions to this work include creating an additional set of models for energy use prediction, validating the execution time and energy use prediction models on an additional multicore processor, and creating and analyzing a co-location aware scheduling heuristic that utilizes prediction models generated by our modeling methodology for making intelligent co-location decisions.

References

1. Verma A, Ahuja P, Neogi A (2008) Power-aware dynamic placement of HPC applications. In: 22nd Annual International Conference on Supercomputing (ICS '08), pp 175–184
2. Zhu Q, Zhu J, Agrawal G (2010) Power-aware consolidation of scientific workflows in virtualized environments. In: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '10), pp 1–12
3. Tang L, Mars J, Vachharajani N, Hundt R, Soffa M (2011) The impact of memory subsystem resource sharing on datacenter applications. In: 38th Annual International Symposium on Computer Architecture (ISCA '11), pp 283–294
4. Sandberg A, Sembrant A, Hagersten E, Black-Schaffer D (2013) Modeling performance variation due to cache sharing. In: IEEE 19th International Symposium on High Performance Computer Architecture (HPCA '13), pp 155–166
5. Choi J, Dukhan M, Liu X, Vuduc R (2014) Algorithmic time, energy, and power on candidate HPC compute building blocks. In: IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14), pp 447–457
6. Dauwe D, Friese R, Pasricha S, Maciejewski AA, Koenig GA, Siegel HJ (2014) Modeling the effects on power and performance from memory interference of co-located applications in multicore systems. In: The 2014 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '14), pp 3–9
7. Subramanian L, Seshadri V, Ghosh A, Khan S, Mutlu O (2015) The application slowdown model: quantifying and controlling the impact of inter-application interference at shared caches and main memory. In: 48th International Symposium on Microarchitecture (MICRO-48 '15), pp 62–75
8. Merkel A, Stoess J, Bellosa F (2010) Resource-conscious scheduling for energy efficiency on multicore processors. In: 5th European Conference on Computer Systems (EuroSys '10), pp 153–166
9. Luque C, Moreto M, Cazorla FJ, Gioiosa R, Buyuktosunoglu A, Valero M (2012) CPU accounting for multicore processors. IEEE Trans Comput 61(2):251–264
10. Mars J, Tang, L, Hundt R, Skadron K, Soffa M (2011) Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In: IEEE/ACM 44th International Symposium on Microarchitecture (MICRO '11), pp 248–259
11. Dwyer T, Fedorova A, Blagodurov S, Roth M, Gaud F, Pei J (2013) A practical method for estimating performance degradation on multicore processors, and its application to HPC workloads. In: ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12), pp 83:1–83:11
12. Cazorla FJ, Ramirez A, Valero M, Fernandez E (2004) Dynamically controlled resource allocation in SMT processors. In: 37th International Symposium on Microarchitecture (MICRO-37 '04), pp 171–182
13. De Vuyst M, Kumar R, Tullsen DM (2006) Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In: IEEE 20th International Parallel and Distributed Processing Symposium (IPDPS '06), pp 10–20
14. Feliu J, Sahuquillo J, Petit S, Duato J (2015) Addressing fairness in SMT multicores with a progress-aware scheduler. In: IEEE 29th International Parallel and Distributed Processing Symposium (IPDPS '15), pp 187–196
15. Young BD, Apodaca J, Briceño LD, Smith J, Pasricha S, Maciejewski AA, Siegel HJ, Khemka B, Bahirat S, Ramirez A, Zou Y (2013) Deadline and energy constrained dynamic resource allocation in a heterogeneous computing environment. J Supercomput 63(2):326–347
16. Al-Qawasmeh AM, Pasricha S, Maciejewski AA, Siegel HJ (2015) Power and thermal-aware workload allocation in heterogeneous data centers. IEEE Trans Comput 64(2):477–491

