Performance Prediction for Multi-threaded Applications

Nitish Agarwal *
Computer Science Department
New York University
New York, NY, USA
nitish.agarwal@nyu.edu

Tulsi Jain *
Computer Science Department
New York University
New York, NY, USA
tulsi.jain@nyu.edu

Mohamed Zahran
Computer Science Department
New York University
New York, NY, USA
mzahran@cs.nyu.edu

***Abstract*—With multicore processors becoming the norm, parallel programming is routinely being employed in order to gain an efficient use of hardware resources. However, with increased parallelism, performance gain is not always guaranteed as the gains achieved are susceptible to be overwhelmed by the overheads of creation of and communication among processes/threads. In addition, given hardware with limited number of cores, increased parallelism can result in competition among software threads and the added overhead of preemption and scheduling may be severe. Therefore, application performance prediction has become an important topic in parallel programming research.**

Though estimation of application performances with varying levels of parallelism can be achieved by executing the application multiple times with different degrees of parallelism on the target machine, it is often not feasible in the interest of time. In this paper, we propose an efficient learning based approach to estimate application performance with varying degrees of parallelism for a specific hardware. We evaluate various machine learning models with respect to how accurately they estimate the speed-up gains for varying levels of multi-threading relative to single threaded execution. We evaluate our predictions on two publicly available benchmarks suits PARSEC 3.0 and SPLASH 3.0 for applications parallelized using p-threads in C. Gaussian Process Regression performs the best, achieving correlation coefficients of estimated speed-ups vs actual speed-ups ranging from 0.67 to 0.82 for different number of threads.

***Index Terms*—multi-core processors, parallel programming, program performance prediction, machine learning,**

I. Introduction

Improvement in computation speeds over the past couple of decades cannot be ascribed to faster transistors, as had been the norm in the previous five decades afforded by Dennard Scaling. Dennard scaling appears to have broken down since around fifteen years and transistors cannot be made much faster without making the processors consume gigantic amounts of energy. [1] However, with the rise of multi-core processors, tasks considered to be computationally intractable two decades ago are becoming quite feasible thanks to massive clusters of computational cores working together to solve a problem. Consequently, since the past few years, multi-core processors have found a ubiquitous usage in all kinds of computer systems ranging from personal computers to supercomputers. Although parallel programming on multi-core processors sounds promising, in order to exploit the maximum computation out of multi-core processors in a given amount of time, it is imperative to identify the optimal level of parallelism that should be employed for a given task, performing the best for the available multi-core architecture. Being able to predict the performance of an application with varying levels of parallelization (various numbers of software threads) without physically executing the application with all the different number of threads is thus a necessary component to enable such efficient use of the Central Processing Unit (CPU). However, due to the increasing complexity of the system architecture, modeling and predicting the performance of programs is a challenging task.

Traditional analytical performance models in closed-form expressions or functions of hardware characteristics and programs features are efficient to compute. However, they are typically not accurate enough or still require at least a partial execution model (such as a functional instruction set simulators) of the target processor to collect target-specific traces or execution statistics, such as instruction counts, memory traces, or branch statistics which can still be slow [2]. Consequently, we propose a learning based approach to estimate application performance with varying degrees of parallelism for a specific hardware. We achieve this by extracting application features characterizing the ease of parallelizability and predict speedups for various number of threads with respect to a single thread. These features are extracted after executing the application of interest on the target hardware for single thread only. Based on features extracted from this execution, run-time and hence, speed-ups are estimated for various multi-threaded executions of the application on the target hardware. This enables the programmer(s) of a multi-threaded application to identify the optimal degree of parallelizability (the number of software threads) that should be employed after executing only a single threaded version of the same. Clearly, this is not helpful at all for applications intended to be executed merely once or twice in their lifetimes since the application needs to be executed once on a single thread in order to extract features for identifying the optimal number of threads. However, identifying the optimal number of threads can be a boon for applications intended to be run multiple times.

Our main contributions are the following:

- We propose a framework to predict program perfor-

* equal contribution

mance for parallel applications parallelized using the *p-threads* library in C using low level feature set obtained from single threaded execution and evaluate our approach for multi-threaded applications in SPLASH-3.0 and PARSEC-3.0.
- We compare multiple machine learning algorithms on multiple error metrics and find the best performing model for performance prediction.
- We analyze low-level features and find the features that have a more active role in predicting speed-ups.

