Accepted Manuscript

Journal of Circuits, Systems, and Computers

Article Title:	Regression-Based Prediction for Task-Based Program Performance
Author(s):	Isil Oz, Muhammad Khurram Bhatti, Konstantin Popov, Mats Brorsson
DOI:	10.1142/S0218126619500609
Received:	15 October 2017
Accepted:	21 May 2018
To be cited as:	Isil Oz *et al.*, Regression-Based Prediction for Task-Based Program Performance, *Journal of Circuits, Systems, and Computers*, doi: 10.1142/S0218126619500609
Link to final version:	https://doi.org/10.1142/S0218126619500609

Journal of Circuits, Systems, and Computers

Regression-Based Prediction for Task-Based Program Performance

Isil Oz
Computer Engineering Department, Izmir Institute of Technology, Izmir, Turkey

Muhammad Khurram Bhatti
Information Technology University, Lahore, Pakistan

Konstantin Popov
SICS Swedish ICT AB, Stockholm, Sweden

Mats Brorsson
KTH Royal Institute of Technology, Stockholm, Sweden

As multicore systems evolve by increasing number of parallel execution units, parallel programming models have been released to exploit parallelism in the applications. Task-based programming model uses task abstractions to specify parallel tasks and schedules tasks onto processors at runtime. In order to increase the efficiency and get the highest performance, it is required to identify which runtime configuration is needed and how processor cores must be shared among tasks. Exploring design space for all possible scheduling and runtime options, especially for large input data, becomes infeasible and requires statistical modeling. Regression-based modeling determines the effects of multiple factors on a response variable, and makes predictions based on statistical analysis.

In this work, we propose a regression-based modeling approach to predict task-based program performance for different scheduling parameters with variable data size. We execute a set of task-based programs by varying runtime parameters, and conduct a systematic measurement for influencing factors on execution time. Our approach uses executions with different configurations for a set of input data, and derives different regression models to predict execution time for larger input data. Our results show that regression models provide accurate predictions for validation inputs with mean error rate as low as 6.3%, and 14% in average among four task-based programs.

Keywords: performance prediction, task-based programs, regression

1. Introduction

With the increasing availability of larger multicore architectures, parallel programming models have been emerged to exploit the performance out of parallel architectures and to provide easier programmability for high-performance systems. Task-based programming model allows programmer to divide computation into task abstractions and maintains dynamic scheduling techniques for load imbalance.[1,2,3,4] For instance, OpenMP implements task parallelism by using explicit *task* constructs as parallel work units.[3]

In order to get the highest performance for task-based programs processing large data and to use parallel resources efficiently, it is required to identify which runtime configuration is needed and how processor cores must be shared among tasks. For example, Figure 1 demonstrates execution time variability for different data sizes of *fft* program. The configurations include a set of thread numbers and scheduling strategies. Exploring design space for all possible scheduling and runtime options, especially for large data, becomes infeasible and requires estimations for execution time.

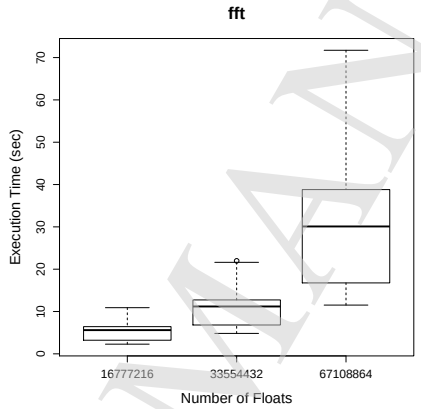


Fig. 1. Execution time deviations for task-based *fft* program with different configurations for a set of data size.

Techniques in statistical inference are becoming increasingly popular for performance prediction purposes[5,6,7,8]. Statistical models offer comparable prediction results with a small set of design space exploration. Building an accurate statistical model for performance prediction requires a systematic and detailed analysis by considering influencing factors and factor interactions[9]. While omitting important factors causes bias on estimation results, inclusion of too many factors complicates the performance model.

In this paper, we propose a regression-based approach to predict task-based program performance for different scheduling parameters with variable data size. Our approach uses executions with different configurations for a set of input data, and presents different regression models to predict execution time for larger data. We can summarize the main contributions of this work as follows:

- We present a set of factors that affect task-based program performance, and derive regression models to predict execution times. Our regression models use execution times for a set of input data by varying the values of factors, and predict the performance for larger data.

- We conduct experiments for a set of task-based OpenMP programs from BOTS benchmark suite[10] to apply the regression models. We analyze the accuracy of proposed regression models on training data, and select the models with the lowest mean percentage error to apply on larger data.
- We apply the selected regression models on larger input data for the target programs, and predict the execution times for given configurations. Our experimental evaluation shows that regression models predict performance for validation inputs with mean error rate as low as 6.3%, and 14% in average among four task-based programs.

The remainder of this paper is structured as follows: Section 2 provides a formal definition of our target problem and the proposed method. Section 3 gives the details of our regression-based approach for task-based program performance analysis, and experimental methodology to apply the regression models is presented in Section 4. Section 5 presents the experimental evaluation and discussion about the results. We discuss related work in Section 6 and conclude the paper in Section 7.

2. Problem Formulation

2.1. *System Model*

We consider a parallel system running task-based parallel programs with a given configuration. The programs can be executed by different runtime parameters and different input sizes, and perform computations in a time interval:

$$T = P(C), \tag{1}$$
$$C = (I = i, R_1 = a1, R_2 = b1, R_3 = c1, ..., R_n = z1),$$

where T is the execution time of the task-based program P executed for the given configuration C, and I is the input size, R_1, R_2,..., R_n are the runtime parameters with the assigned values for each parameter. If we run fft application by using Breadth-First (bf) scheduler for 16777216 data points, we can have the representation for each specific execution:

$$T = fft(I = 16777216, Scheduler = bf) \tag{2}$$

In order to get the highest performance for task-based programs processing large data and to use parallel resources efficiently, we want to decide which runtime configuration is needed. Since exploring design space for all possible runtime options is not infeasible, we aim to have a prediction mechanism for the execution times of task-based programs. We want to have a prediction for the execution time of a specific configuration before executing it.

2.2. *Objective*

In this paper, we propose a regression-based performance prediction method to predict the execution times of task-based programs. We assume that we have a

set of execution times for different runtime configurations for a program, and our aim is to get the execution time for a large input set for the given runtime options without executing the program.

For instance, given that T_1, T_2, T_3 for the configurations C_1, C_2, C_3 respectively, we want to have an execution time prediction $\hat{\mathrm{T}}$ for the specified configuration C_x, which is not the same as C_1, C_2, or C_3.

$$\text{Given that } T_1 = P(C_1), T_2 = P(C_2), T_3 = P(C_3) \qquad (3)$$
$$\text{Find } \hat{\mathrm{T}} \text{ for any } C_x$$

3. Regression-Based Performance Prediction

In this section, we describe our regression-based approach for performance prediction of task-based programs. We explain the motivation behind the regression-based approach for performance prediction, then present a brief information about our method. Then we describe the input variables included in our analysis, and finally we detail our regression model derivation.

3.1. *Motivation*

Given large runtime parameter options, we need to predict the execution time of task-based programs. We use regression-based approach due to its practical and feasible charactarestics for our purpose. The other prediction approaches can fail for our problem. For instance, analytical model construction requires exhaustive analysis of each runtime option, and this is not very practical for runtime systems of task-based programs. Although the simulation is an effective approach for performance prediction, the simulation of the runtime system is not straightforward[11]. Moreover, the simulation time for a detailed simulation including all runtime considerations becomes impractical especially for large input sets. Therefore, we select regression-based approach, which is both practical and feasible. It relies on statistical analysis which shows the relationship between the parameters and application performance, and only requires an initial set of data for model training, obtained from a small set of measurements in the larger runtime options. We construct predictive models for smaller input sets, and predict the execution times for large input data.

3.2. *Method Overview*

Regression analysis is a statistical technique to estimate the value of a dependent variable given the independent variables.[12] For example, a linear regression model can be built by assuming that two variables (x_1 and x_2) determine the response variable (y):

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon, \qquad (4)$$

where y is dependent variable, x_1 and x_2 are independent variables, β_0, β_1, and β_2 are estimated parameters, and ϵ is an error term. This model assumes that two factors affect the response, and they do not interact with each other. The accuracy of the regression-based prediction relies on the model construction. While omitting important factors in the model causes bias on estimation results, inclusion of too many factors complicates performance model.