17. Khemka B, Friese R, Pasricha S, Maciejewski AA, Siegel HJ, Koenig GA, Powers S, Hilton M, Rambharos R, Poole S (2015) Utility maximizing dynamic resource management in an oversubscribed energy-constrained heterogeneous computing system. Sustain Comput Inf Syst 5:14–30
18. Oxley M, Pasricha S, Maciejewski AA, Siegel HJ, Apodaca J, Young D, Briceño L, Smith J, Bahirat S, Khemka B, Ramirez A, Zou Y (2015) Makespan and energy robust stochastic static resource allocation of bags-of-tasks to a heterogeneous computing system. IEEE Trans Parallel Distrib Syst 2791–2805
19. Talby D, Feitelson DG (1999) Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In: 13th International Parallel Processing Symposium (IPPS '99), pp 513–517
20. Sadhasivam S, Nagaveni N, Jayarani R, Ram RV (2009) Design and implementation of an efficient two-level scheduler for cloud computing environment. In: International Conference on Advances in Recent Technologies in Communication and Computing (ARTCom '09), pp 884–886
21. Utrera G, Corbalan J, Labarta J (2014) Scheduling parallel jobs on multicore clusters using CPU oversubscription. J Supercomput 68(3):1113–1140
22. Lifka DA (1995) The ANL/IBM SP scheduling system. In: Job scheduling strategies for parallel processing, pp 295–303
23. Jolliffe I (2002) Principal component analysis. Wiley, Hoboken, NJ
24. Chong EK, Zak SH (2013) An introduction to optimization. Wiley, Hoboken, NJ
25. LeCun YA, Bottou L, Orr GB, Müller K (2012) "Efficient backprop", neural networks: tricks of the trade. Springer, New York
26. Bishop CM (2006) Pattern recognition and machine learning. Springer, New York, NY
27. Ubuntu 14 Release Notes. https://wiki.ubuntu.com/TrustyTahr/ReleaseNotes. Accessed Jan 2016
28. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 1,2A,2B,2C,3A,3B,3C and 3D, Technical Report 2015. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462. Accessed Jan 2016
29. Intel Xeon E3-1225v3 Processor http://ark.intel.com/products/75461/. Accessed Jan 2016
30. Intel Xeon E5649 Processor http://ark.intel.com/products/52581/. Accessed Jan 2016
31. Intel Xeon E5-2697v2 Processor http://ark.intel.com/products/75283/. Accessed Jan 2016
32. Performance application programming interface http://icl.cs.utk.edu/papi/. Accessed Jan 2016
33. HPCToolkit http://hpctoolkit.org/. Accessed Jan 2016
34. Watts Up? Plug Load Meters https://www.wattsupmeters.com/secure/products.php?pn=0. Accessed Jan 2016
35. PARSEC Benchmark Suite http://parsec.cs.princeton.edu/. Accessed Jan 2016
36. NAS Parallel Benchmarks http://www.nas.nasa.gov/publications/npb.html. Accessed Jan 2016
37. Efron B, Tibshirani RJ (1994) An introduction to the bootstrap. CRC Press, New York, NY
38. Khemka B, Friese R, Pasricha S, Maciejewski AA, Siegel HJ, Koenig GA, Powers S, Hilton M, Rambharos R, Wright M, Poole S (2015) Comparison of energy-constrained resource allocation heuristics under different task management environments. In: The 2015 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2015), pp 3–12
39. Khemka B, Friese R, Briceno LD, Siegel HJ, Maciejewski AA, Koenig GA, Groer C, Okonski G, Hilton MM, Rambharos R, Poole S (2015) Utility functions and resource management in an oversubscribed heterogeneous computing environment. IEEE Trans Comput 64(8):2394–2407
40. Dauwe D, Jonardi E, Friese R, Pasricha S, Maciejewski AA, Bader DA, Siegel HJ (2015) A methodology for co-location aware application performance modeling in multicore computing. In: 17th Workshop on Advances on Parallel and Distributed Computing Models (APDCM '15), pp 434–443