The remainder of the paper is organized as follows: Section II surveys related work in this area. Section III discusses the proposed idea for program performance prediction using machine learning algorithms and Section IV discusses the experiment setup in brief, tools used to extract features in order to construct the training and testing datasets. This is followed by evaluation and discussion of our experiments, empirical results in Section V. Finally, Section VI concludes with a conclusion of this work.

II. Related Work

The key element in any application performance prediction model is the set of features extracted from the parallel applications. Several researchers have proposed and endorsed the use of branch predictability as an important feature characterizing the ease of parallelizability for a multi-threaded application. Cache hit rates for various cache levels and instruction level statistics (number of floating point operations, loops, etc) have also been used as inputs to their models by Zheng et. al. [2] and Nemirovsky et. al. [3].

There has been much work on performance prediction for multi-threaded applications albeit with different use cases in the recent years. These use cases lie in two major classes: *(A)* cross-platform performance prediction, and *(B)* performance prediction based on micro-architecture independent features. Works in class *A* make use of the execution statistics of the applications when executed on a given hardware to estimate performance on other hardware configurations. On the other hand, works in class *B*, most notably by Hoste et. al. [4] attempt to predict performance on a given hardware solely on the basis of hardware independent program features along with the processor architecture without ever executing the applications. Micro-architecture program features have been extracted from instrumentation tools such as ATOM in [5] and Valgrind in [6].

We consider a different problem where we aim to predict speed-ups with respect to single threaded execution as a measure of performance on a single microprocessor configuration for various numbers of threads using execution statistics obtained from single-threaded execution the same application on the given hardware.

III. Approach

Our model takes a set of program features (extracted from instrumentation and profiling tools along with single-threaded execution statistics) as input and outputs the estimated speed-up for an application of interest on the required number of threads. Figure 1 illustrates this performance prediction framework for speed-up estimation for 4 threads. The models are trained in a supervised fashion on 17 workloads obtained from SPLASH 3.0 and PARSEC 3.0 using actual speedups obtained from execution on the hardware for the required number of threads. Each of the 17 workloads has been executed for 4 different problem sizes providing us 68 applications in total.

We now go through the two components of this framework, namely Feature Extraction and models to predict speed-up for various numbers of threads.

A. Feature Extraction

Using profiling tools such as *Perf* and *Valgrind* along with single-threaded execution statistics, we extracted the following features from each of the applications.

Number of Cycles, IPC, Number of Branches, Branch Predictability, Cache References (L1 and last level cache references separately), and Cache Miss Rates (L1 and last level cache miss rates separately).

The intuition behind the selection of these features among all the other statistics produced by the instrumentation and profiling tools is that we expect that these features combine both program characteristics and hardware features without the need to include a large number of microarchitecture features. Also, they are good indication of the ease or difficulty of parallelizability.

Number of cycles is characteristic of the application size. Larger applications (workloads given larger problem sizes as input) are expected to benefit more from increased parallelization whereas smaller applications may not benefit as much from parallelization as compared to the overheads associated with thread creation, deletion, communication, etc.

Branch Predictability characterizes how well branches can be predicted before actually evaluating them. High branch predictability is expected to lead to better speedups.

Cache Misses reduce the performance of parallel programs by introducing overheads due to cache coherence.

Later, we discuss that speed-up estimations were observed to be most heavily characterized the number of cycles, branches and branch miss-predictions.

B. Models

Given the feature set obtained from instrumentation and single-threaded execution along with speedups for various multi-threaded implementations for applications in the benchmark suite, our objective is to interpolate a function between them, such that for features obtained from any new application, that function can predict the speedups for various levels of multi-threaded executions. Here, we discuss the two best performing models among the ones we investigated to parameterize such a function.