Our regression-based method first derives a model based on a training set of previously executed configurations and their measured performance. It then predicts performance for unknown configuration based on this model. Each configuration consists of runtime parameters, that are individually measurable properties. By using the model formulation given in Equation 4, we can define independent variables x_1, x_2, ..., x_n for each runtime parameter, and treat execution time as a separate dependent variable y.

$$T = \beta_0 + \beta_1 R_1 + \beta_2 R_2 + ... + \beta_n R_n + \epsilon, \tag{5}$$

where T is the execution time, and R_1, R_2,..., R_n are the runtime parameters.

In this work, we make some assumptions. First, we assume that input data for the target program can be specified by its size or can be measured quantitatively. Since executing larger problems takes time and exploring complete design space becomes infeasible, our approach proposes a model to make predictions for larger input data by executing small-size inputs. We further assume that we know which variables contribute significantly to execution time. Our models are defined by some function of these variables. We consider the variables given in Section 3.3 in our regression model, and try to avoid errors from different executions by including results from multiple runs in the training data. Finally, we assume that a program can be run using any configuration of the input variables.

3.3. *Input Variables*

We aim to estimate execution times for task-based programs with different configurations to be able to determine runtime parameters for performance prediction. In this work, we focus on OpenMP programs implemented by explicit task constructs, and deal with task-centric parameters. We include scheduler, cut-off policy, number of threads, and input size as factors in our regression model, and predict the execution time of target program for a given configuration.

3.3.1. *Task Scheduler*

Scheduling of tasks plays an important role in the execution of task-based programs running on multicore systems.[13] The scheduling policy defines the order of execution of tasks and the resource where each task will be executed. The choice for assigning the tasks to the available threads/cores in the system becomes critical especially for irregular applications with heterogeneous tasks. Queueing and work-stealing strategies also induce performance impacts on the program executions.

We consider different combinations of schedulers and queuing policies provided by Nanos++ runtime system.[14] Mainly, we explore Breadth-First (bf) and Work-First (wf) schedulers with specific options.[13]

- **Breadth-First (bf):** This scheduler policy only implements a single/global ready queue. When the parent task spawns children, all ready child tasks are placed into this ready queue (ordered as FIFO by default) and execution of parent task continues. All successor tasks are generated immediately before a thread executes tasks from the next level.
- **Work-First (wf):** This scheduler policy implements a local ready queue per thread. When the parent task creates a task, the execution continues with the new created task by leaving the parent task into current threads ready queue.

Table 1 presents the schedulers used in our evaluation. *Steal Parent* option enables wf scheduler to steal parent task if there is no task in the local ready queue, which makes the scheduler equivalent to cilk scheduler.[1] *Queue Access for Ready Queue* option defines queue access algorithm for the ready queue (i.e., global queue for bf scheduler, local queue for wf scheduler), while *Queue Access for Stealing* option defines the algorithm for stealing. A queue is ordered following a FIFO (First In First Out) or LIFO (Last In Last Out) algorithm.

Table 1. Schedulers used in our evaluation.

Scheduler Name	Scheduler Type	Steal Parent	Queue Access for Ready Queue	Queue Access for Stealing
bff	Breadth-First	-	FIFO	-
bfl	Breadth-First	-	LIFO	-
wfpff	Work-First	Yes	FIFO	FIFO
wfpfl	Work-First	Yes	FIFO	LIFO
wfplf	Work-First	Yes	LIFO	FIFO
wfpll	Work-First	Yes	LIFO	LIFO
wfff	Work-First	No	FIFO	FIFO
wffl	Work-First	No	FIFO	LIFO
wflf	Work-First	No	LIFO	FIFO
wfll	Work-First	No	LIFO	LIFO

3.3.2. *Cut-off Policy*

Several works have proposed how to reduce the overhead of task creation by means of using cut-off policies.[13,15] To limit the size of the number of task creations, Nanos++ runtime system provides an internal cut-off strategy (throttling). We include three throttling policies in our evaluation:[14]

- **Task depth:** This throttle policy takes the decision according to the depth

of the task. If the maximum nested level reaches a fixed limit, the runtime does not create new tasks.

- **Ready tasks:** This throttle policy takes the decision according to the number of ready tasks. If the number of ready tasks in the system is greater than a fixed limit, the runtime does not create new tasks.
- **Number of tasks:** This throttle policy takes the decision according to the existing number of tasks. If the number of tasks in the system is greater than a fixed limit, the runtime does not create new tasks.

Table 2 presents the throttle mechanisms and throttling limit values used in our evaluation.

Table 2. Throttle policies used in our evaluation.

Throttle Name	Throttle Type	Throttle Limit
taskdepth	Task depth	4
readytasks	Ready tasks	100
numtasks	Number of tasks	100

Figure 2 presents execution time values for different scheduler/throttle pairs for task-based *fft* program executed with 2-threads. While program performance stays stable for *taskdepth* throttle policy, it exhibits different behavior for varying schedulers if throttle policy is *readytasks* or *numtasks*. This example case shows that schedulers may affect the program performance in a different way if throttle policies vary in the execution.

3.3.3. *Number of Threads*

The number of threads identifies the parallel program performance for most of the applications.[16,17] Since parallelism makes use of resources in a multicore system, the performance of the applications with higher degree of parallelism increases as the number of threads/cores increases. On the other hand, intensity of shared cache contention rises with the number of threads. Since threads, which are running simultaneously, request cache lines at the same time, this contention incurs significant performance penalties if they work on different data. We include *two-thread*, *four-thread*, and *eight-thread* execution cases for our evaluation. We assign one thread per core in a multicore architecture, and use *thread* and *core* interchangeably throughout the paper.

While we include those factors in our analysis, there are other potential parameters influencing parallel program performance. Most importantly, memory/cache

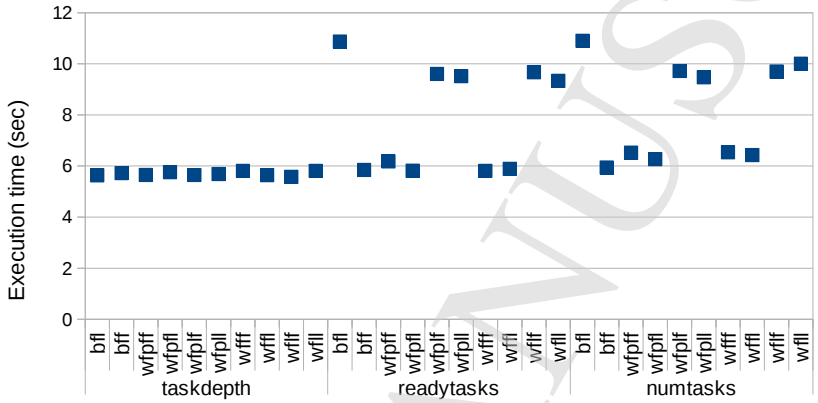


Fig. 2. Performance values for *fft* program with different scheduler/throttle configurations, 2-thread execution for Input Size=16777216.

hierarchy may affect execution time if tasks have strong contention for memory resources. On the other hand, cache sharing does not impact the performance for some parallel workloads.[18] In this work, we do not aim to analyze the potential performance effect of the memory, so we assume that our system has enough memory resources for parallel programs, and do not include memory-related factors in our analysis.

3.4. *Model Derivation*

We derive a regression model, by including four variables affecting performance of task-based programs, and try to determine the execution time as including it as the response variable in our model. We can formulate the proposed model as follows:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \epsilon, \tag{6}$$

where y is the execution time of the program, x_1 is the input size, T is the number of threads, S is the scheduler type, and C is the cut-off policy.

We propose regression models using several runs of the same program with smaller input sets, and make performance prediction for larger input execution cases. We vary the values of input variables for a set of small-size input, where runs take reasonable times for different configurations. We derive regression models by using relationship between input variables and execution time to predict execution time for larger input data. Our regression models are functions of the independent

variables:

$$ET = F(I, T, S, C), \tag{7}$$

where ET is the execution time of the program, I is the input size, T is the number of threads, S is the scheduler type, and C is the cut-off policy. While the input size and the number of threads are quantitative factors in the model, the scheduler type and the cut-off policy values are included as qualitative factors.

