

ARGONNE NATIONAL LABORATORY  
9700 South Cass Avenue  
Argonne, Illinois 60439

## Exploiting Performance Portability in Search Algorithms for Autotuning<sup>1</sup>

**Amit Roy, Prasanna Balaprakash,  
Paul D. Hovland, and Stefan M. Wild**

Mathematics and Computer Science Division

Preprint ANL/MCS-P5397-0915

September 2015 (*Revised February 2016*)

**Updates to this preprint may be found at**  
<http://www.mcs.anl.gov/publications>

---

<sup>1</sup>This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under contract number DE-AC02-06CH11357.

# Exploiting performance portability in search algorithms for autotuning

Amit Roy<sup>\*†</sup>

<sup>\*</sup> School of Computing  
University of Utah  
Salt Lake City, Utah 84112  
aroy@cs.utah.edu

Prasanna Balaprakash<sup>†</sup>, Paul D. Hovland<sup>†</sup>, and Stefan M. Wild<sup>†</sup>

<sup>†</sup>Mathematics & Computer Science Division  
Argonne National Laboratory  
Lemont, Illinois 60439  
{pbalapra,hovland,wild}@anl.gov

**Abstract**—Autotuning seeks the best configuration of an application by orchestrating hardware and software knobs that affect performance on a given machine. Autotuners adopt various search techniques to efficiently find the best configuration, but they often ignore lessons learned on one machine when tuning for another machine. We demonstrate that a surrogate model built from performance results on one machine can speedup the autotuning search by 1.6X to 130X on a variety of modern architectures.

**Keywords**—Autotuning, Empirical search heuristics, Performance portability

## I. INTRODUCTION

The ever-increasing complexity of mapping large-scale scientific codes to High-performance computing architectures presents significant obstacles for scientific productivity. Application developers and performance engineers often resort to manually rewriting the code for the target machine, a painstaking and time-consuming effort that is neither scalable nor portable. Automatic empirical performance tuning (in short, *autotuning*) attempts to address the limitations of such manual tuning. Autotuning consists of identifying relevant application, hardware, and system-software parameters; assigning a range of parameter values using hardware expertise and application-specific knowledge; and then systematically exploring this parameter space to find the best-performing parameter configuration on the target machine.

A given application is typically autotuned from scratch and independently on each new target machine. This approach is motivated by the conventional wisdom that the performance of a code configuration on one machine is not portable to another machine because of the differences in hardware and system software. Although this is generally true, one has to wonder whether sets of high-quality parameter configurations will be similar for similar machines, such as consecutive generations within a particular vendor’s product line. We hypothesize that although the best configuration obtained from one machine might not be the best on another machine, knowledge on the high- and low-performing parameter configurations from one machine can be exploited to speedup autotuning on another machine.

For illustration, consider 200 configurations of a LU decomposition kernel, where each configuration is generated

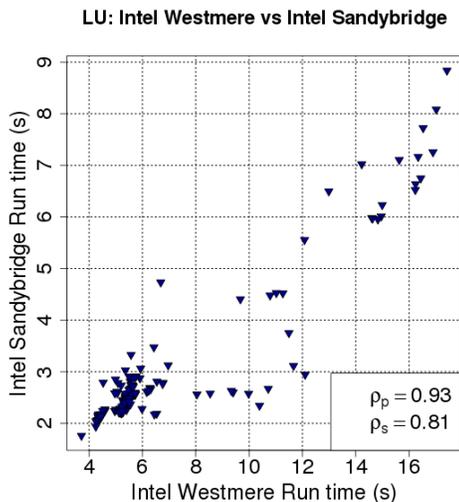


Figure 1. Run times of LU decomposition kernel code variants on Intel E5645 and E5-2687W machines.

with a specific loop unroll, cache, and register tiling value. Figure 1 shows the run times of these configurations on Intel E5645 (Westmere) and E5-2687W (Sandybridge) machines. Despite the difference in the actual run times, we observe a strong correlation (Pearson and Spearman correlation coefficients  $\rho_p$  and  $\rho_s$  greater than 0.8, respectively); the configurations with shorter (longer) run times on Westmere and Sandybridge are similar. Given such a trend, the results on Westmere can be exploited to speed autotuning on Sandybridge (and vice versa).