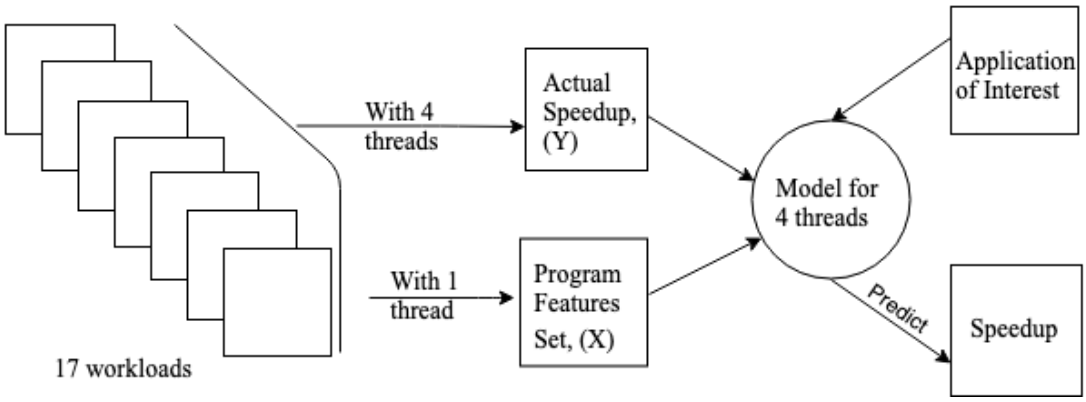

Fig. 1. Training and Speedup Estimation Framework for 4 Threads

1) Linear Regression: Linear Regression aims to find a linear function of the feature vector that minimizes the sum of squared differences between the outputs of the function given an input and the expected value. In our framework, the feature vector is obtained from single-threaded execution and instrumentation where as the expected value is the speedup observed. Although applicable, we used lasso (L1) and ridge (L2) regularization respectively instead of ordinary linear regression since it is susceptible to overfitting.

2) Gaussian Process Regression: Gaussian Process Regression finds a distribution over the possible functions $f(x)$ that are consistent with the observed data. It is fully specified by its mean function $m(x)$ and its covariance function $k(x, x')$. This is a natural generalization of the Gaussian distribution whose mean and covariance is a vector and matrix, respectively. [7]

Let $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$ be a training set of i.i.d. examples from some unknown distribution. In the Gaussian process regression model,

$$y^{(i)} = f(x^{(i)}) + \epsilon^{(i)}, i = 1, ..., m \quad (1)$$

where $f(x)$ is characterized by $m(x)$ and $k(x, x')$ and $\epsilon^{(i)}$ are i.i.d. noise variables with mean zero.

In our case, $x^{(i)}$ is the program feature vector and $y^{(i)}$ is speedup for the i^{th} application in the benchmark suites. A prior distribution is assumed over functions $f()$. Given a set of i.i.d. validation points drawn from the same unknown distribution as S, the posterior predictive distribution is computed over the validation outputs.

IV. Experimental Setup

A. Benchmarks

We evaluate our approach on two well-known benchmark suits - PARSEC 3.0 [8] and SPLASH 3.0 [9]. Both the benchmarks suites comprise of multi-threaded workloads for a wide variety of complex applications like computer graphics, financial modelling, and numerical analysis. Applications in the SPLASH-3.0 have been parallelized using p-threads in C whereas PARSEC-3.0 comprises of applications parallelized using OpenMP and p-threads in C. However, owing to differences in the parallelization paradigms employed by OpenMP and p-threads, we have carefully selected applications employing only one of these libraries, i.e. p-threads. We finalized a total of 17 workloads, 10 from PARSEC and 7 from SPLASH benchmark suits where all the applications employ p-threads for parallelization. Table I contains all the 17 workloads selected by us where each workload was executed with four different input sizes.

TABLE I
Parallel Applications Used for Training and Evaluation

Parallel Application	Benchmark Suite	Application Domain
Black-Scholes	Parsec-3.0	Financial Analysis
Body Track	Parsec-3.0	Computer Vision
Canneal	Parsec-3.0	Engineering
Faceism	Parsec-3.0	Animation
Ferret	Parsec-3.0	Similarity Search
Fluid Animate	Parsec-3.0	Animation
Stream Cluster	Parsec-3.0	Data Mining
Swaptions	Parsec-3.0	Financial Analysis
VIPS	Parsec-3.0	Media Processing
X264	Parsec-3.0	Media Processing
Radix Sort	Splash-3.0	Non-Comparative Sorting
Ocean Contiguous	Splash-3.0	High-Perf. Computing
Ocean Non-Contiguous	Splash-3.0	High-Perf. Computing
LU Contiguous	Splash-3.0	High-Perf. Computing
LU Non-Contiguous	Splash-3.0	High-Perf. Computing
FFT	Splash-3.0	Signal Processing
Radiosity	Splash-3.0	Graphics

B. Tools

Various tools have been and are used by researchers for extracting program specific features. We generate execution statistics of applications with *perf* [10], a performance analysis tools for Linux while executing them on a real linux machine rather than using cycle accurate simulation. In addition, we use *Valgrind* [11], an instrumentation framework for building dynamic analysis tools, in order to extract program memory and cache access patterns for analyzing multi-threaded speedups.