3.4.1. *ANOVA Analysis*

First, we apply Analysis of variance (ANOVA) to determine the factors and factor interactions to be included in the regression model for performance prediction.[12] ANOVA, which is a collection of statistical models, provides information about how much each factor and interaction between factors contribute to variance of data. The fundamental technique relies on a partitioning of the total sum of squares (SS) into components related to the effects used in the model. For example, the model for two-factor ANOVA at different levels can be presented as follows:

$$\begin{aligned} SS_{Total} &= SS_A + SS_B + SS_{AB} + SS_E \\ \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}(y_{ijk} - \overline{y}_{...})^2 &= bn\sum_{i=1}^{a}(\overline{y}_{i..} - \overline{y}_{...})^2 + an\sum_{j=1}^{b}(\overline{y}_{.j.} - \overline{y}_{...})^2 + \\ & n\sum_{i=1}^{a}\sum_{j=1}^{b}(\overline{y}_{ij.} - \overline{y}_{i..} - \overline{y}_{.j.} - \overline{y}_{...})^2 + \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}(y_{ijk} - \overline{y}_{ij.})^2, \end{aligned} \tag{8}$$

where y represents each observation, a, b, and n represent the number of levels for Factor A and Factor B, and the number of replications for each factor level, respectively. This equation indicates that total variance (SS_{Total}) may be explained by the effect of Factor A (SS_A), Factor B (SS_B), the interaction between Factor A and B (SS_{AB}), and the experimental error within observations (SS_E).

3.4.2. *Regression Models*

After screening out unimportant factor interactions, regression model can be employed to determine a relationship between the factors and response variables. Firstly, we consider that main factors and 2-factor interactions contribute to performance deviation and derive models based on this assumption without further analysis. We also derive models by using ANOVA results to determine the factors to be included in the model. In order to get better fits, we use log transformation for quantitative variables.[19] While $log(x)$ represents the log transformed value of the quantitative factor x, $factor(y)$ represents the class of the qualitative factor y. If the model considers factor interactions (simultaneous influence of two variables), it includes a term represented by the multiplication of two factors (e.g. $factor(x) * factor(y)$) Mainly, we derive six regression models:

- **Log transformed linear model (modeLog):** Linear model with response variable and quantitative factors transformed, main factors only.

$$log(ET) \sim (log(I) + log(T) + factor(S) + factor(C)).$$

- **Log transformed quadratic model (modelQuad):** Log transformed model with quadratic terms for quantitative factors included.

$$log(ET) \sim (log(I) + (log(I))^2 + log(T) + (log(T))^2 + factor(S) + factor(C)).$$

- **2-factor interaction model (model2):** Log transformed model with 2-factor interactions included.

$$\begin{aligned} log(ET) \sim (&log(I) + log(T) + factor(S) + factor(C) + \\ &log(I) * log(T) + log(I) * factor(S) + log(I) * factor(C) + \\ &log(T) * factor(S) + log(T) * factor(C) + factor(S) * factor(C)). \end{aligned}$$

- **All interactions model (modelAll):** Log transformed model with all 2-factor, 3-factor, 4-factor interactions included.

$$\begin{aligned} log(ET) \sim (&log(I) + log(T) + factor(S) + factor(C) + \\ &log(I) * log(T) + log(I) * factor(S) + log(I) * factor(C) + \\ &log(T) * factor(S) + log(T) * factor(C) + factor(S) * factor(C) + \\ &log(I) * log(T) * factor(S) + log(I) * log(T) * factor(C) + \\ &log(I) * factor(S) * factor(C) + log(T) * factor(S) * factor(C) + \\ &log(I) * log(T) * factor(S) * factor(C)). \end{aligned}$$

- **Anova interactions model (modelAov):** Log transformed model with factor interactions that are significant in ANOVA analysis included.
- **Anova 2-factor interaction model (modelAov2):** Log transformed model with 2-factor interactions that are significant in ANOVA analysis included.

3.4.3. *Model Evaluation*

We compare regression models for each program, and apply the model with minimum prediction error on the unseen input sets (i.e., validation data). To assess the model accuracy, we use Root Mean Squared Error (RMSE), Mean Absolute Error

(MAE), and Mean Absolute Percentage Error (MAPE) metrics:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \widehat{y}_i|$$

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \widehat{y}_i}{y_i}\right|$$

where n is the number of configurations, y_i is the observed value of the ith configuration, and $\widehat{y}_i$ is the predicted value of the ith configuration.

We calculate each metric value for training data, and select the models that have the lowest errors to be applied on validation data.

Figure 3 presents the flow of our regression modeling. First, we derive models by using training data gathered from real executions. Then we predict the execution times of the configurations with larger data by using the best model with minimum error rate.

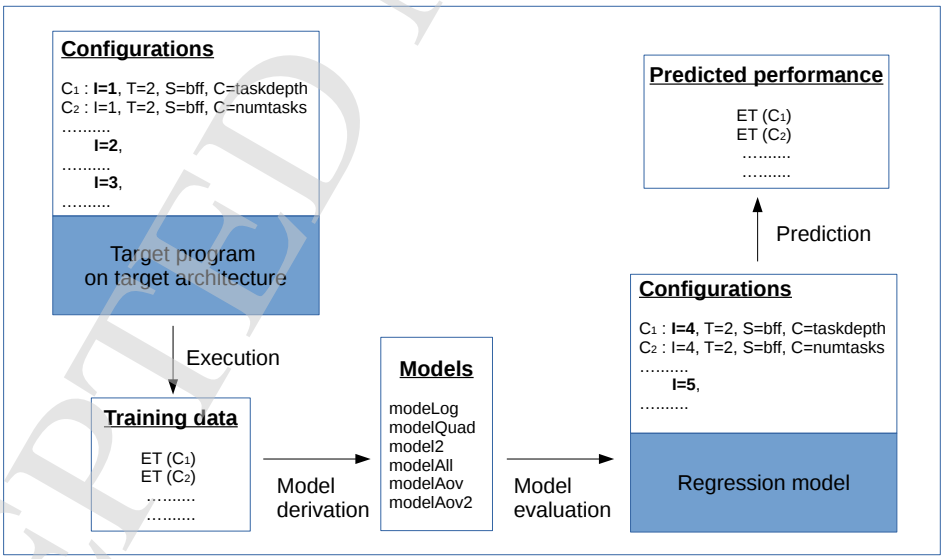


Fig. 3. Our regression modeling flow.

4. Experimental Methodology

In our experimental analysis, we use a set of task-based programs from Barcelona OpenMP Tasks Suite (BOTS).[10] BOTS provides a set of programs using explicit task parallelism and the tasking model in OpenMP 3.0.[20] We select four applications from the benchmark suite including *fft*, *sort*, *sparselu*, and *strassen*.

- ***fft*:** computes recursively the one-dimensional Fast Fourier Transform of a vector of n complex values using the Cooley-Tukey algorithm.
- ***sort*:** sorts a random permutation of n 32-bit numbers with a fast parallel sorting variation of mergesort.
- ***sparselu*:** computes an LU matrix factorization over sparse matrices, with n matrix size.
- ***strassen*:** computes matrix multiplication of large dense matrices using hierarchical decomposition algorithm, with n matrix size.

To evaluate regression models, we use R, a statistical computing environment.[21] We build our modeling framework based on R packages.

We execute our task-based programs on an AMD Opteron 6172 processor-based multicore system. First, we execute each program with three different input sizes (training data) by altering input variables and train regression models. Then, we predict execution times for two other input sets (validation data) for the same configurations. Table 3 presents our training and validation input data sizes for each benchmark program, where input size is represented by n above definitions. The input size represents the number of complex numbers for fft, the number of integers for *sort*, the size of the matrix (nxn blocks) for *sparselu*, and the size of the matrix (nxn elements) for *strassen*.

Table 3. Size of input sets in our evaluation.

Benchmark	Training Input	Validation Input
fft	16777216, 33554432, 67108864	134217728, 268435456
sort	1048576, 2097152, 4194304	8388608, 16777216
sparselu	25, 50, 75	100, 200
strassen	512, 1024, 2048	4096, 8192

5. Experimental Results

To train our regression models, we execute each program for three different input sizes by varying configurations. We use five replications for each case. Namely, for each program, we have $3(input) \times 3(thread) \times 10(scheduler) \times 3(throttle) \times$

$5(replica) = 1350$ runs for training data. After training regression models with those data and comparing the performance of the models, we apply the models with the lowest error to the configurations with larger input size. To assess the performance of the models for this validation data, we also execute programs with larger input data. By comparing prediction results and observed values, we evaluate the accuracy of regression-based approach on performance prediction.

5.1. *ANOVA Results*

First, we apply ANOVA analysis to determine the significance of factors and factor interactions. In our analysis, we include input size (I), number of threads (T), cut-off policy (C), and scheduler (S). We use five replications (*subject*) as the error term in ANOVA equation.

$$\begin{aligned} log(ET) \sim ((&log(I) + log(T) + factor(C) + factor(S) + \\ &log(I) * log(T) + log(I) * factor(C) + log(I) * factor(S) + \\ &log(T) * factor(C) + log(T) * factor(S) + factor(C) * factor(S) + \\ &log(I) * log(T) * factor(C) + log(I) * log(T) * factor(S) + \\ &log(I) * log(T) * factor(S) + log(T) * factor(C) * factor(S) + \\ &log(I) * log(T) * factor(C) * factor(S)) + \\ &Error(factor(subject)/(log(I) + log(T) + factor(C) + factor(S)))). \end{aligned}$$

Table 4. ANOVA table for *fft*.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(I)	1	646.87	646.87	590561.54	0.0000
log(T)	1	124.35	124.35	179231.93	0.0000
factor(C)	2	61.63	30.81	48630.42	0.0000
factor(S)	9	60.12	6.68	6706.07	0.0000
log(I):log(T)	1	0.06	0.06	5.91	0.0152
log(I):factor(C)	2	0.44	0.22	21.89	0.0000
log(I):factor(S)	9	1.34	0.15	14.98	0.0000
log(T):factor(C)	2	3.12	1.56	156.54	0.0000
log(T):factor(S)	9	1.84	0.20	20.50	0.0000
factor(C):factor(S)	18	34.01	1.89	189.84	0.0000
log(I):log(T):factor(C)	2	0.03	0.02	1.68	0.1867
log(I):log(T):factor(S)	9	0.04	0.00	0.49	0.8796
log(T):factor(C):factor(S)	18	1.35	0.07	7.52	0.0000
log(I):factor(C):factor(S)	18	0.69	0.04	3.87	0.0000
log(I):log(T):factor(C):factor(S)	18	0.10	0.01	0.54	0.9379

Table 5. ANOVA table for *sort*.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(I)	1	472.85	472.85	336647.26	0.0000
log(T)	1	142.62	142.62	158877.10	0.0000
factor(C)	2	92.37	46.18	101565.56	0.0000
factor(S)	9	5.36	0.60	1180.45	0.0000
log(I):log(T)	1	0.08	0.08	17.03	0.0000
log(I):factor(C)	2	0.20	0.10	21.03	0.0000
log(I):factor(S)	9	0.40	0.04	9.11	0.0000
log(T):factor(C)	2	7.46	3.73	770.86	0.0000
log(T):factor(S)	9	0.55	0.06	12.73	0.0000
factor(C):factor(S)	18	5.71	0.32	65.60	0.0000
log(I):log(T):factor(C)	2	0.05	0.02	5.05	0.0065
log(I):log(T):factor(S)	9	0.05	0.01	1.14	0.3340
log(T):factor(C):factor(S)	18	1.24	0.07	14.28	0.0000
log(I):factor(C):factor(S)	18	0.34	0.02	3.88	0.0000
log(I):log(T):factor(C):factor(S)	18	0.06	0.00	0.68	0.8362

Table 6. ANOVA table for *sparselu*.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(I)	1	2057.47	2057.47	149583253.79	0.0000
log(T)	1	297.10	297.10	42018340.39	0.0000
factor(C)	2	0.00	0.00	70.75	0.0000
factor(S)	9	4.19	0.47	418261.13	0.0000
log(I):log(T)	1	1.97	1.97	1346.67	0.0000
log(I):factor(C)	2	0.00	0.00	0.03	0.9738
log(I):factor(S)	9	0.38	0.04	28.81	0.0000
log(T):factor(C)	2	0.00	0.00	0.01	0.9931
log(T):factor(S)	9	1.68	0.19	128.10	0.0000
factor(C):factor(S)	18	0.00	0.00	0.02	1.0000
log(I):log(T):factor(C)	2	0.00	0.00	0.00	0.9962
log(I):log(T):factor(S)	9	0.15	0.02	11.68	0.0000
log(T):factor(C):factor(S)	18	0.00	0.00	0.00	1.0000
log(I):factor(C):factor(S)	18	0.00	0.00	0.01	1.0000
log(I):log(T):factor(C):factor(S)	18	0.00	0.00	0.00	1.0000

Table 4, Table 5, Table 6, and Table 7 present ANOVA tables for *fft*, *sort*, *sparselu*, and *strassen* programs, respectively. The rows in the tables present the statistics about each factor (e.g. $log(I)$ for input size factor) or factor interactions (e.g. $log(I) : log(T)$ for the interaction between input size and number of threads).

Table 7. ANOVA table for *strassen*.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
log(I)	1	3039.29	3039.29	88243.22	0.0000
log(T)	1	168.43	168.43	2694.30	0.0000
factor(C)	2	2.34	1.17	505.28	0.0000
factor(S)	9	3.79	0.42	161.03	0.0000
log(I):log(T)	1	2.37	2.37	757.21	0.0000
log(I):factor(C)	2	1.14	0.57	183.07	0.0000
log(I):factor(S)	9	0.26	0.03	9.09	0.0000
log(T):factor(C)	2	0.31	0.15	49.42	0.0000
log(T):factor(S)	9	0.11	0.01	4.05	0.0000
factor(C):factor(S)	18	1.31	0.07	23.32	0.0000
log(I):log(T):factor(C)	2	0.13	0.07	20.96	0.0000
log(I):log(T):factor(S)	9	0.03	0.00	1.14	0.3341
log(T):factor(C):factor(S)	18	0.38	0.02	6.78	0.0000
log(I):factor(C):factor(S)	18	0.54	0.03	9.59	0.0000
log(I):log(T):factor(C):factor(S)	18	0.07	0.00	1.29	0.1818

The columns in the tables present different statistics, the important statistic for our analysis is *P-value* (Pr(>F)) that specifies if the factor on this row significantly influences the response variable (Refer to[12] for the details of the additional data given in tables). 0.0000 *P-value* (Pr(>F)) indicates that the factor on this row significantly influences the response variable.

For *fft*, ANOVA results indicate that all main factors affect the execution time of the application. Moreover, two-factor interactions including scheduler or cut-off policy, and three-factor interactions including both scheduler and cut-off policy contribute to the variance of the performance. As demonstrated in,[13] scheduler-related decisions play an important role on the performance of FFT computations and our results are compatible with this observation.

For *sort* and *strassen*, all main factors, two-factor interactions, and most of three-factor interactions have *zero* P values that contribute to performance variation.

For *sparselu*, all main factors and interactions, except the ones including cut-off policy, affect the program performance. As indicated in,[13] all schedulers perform similarly for different cut-off policies for this program.

5.2. *Results for Training Data*

As discussed in Section 3.4, we train different regression models based on ANOVA results. We apply each model for each program, and calculate prediction errors in terms of RMSE, MAE, and MAPE. Table 8 presents error values for six regression models.

Table 8. Prediction errors for different regression models.

		modeLog	**modelQuad**	**model2**	**modelAll**	**modelAov**	**modelAov2**
fft	RMSE	4.303	3.903	2.184	1.925	1.984	2.219
	MAE	2.618	2.441	1.328	1.224	1.245	1.350
	MAPE	0.167	0.156	**0.084**	**0.079**	0.079	**0.085**
sort	RMSE	0.021	0.020	0.013	0.011	0.011	0.013
	MAE	0.014	0.013	0.008	0.007	0.008	0.008
	MAPE	0.096	0.089	**0.056**	**0.049**	**0.051**	**0.056**
sparselu	RMSE	1.206	0.863	0.502	0.472	0.472	0.502
	MAE	0.522	0.342	0.318	0.319	0.319	0.318
	MAPE	0.049	0.038	**0.033**	**0.032**	**0.032**	**0.033**
strassen	RMSE	0.100	0.093	0.048	0.037	0.038	0.048
	MAE	0.051	0.048	0.024	0.018	0.018	0.024
	MAPE	0.065	0.062	**0.042**	**0.035**	**0.036**	**0.042**

The mean percentage error (*MAPE*) stays below 17% (maximum at *modeLog* model for *fft* program) for all models and programs. Moreover, for all four programs, models with factor interactions (*model2*, *modelAll*, *modelAov*, *modelAov2*) yield lower error rates, less than 10% for each case. Including all factors or including only factors from ANOVA analysis does not change error percentage significantly. Even in some programs, *MAPE* values are exactly the same for both models (*modelAll* and *modelAov*); 0.079 for *fft*, 0.032 for *sparselu* program.

After evaluating six regression models, we decide to apply the models *model2*, *modelAll*, *modelAov*, *modelAov2* for further analysis due to their lower error rates. Since the difference between these models is not significant, we do not want to eliminate any of them and evaluate their performance for validation data, which has configurations with larger input size.

5.3. *Results for Validation Data*

To validate the efficiency of regression-based approach on performance prediction across input sizes, we apply regression models on unseen input data. We predict execution times of programs for two additional input sets with varying configurations. In order to compare our predicted values, we also execute the programs with the same input sizes and the same configurations. We get five runs for each configuration to avoid the error between different runs, and use the average of five runs as the observed execution time. For this part of our evaluation, we have $2(input) \times 3(thread) \times 10(scheduler) \times 3(throttle) \times 5(replica) = 900$ runs for each program.