C. Data-set preparation

Obtaining and/or preparing a large, diverse, and useful dataset is one of the most crucial aspects in machine learning

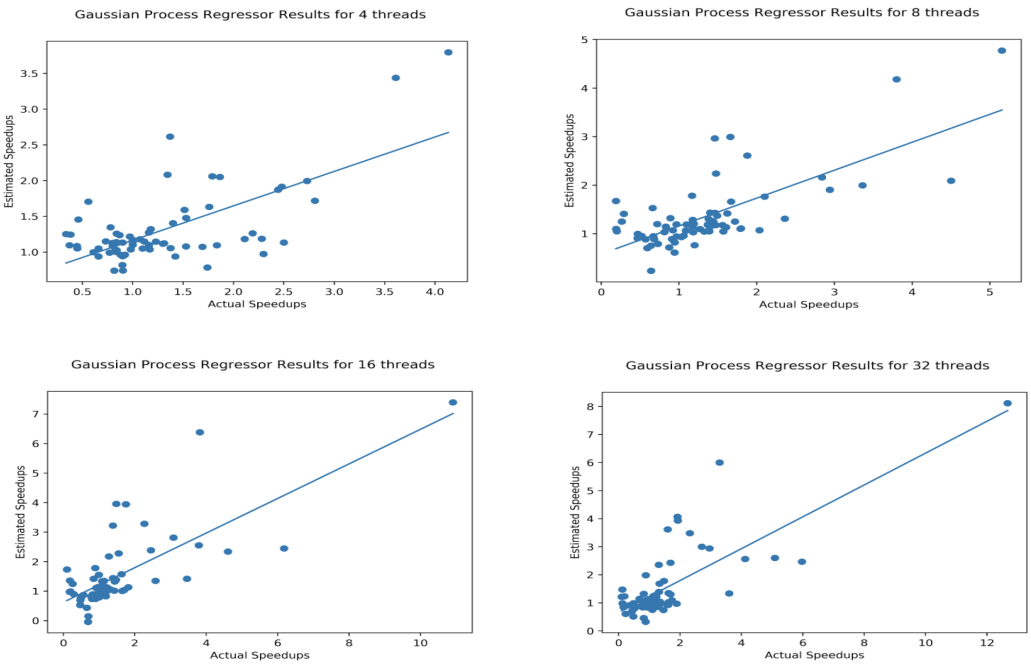

Fig. 2. Predicted speedups v/s actual speedups

applications. Most of our efforts were spent on building a reliable feature set capable of estimating the desired performance metrics. Each workload was executed with 4 different input sizes on 1, 4, 8 16, and 32 threads separately with valgrind and perf on a real machine. We noted down the required program behaviour parameters described in the previous section averaged over five executions.

The features we use to estimate speedups, such as instruction count (ranging from millions to billions) and L1 cache miss rate (always less than 1) vary greatly in magnitudes. Gradient-based optimization algorithms have a very hard time to move the weight vectors towards a good solution when the features vary wildly in magnitudes. However, when the features are normalized, the cost surface is less elongated and gradient-based optimization methods perform much better. Consequently, we normalized all the features such that they have a mean of 0 and variance of 1.

D. Training and Prediction Setup

Figure 1 illustrates the performance prediction framework for speedup estimations with 4 threads. Program features are extracted from single-threaded execution and actual speedups (the target variables) are computed by dividing single-threaded execution time with execution time on the desired number of threads. The models are then trained in a supervised fashion on 17 workloads obtained from SPLASH 3.0 and PARSEC 3.0 using the extracted features and actual speedups. Each of the 17 workloads is executed for 4 different problem sizes providing us 68 applications in total. We train six models, namely Linear Regression, K-Nearest Neighbors, Support Vector Regression, Random Forest, Decision Trees and Gaussian Process Regression each with 1 hold out validation. Different models are trained for different numbers of threads, i.e. we have a separate model for 4, 8, 16, and 32 threads. We evaluate all these algorithms on multiple metrics and observe that Gaussian Process Regression performs the best for all evaluation metrics. We make use of *sklearn*, a python based machine learning library, to implement all the above mentioned algorithms.

The trained model is then used to estimate the speedup of an unobserved application using features obtained from the execution of the application on a single thread. Since different models are trained for different numbers of threads, in order to estimate the speedup for an unknown application for a desired number of threads, the corresponding model is used. The final prognosis is to execute an unobserved application on the number of threads for which the estimated speedup is the maximum (possibly just on a single thread if the estimated speedups for all numbers of threads are less than 1).

V. Evaluation

In order to validate our confidence in the models discussed in the previous sections, we begin by plotting the actual speedups versus the estimated speedups for applications in our benchmark suites as illustrated in figure 2. The top-left, top-right, bottom-left, and bottom-right graphs represent the plots for 4, 8, 16, and 32 threads respectively. In all the graphs, the horizontal axis represents the actual speedups while the vertical axis represents the estimated speedups, which are predicted using the Gaussian Process Regression model. Each dot in these scatter plots represents one workload with an input type; there are 17 workloads with four different input types,

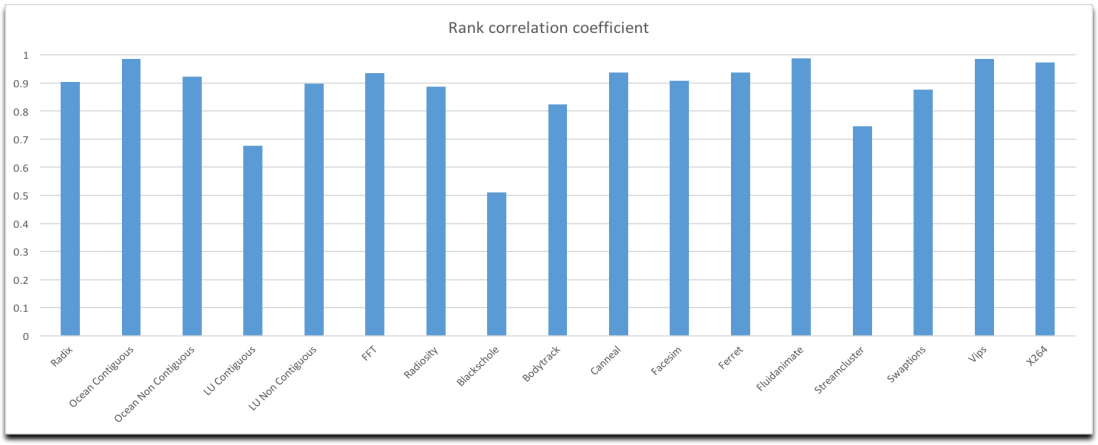

Fig. 3. R co-relation coefficient for each workload

thus a total of 68 points. Different models are trained on those 68 applications with one hold-out validation, i.e. each model is trained on 67 of the 68 (17×4) applications and validated on the application excluded from the training set. This process of validation is repeated for all the 68 applications. As can be observed, points are scattered near a straight line with y intercept zero and slope one. One key observation which is interesting to note is that the model is able to accurately predict speedup for an outlier (workload swaptions in the PARSEC benchmark for the largest input size) which alone has double digits speedups for 16 and 32 threads among all the other workloads and input types.

For most of the applications, we observed that actual as well as estimated speedups improved unanimously for all numbers of threads with increasing problem sizes. This is to be expected since larger applications are able to justify the overheads associated with thread creation, communication, etc with the gains achieved by parallelization whereas these gains are dwarfed by overheads for smaller applications.

Spearman's rank correlation coefficient, a measure for how well the estimated rank corresponds to the actual rank, has been used to quantify the accuracy of predicting ranks of machines based on predicted speedups. [12] To have more insight about model performance, we have evaluated our algorithms on two main errors metrics used to evaluate regression based models i.e. R correlation coefficient and Mean absolute error.

Table II shows R correlation coefficients, a measure of strength of the straight-line or linear relationship between two variables, between the actual and predicted speedups for various number of threads for the six machine learning algorithms we have considered, as discussed in section IV. As is evident from the table, linear regression and Gaussian Process regression outperform the other 4 models investigated by us. In particular, Gaussian Process Regression, resulting in a correlation coefficient of 0.79 and 0.82 for 16 and 32 threads respectively, is able to predict the speedups for higher number of threads exceedingly well. The reason behind more accurate predictions for higher number of threads for most of the models can be ascribed to higher reliance of features characteristic of the ease of parallelizability for a higher level of parallelization (the use of more threads).