Figure 4 and Figure 5 show observed and predicted values (for four models) for different configurations of *fft* executions, with 134217728 and 268435456 floats, respectively. There are $3(thread) \times 10(scheduler) \times 3(throttle) = 90$ configurations for each case. Figure 6 also presents percentage error rates for different regression models. The prediction error for larger data is higher due to too many tasks created by the execution (Figure 9(b)). Since there are simultaneously executed parallel tasks in the system, the contention between those tasks has effect on execution time

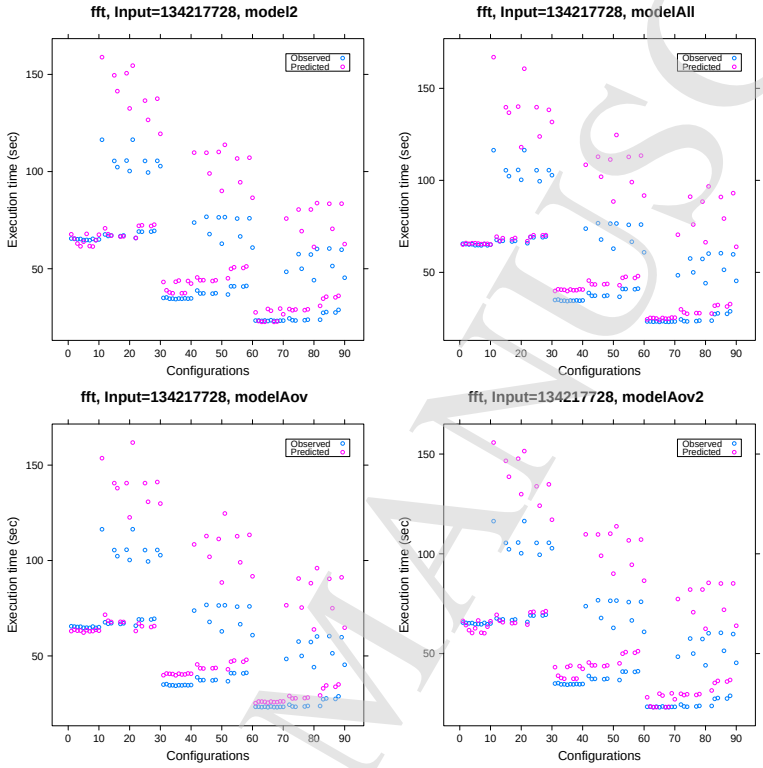


Fig. 4. Observed and predicted values for *fft*, Input size=134217728.

which we do not consider in our analysis. Those results lead us for future work that examines the contention factor in parallel program performance. The models that include all factor interactions (*modelAll*) and factor interactions decided by ANOVA analysis (*modelAov*) seem more successful. Both models fit data more accurately especially when the data exhibits more steady behavior (e.g., configurations between 1 and 10, configurations between 60 and 70, as shown in Figure 4 and Figure 5).

Similarly, Figure 7, Figure 8 and Figure 9 present results for *sort* program. In terms of error percentages, the performance of regression models do not differ significantly. However, *modelAll* exhibits lower rates for the largest input size (Figure 9(b)).

As discussed in Section 5.1, ANOVA analysis for *sparselu* yields the largest F values for input size (I) and number of threads (T) factors, which indicate the most important factors in performance deviation. We can see this behavior in *Observed* values on the Figure 10 and Figure 11. The configuration sets in the intervals 0-30, 30-60, and 60-90 differ by number of threads, and the performance deviations inside these intervals are not significant. With this predictable behavior, regression-based approach exhibits high prediction accuracy for *sparselu* program. As seen in Table

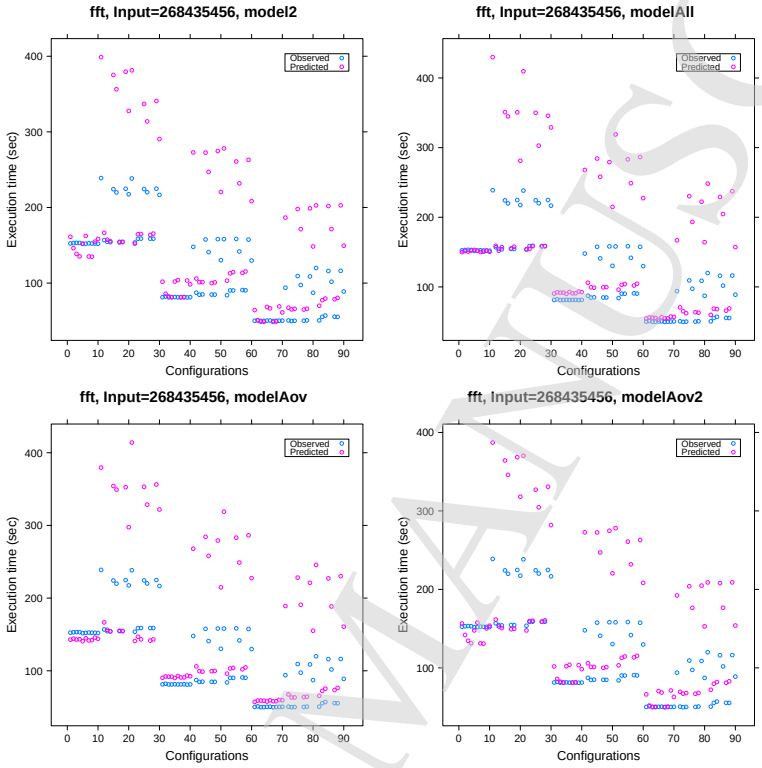
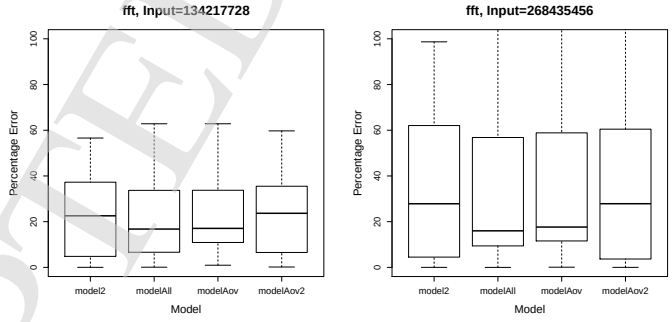


Fig. 5. Observed and predicted values for *fft*, Input size=268435456.



(a) fft, Input size=134217728 (b) fft, Input size=268435456

Fig. 6. Results for predicting *fft* validation input sets.

8, even error rate for simple model (i.e., *MAPE* for *modeLog*) is not very high (which is equal to 0.049) compared to other models.

For *strassen*, 2-factor interaction models (*model2* and *modelAov2*) perform

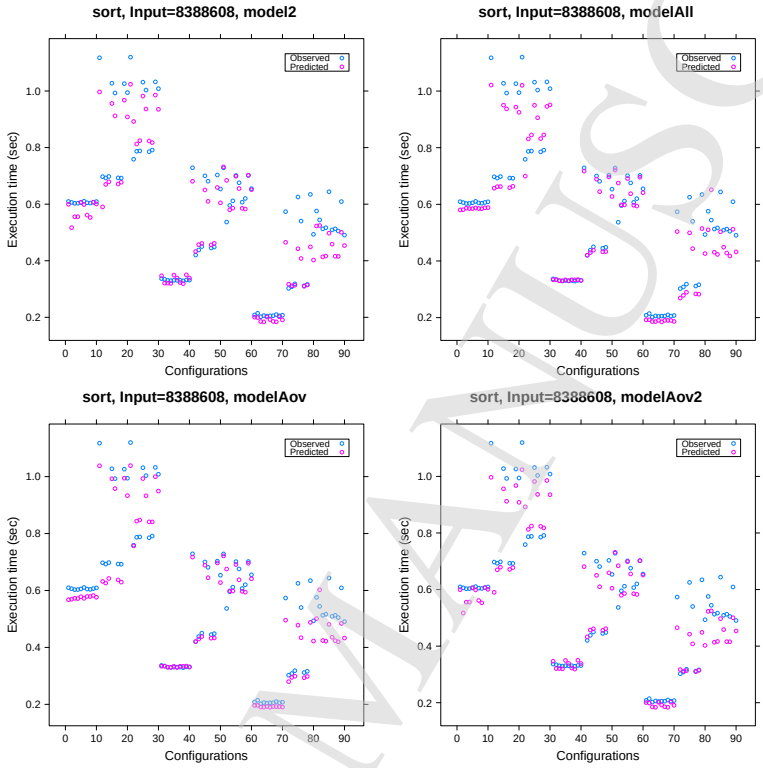


Fig. 7. Observed and predicted values for *sort*, Input size=8388608.

slightly better as in the training phase. While the difference is not clear in Figure 13 and Figure 14, Figure 15(b) exposes the difference with smaller median errors.

Figure 16 presents overall percentage error results for validation input sets. While error rates for training data stay below 10% for the selected models (see Table 8), it increases to 25% for validation data size of *fft* program. When we look at *fft* program characteristics, we can see that it forks many tasks (approximately 10M tasks for 128M input size), and the tasks compete for memory resources substantially.[10] Therefore, the cache effect might play an important role in performance of this program. Since we do not include this effect in our analysis, we get the largest error rate for *fft*. This investigation guides us for future work to work on memory-related factors.

On the other hand, for more easily predictable *sort* program, the error rate remains almost the same for validation data (7.3% at maximum) compared to training data error rate (5.6% at maximum). While we use ANOVA analysis to specify significant factor interactions, models derived without ANOVA analysis (*model2* and *modelAll*) perform similarly with models based on ANOVA analysis (*modelAov* and

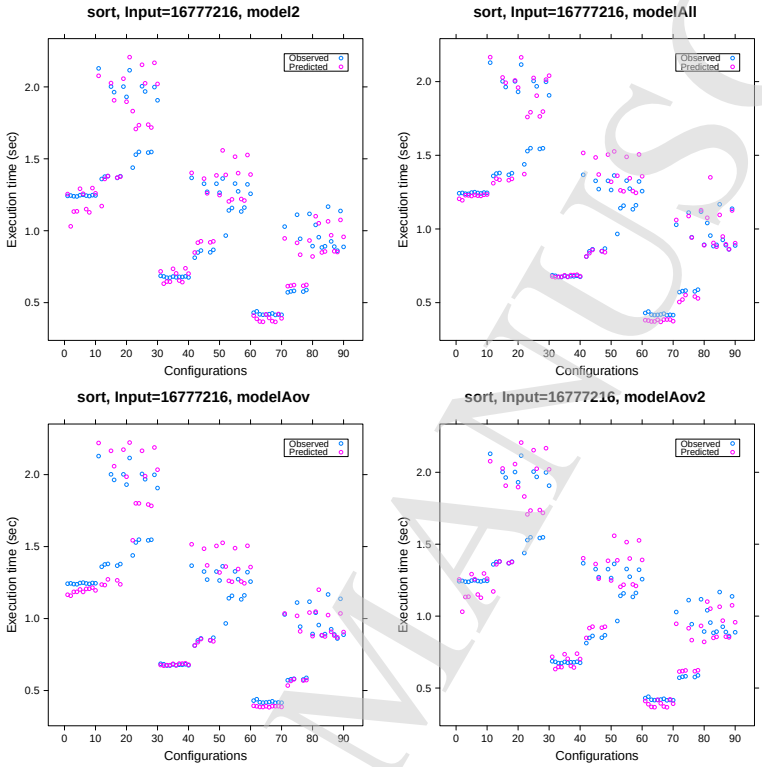
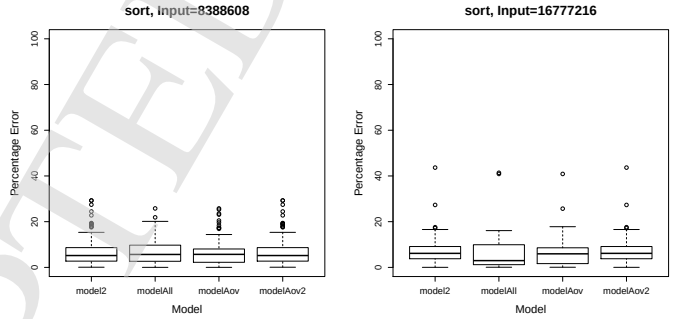


Fig. 8. Observed and predicted values for *sort*, Input size=16777216.



(a) sort, Input size=8388608 (b) sort, Input size=16777216

Fig. 9. Results for predicting *sort* validation input sets.

modelAov2). This demonstrates that additional factors (not coming from ANOVA analysis) do not disrupt prediction accuracy.

We also conduct a simulation-based comparison study for our target programs

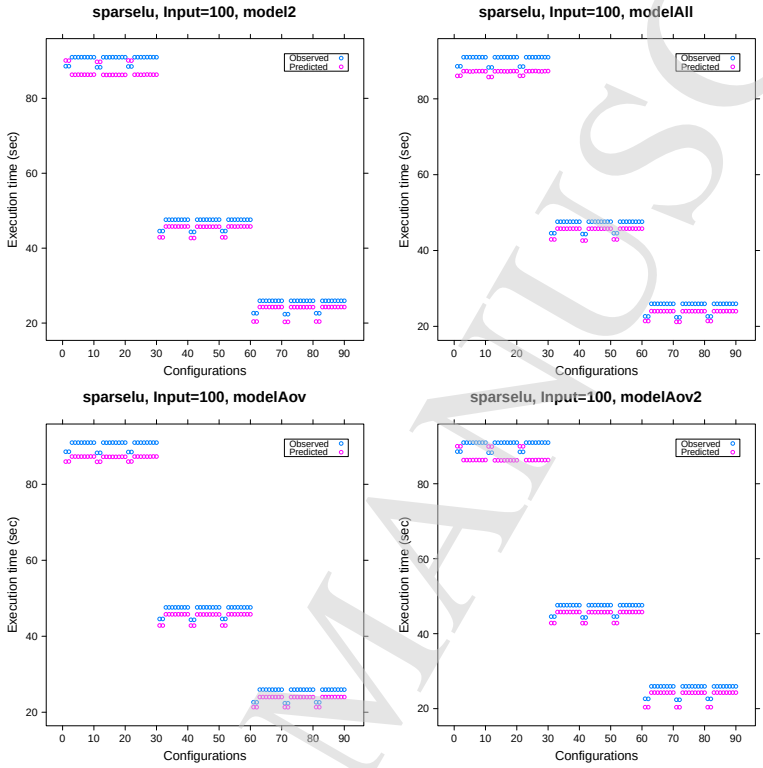


Fig. 10. Observed and predicted values for *sparselu*, Input size=100.

to compare with our regression-based approach. We simulate our applications on Sniper multi-core simulator[22]. To make the simulation configuration easier and to make the simulation time practical, we use a dual-core architecture (Nehalem-like) and include 2-thread execution configurations for our comparison study. We exclude *fft*, since Sniper simulation crashed during the execution for the validation input sets; and do not include *sparselu* due its impractical simulation times. We choose the best-performed (the lowest prediction error) model to represent our regression-based approach. Figure 17 presents the percentage error results of these configurations for our model and simulation execution. The results demonstrate that while our regression-based approach performs better than 10% error rates, the prediction error of the simulation-based approach can be increased to almost 20% for some cases. Although the simulation has lower error rate than our regression-based approach for *strassen*, with input size *4096*; since the error rates are not too high for this case, the simulation does not yield very helpful prediction (both rates (6% for regression, and 3% for simulation) are acceptable in general). For the other two applications, we even do not have any results due to simulation limitations. We also measure the time required for the simulation and our regression-based