TABLE II
Correlation Coefficients between Predicted and Actual Speedups

Algorithms	4 Thread	8 Thread	16 Thread	32 Thread
Linear Regress.	**0.67**	**0.72**	0.74	0.77
KNN	0.46	0.57	0.49	0.50
SVR	0.43	0.46	0.44	0.41
Random Forest	0.43	0.53	0.53	0.56
Decision Tree	0.56	0.51	0.52	0.51
Gauss. Process	**0.67**	0.69	**0.79**	**0.82**

In figure 3, we show R-correlation coefficients for each workload when trained using Gaussian Processes Regression. For few workloads such as *Ocean*, *Fluidanimate*, *Vips*, and *X264*, R-coefficient is close to 1 implying strong linear relationship between actual and predicted speedups. In general, overall performs of gaussian process is very convincing.

Table III shows mean absolute error between the actual and predicted speedups for various number of threads for all the six machine learning algorithms. Again, linear regression and Gaussian Process regression outperform the other 4 models. We also observed similar trends for adjusted r squared error showing robustness of our evaluation across multiple evaluation metrics.

Finally, we observe that speedup estimations are most heavily characterized by the number of cycles, branches,

TABLE III
Mean Absolute Error between Predicted and Actual Speedups

Algorithms	4 Thread	8 Thread	16 Thread	32 Thread
Linear Regress.	0.41	**0.46**	0.63	0.69
KNN	0.49	0.54	0.71	0.78
SVR	0.46	0.52	0.64	0.72
Random Forest	0.5	0.53	0.72	0.77
Decision Tree	0.57	0.7	1.02	0.83
Gauss. Process	**0.43**	0.51	**0.62**	**0.66**

and branch mispredictions. This is expected since the size of the application, and hence the amount of workload to be parallelized, is characterized strongly by the number of cycles and high branch misprediction rate severely degrades performance parallel applications.

VI. Conclusion and Future Work

This paper demonstrates the use of learning based approaches to predict parallel speedup compared to single threaded execution. We employed machine learning algorithms capable of learning relationships between low-level features like cycle counts, instruction counts, branch predictability, cache performance, etc. to interpolate a function that outputs the speedup. We achieved convincing results using Gaussian Process regression and linear regression. Next, we analyzed features exhibiting high dependence to speedup prediction and observe that few features are more active than others to predict speedup. Number of cycles, branch counts, and branch predictability are the key factors behind speedup prediction.

In the future, this approach can be extended to other parallel paradigms i.e. OpenMP, MPI, etc. Also, recent success of neural network can be leveraged in performance prediction of parallel applications. However, being data hungry, neural network tend to over-fit without large amounts of data. So obtaining a diverse and bigger dataset would be an important step to achieve even better results.

In addition, we also see the possibility of exploring fast instrumentation tools capable of approximating program statistics accurately without requiring program execution, thus allowing for much faster speedup estimations.

References

[1] M. Bohr, "A 30 year retrospective on dennard's mosfet scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[2] X. Zheng, P. Ravikumar, L. K. John, and A. Gerstlauer, "Learning-based analytical cross-platform performance prediction," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 52–59, IEEE, 2015.

[3] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, and A. Cristal, "A machine learning approach for performance prediction and scheduling on heterogeneous cpus," in *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 121–128, IEEE, 2017.

[4] K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," *IEEE micro*, vol. 27, no. 3, pp. 63–72, 2007.

[5] K. Hoste and L. Eeckhout, "Comparing benchmarks using key microarchitecture-independent characteristics," in *2006 IEEE International Symposium on Workload Characterization*, pp. 83–92, IEEE, 2006.

[6] T. Hoshi, K. Ootsu, T. Ohkawa, and T. Yokota, "Runtime overhead reduction in automated parallel processing system using valgrind," in *2013 First International Symposium on Computing and Networking*, pp. 572–576, IEEE, 2013.

[7] C. E. Rasmussen, "Gaussian processes in machine learning," in *Summer School on Machine Learning*, pp. 63–71, Springer, 2003.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 72–81, ACM, 2008.

[9] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 101–111, IEEE, 2016.

[10] A. C. De Melo, "The new linuxperftools," in *Slides from Linux Kongress*, vol. 18, 2010.

[11] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, pp. 89–100, ACM, 2007.

[12] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pp. 114–122, ACM, 2006.