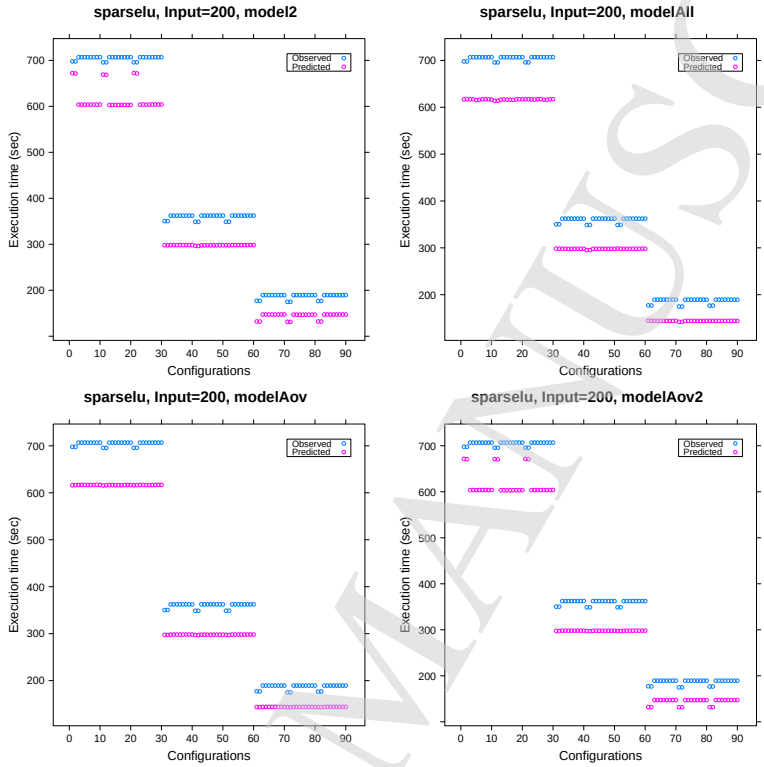
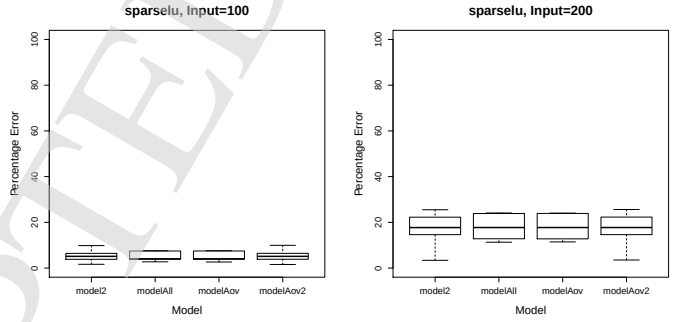


Fig. 11. Observed and predicted values for *sparselu*, Input size=200.



(a) sparselu, Input size=100 (b) sparselu, Input size=200

Fig. 12. Results for predicting *sparselu* validation input sets.

approach. We include the time for training data collection for regression-based approach (total time for 1350 executions), and the simulation time (total time for simulations) for simulation-based approach. Table 9 presents the elapsed time

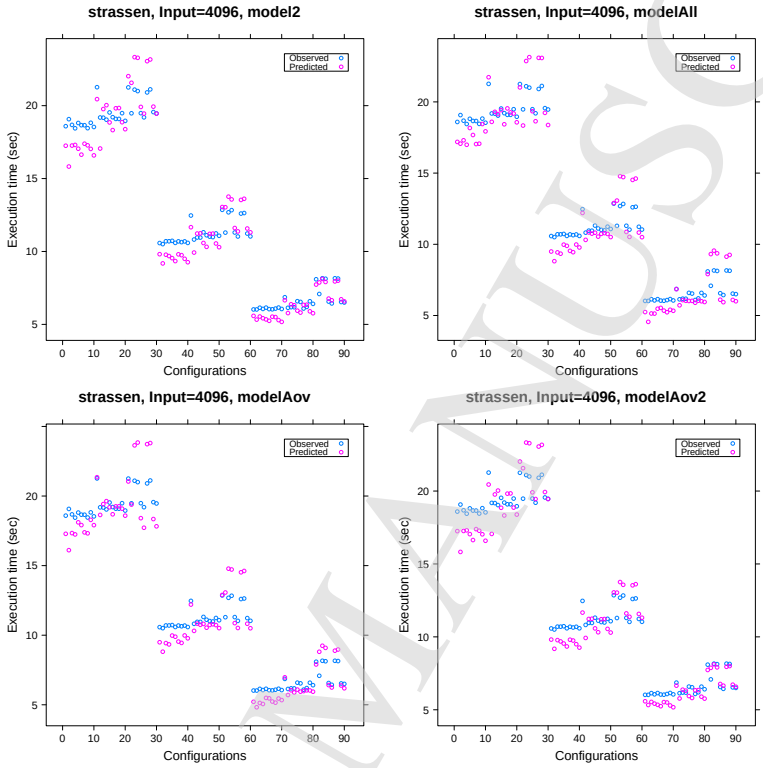


Fig. 13. Observed and predicted values for *strassen*, Input size=4096.

for each case, the simulation takes much longer time even training data collection requires more executions.

Table 9. Timing of regression-based and simulation-based prediction methods (in minutes).

	sort (I=8388608)	**sort (I=16777216)**	**strassen (I=4096)**	**strassen (I=8192)**
regression	3,319	3,319	16,761	16,761
simulation	184	376	333.5	1892

6. Related Work

In this section, we present the related work about performance analysis and prediction of parallel applications.

Measurement-based performance analysis: Duran et.al[13] present OpenMP task scheduling strategies and their impact on execution time by eval-

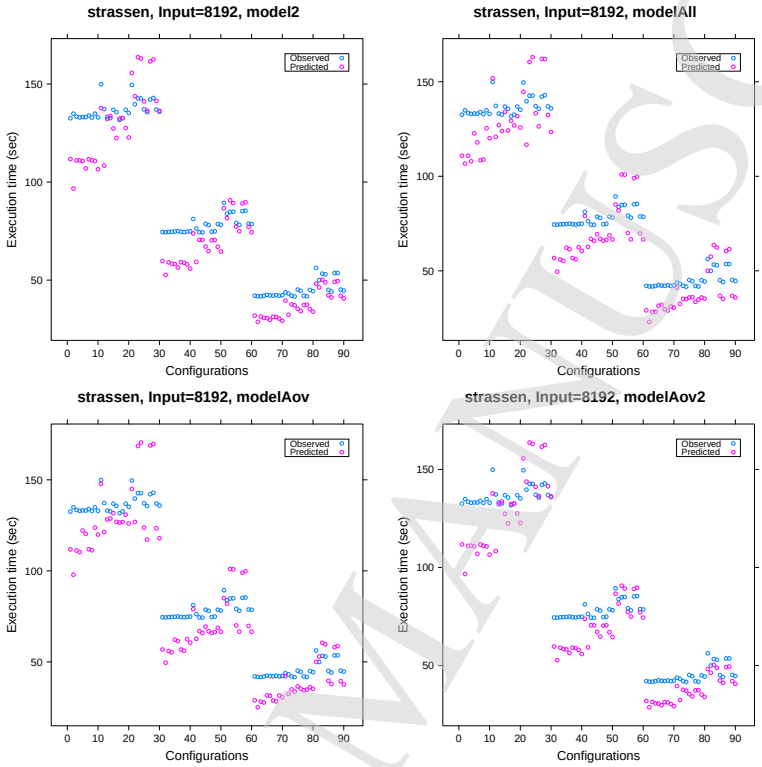
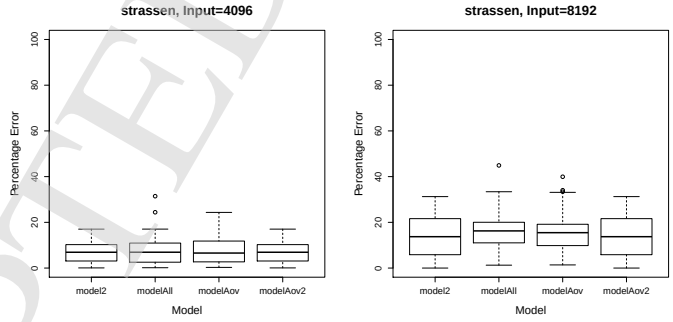


Fig. 14. Observed and predicted values for *strassen*, Input size=8192.



(a) strassen, Input size=4096 (b) strassen, Input size=8192

Fig. 15. Results for predicting *strassen* validation input sets.

uating combinations of different scheduling components with a set of applications. The effect of scheduling techniques and cut-off policies is examined by a detailed experimental study. While the study conducts several experiments to see the effect

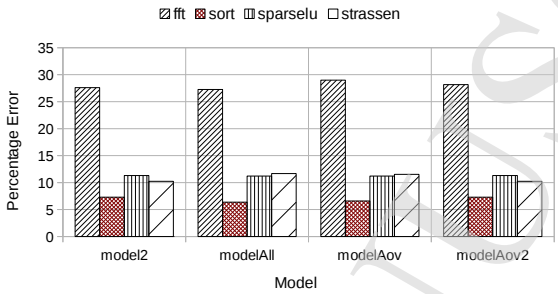


Fig. 16. Results for predicting validation input sets.

Fig. 17. Percentage error of regression-based and simulation-based predictions.

of schedulers, it does not have any statistics. Our work supports those evaluations by rigorous statistical analysis. Schmidl et.al[23] define a set of performance problems related to task parallelism and evaluate their impact on OpenMP applications by using an instrumentation based performance tool. Sibai et.al[24] propose a thread scheduling algorithm for a multicore system, and present a simulation study to compare the performance of the scheduling algorithm with various thread migration policies to other algorithms. These studies give an insight about the task-based program execution, and motivate the study for performance analysis of schedulers in task-based programs.

Simulation-based performance analysis: Rico et.al[11] present TaskSim which is a multi-core architecture simulator providing the simulation of task-based parallel applications on multi-cores using traces. They introduce different architecture model abstractions by excluding timing of operations, and compare the

speed and accuracy of the performance models based on these abstractions. The study presents scalability results for OmpSs programs collected via TaskSim, and demonstrates that the accuracy of the simulation results increases as more architectural properties are included in the simulation. Grass et.al[25] presents a sampled simulation technique for dynamically scheduled task-based parallel programs. They simulate only a fraction of all task instances in order to reduce the simulation time, and observe tolerable accuracy results with faster simulations. The simulation-based performance prediction requires detailed runtime system implementation of task-based programs, and high simulation times for large input may make the work impractical. Our regression-based approach requires only the result of a subset of executions, trains prediction models, and predicts the execution times based on those models. However, the simulation may help our training data collection phase if the target architecture is not available in the earlier design times.

Analytical performance models: Xu et.al[26] propose a shared cache-aware analytical performance model for multicore processors. The model predicts effective cache size for concurrently running processes by considering both cache miss rate and performance degradation as functions of process effective cache size. Tudor et.al[27] also propose an analytical model to analyze speedup of shared-memory programs on multicore systems. The model determines the speedup and speedup loss by using data dependency and memory overhead for various configurations of threads, cores and memory access policies. Wu et.al[28] present a model to predict the parallel execution time for parallel tasks. The prediction approach uses a transformation to derive a normal distribution for the parallel execution time, and utilizes normal features to derive maximum parallel execution time distribution. Shudler et.al[29] propose an empirical method for finding the isoefficiency function of a task-based program, by considering efficiency, input size, and core count. They combine performance modeling with benchmarking, and construct efficiency models with smaller error rates. Ryabko et.al[30] introduce computer capacity metric by providing equations to estimate the performance of a processor. We use a statistical performance model instead of an analytical approach for performance prediction due to the complex runtime options of task-based programs.

Machine-learning based performance prediction: Barnes et.al[7] use regression models to predict parallel program scalability, and achieve small prediction errors for parallel applications. Wang et.al[31] also propose prediction methods to predict the number of threads and the scheduling policy for OpenMP programs. Our performance model deals with task-based OpenMP programs, and focuses on scheduler-related parameters to predict the performance for large data sets. Ipek et.al[5] propose a performance model trained by multilayer neural networks for performance prediction on large-scale parallel platforms. The evaluation adjusts model parameters by training data, and then assesses the performance of the model by prediction error on validation data. Lee et.al[32] derive and validate regression models for performance and power prediction. The experimental evaluation indicates that statistical modeling provides accurate prediction results, and regression can be

used for design decisions instead of exhaustive microarchitectural design space exploration. Lee et.al[6] compare polynomial regression and artificial neural networks for performance prediction of parallel applications. While the accuracy of these techniques is comparable, regression offers more statistical understanding and neural networks offer more usability. Nadeem et.al[33] describe a neural network based model to predict workflow execution time in the grid environment. They model the workflow execution by considering workflow structure and execution runtime attributes, and evaluate their model for real-world applications. With higher prediction rates, the study guides the different optimization strategies such as scheduling policy for grid environments. Although the target architecture of this work is different, it supports the idea of the machine-learning based approach as the runtime behavior prediction.

In this work, we evaluate regression-based modeling to predict execution time of task-based programs across input sizes. We train a set of regression models including different factors and factor interactions, and assess the prediction accuracy for a set of validation data. To the best of our knowledge, this paper is the first work that evaluates task-based OpenMP program performance by considering scheduling parameters, and proposes a statistical model to predict the performance for large data sets.

7. Conclusions

In this paper, we present a regression-based approach for task-based program performance prediction. Our approach uses executions with a set of input data by varying runtime parameters, to predict performance for a larger input data size. Our experimental evaluation shows that we can derive regression models based on training data and predict performance with mean error rate as low as 6.3%, and 14% in average among four task-based programs. Our prediction methodology offers accurate results to guide runtime decisions for high performance.

Acknowledgements

The research leading to these results has received funding from the ARTEMIS joint under-taking under grant agreement number 295440 under Article 171 of the treaty and ERCIM Alain Bensoussan Fellowship Program.

References

1. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP)*, 1995.
2. TBB. Thread building block. http://www.threadingbuildingblocks.org, 2014.
3. Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of

openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.
4. GCD. Grand central dispatch reference. http://libdispatch.macosforge.org, 2014.
5. Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An approach to performance prediction for parallel applications. In *International Euro-Par conference on Parallel Processing*, 2005.
6. Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Symposium on Principles and practice of parallel programming (PPoPP)*, 2007.
7. Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *International conference on Supercomputing (ICS)*, 2008.
8. Arsalan Shahid, Muhammad Yasir Qadri, Martin Fleury, Hira Waris, and Ayaz Ahmad. Ac-dse: Approximate computing for the design space exploration of reconfigurable mpsocs. *Journal of Circuits, Systems and Computers*, 2018.
9. Tomas Berling and Per Runeson. Efficient evaluation of multifactor dependent system performance using fractional factorial design. *IEEE Transactions on Software Engineering*, 29(9):769–781, 2003.
10. Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *International Conference on Parallel Processing (ICPP)*, 2009.
11. Alejandro Rico, Felipe Cabarcas, Carlos Villavieja, Milan Pavlovic, Augusto Vega, Yoav Etsion, Alex Ramirez, and Mateo Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4), 2012.
12. Douglas C. Montgomery. *Design and Analysis of Experiments*. Wiley, 2009.
13. Alejandro Duran, Julita Corbalan, and Eduard Ayguade. Evaluation of openmp task scheduling strategies. In *International conference on OpenMP in a new era of parallelism (IWOMP)*, 2008.
14. Nanos++. Nanos++ runtime system. http://pm.bsc.es/nanox, 2014.
15. Alejandro Duran, Julita Corbalan, and Eduard Ayguade. An adaptive cut-off for task parallelism. In *ACM/IEEE conference on Supercomputing (SC)*, 2008.
16. Bjorn B. Brandenburg, John M. Calandrino, and James H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Real-Time Systems Symposium (RTSS)*, 2008.
17. Shoaib Akram, Manolis Marazakis, and Angelos Bilas. Understanding scalability and performance requirements of i/o-intensive applications on future multicore servers. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2012.
18. Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.
19. J. M. Bland and D. G. Altman. Transformations, means, and confidence intervals. *BMJ*, (312):1079, 1996.
20. OpenMP. Openmp application program interface, v.3.0. http://www.openmp.org/mp-documents/spec30.pdf, 2008.
21. R. R language. http://www.r-project.org, 2014.
22. Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout.

An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(28), 2014.

23. Dirk Schmidl, Peter Philippen, Daniel Lorenz, Christian Rssel, Markus Geimer, Dieter an Mey, Bernd Mohr, and Felix Wolf. Performance analysis techniques for task-based openmp applications. In *International Workshop on OpenMP (IWOMP)*, 2012.
24. Fadi N. Sibai. Simulation and performance analysis of multi-core thread scheduling and migration algorithms. In *International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, 2010.
25. Thomas Grass, Alejandro Rico, Marc Casas, Miquel Moreto, and Eduard Ayguade. Taskpoint: Sampled simulation of task-based programs. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.
26. Chi Xu, Xi Chen, Robert P. Dick, and Zhuoqing Morley Mao. Cache contention and application performance prediction for multi-core systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.
27. Bogdan Marius Tudor and Yong Meng Teo. A practical approach for performance analysis of shared-memory programs. In *IEEE International Parallel and Distributed Processing Symposium*, 2011.
28. Rongteng Wu, Jizhou Sun, and Jinyan Chen. Parallel execution time prediction of the multitask parallel programs. *Performance Evaluation*, 65(10):701–713, 2008.
29. Sergei Shudler, Alexandru Calotoiu, Torsten Hoefler, and Felix Wolf. Isoefficiency in practice: Configuring and understanding the performance of task-based applications. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.
30. Boris Ryabko and Anton Rakitskiy. An analytic method for estimating the computation capacity of computing devices. *Journal of Circuits, Systems and Computers*, 26(5), 2017.
31. Zheng Wang and Michael F.P. OBoyle. Mapping parallelism to multi-cores: A machine learning based approach. In *Symposium on Principles and practice of parallel programming (PPoPP)*, 2009.
32. Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *International conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2007.
33. Farrukh Nadeem, Daniyal Alghazzawi, Abdulfattah Mashat, Khalid Fakeeh, Abdullah Almalaise, and Hani Hagras. Modeling and predicting execution time of scientific workflows in the grid using radial basis function neural network. *Cluster Computing*, 20(3):2805–2819, 2017.