In this paper, we develop a systematic way of using performance data obtained on one machine to benefit autotuning on another machine. The main aspects of the proposed approach are as follows. Given a set of parameter configurations and their corresponding application run times obtained from a particular machine, we build a surrogate performance model using a machine-learning approach. We then use this model to guide the autotuning search on another machine. We investigate two guiding strategies: a conservative pruning strategy, in which the surrogate performance model is used to identify and avoid evaluating potentially

poor configurations, and a greedy biasing strategy, in which the search is restricted to configurations that the model predicts to have shorter run times.

This paper makes the following contributions:

- From a methodological perspective, we propose a novel machine-learning-based approach for using autotuning results from one machine on another machine;
- From an algorithmic viewpoint, we develop new model-based random search variants for autotuning;
- From an experimental evaluation and benchmarking standpoint, we present performance portability experiments on a wide range of modern architectures. In particular, for the first time, we show the existence of performance correlations between Intel Sandybridge and IBM Power 7.

## II. PROBLEM AND SETUP

Given an application (or kernel), a set of tunable parameters, and a target machine, the goal of autotuning is to find an optimal configuration with respect to a user-defined performance metric. Formally, autotuning seeks to solve

$$\min_x \{f(x; \alpha, \beta, \gamma) : x \in \mathcal{D}(\alpha, \beta, \gamma) \subset \mathbb{R}^m\}, \quad (1)$$

where  $x$  is a vector of  $m$  tuning parameter values,  $\mathcal{D}$  denotes the set of allowable configurations,  $\alpha$  denotes the given application,  $\beta$  is a set of hyperparameters, and  $\gamma$  is the target machine. The value  $f(x)$  is the performance metric, typically run time, for the parameter configuration  $x$ . In the hyperparameters  $\beta$ , we include parameters such as the input size of the application, CPU/DRAM frequency, and compiler type and its corresponding flags; each of these can potentially affect the reported value  $f(x)$  but typically remain unchanged and uncontrolled during autotuning. In general, the space of allowable configurations depends on the values of  $\alpha$ ,  $\beta$ , and  $\gamma$ .

In autotuning work, the best configuration obtained by evaluating all configurations in  $\mathcal{D}$  or is approximated by evaluating configurations in a subset  $\mathcal{D}' \subset \mathcal{D}$  obtained by pruning nonpromising configurations using architecture-and/or application-specific information [3]. As codes and architectures become more complex, however, such approaches become computationally prohibitive because of the large number of allowable parameter configurations. Consequently, a scalable approach to autotuning includes using a search algorithm to systematically evaluate a tiny subset of configurations on the target machine in order to identify the best (or approximately best) parameter configurations.

Search algorithms that have been deployed for autotuning include variants of random search, genetic algorithms, simulated annealing, particle swarm, Nelder-Mead, orthogonal search, pattern search, and model-based search [30, 18, 32, 17, 6]. These algorithms share a common theme: evaluations obtained before iteration  $k$  are used to determine configuration(s) to evaluate during iteration  $k$ . For random

search without replacement (henceforth, “RS”), however, this dependence on the past is limited to avoiding repeated evaluation of any configuration.

In this paper, we use RS to illustrate the effectiveness of the proposed approach. We select RS due to its simplicity: parameter configurations are sampled uniformly at random from the feasible domain  $\mathcal{D}$  without replacement. At iteration  $k$ , each unevaluated configuration  $x \in \mathcal{D}$  has probability  $\frac{1}{|\mathcal{D}|-k+1}$  of being selected for evaluation. The algorithm is terminated early when a predefined number of evaluations (or wall clock time limit) has been exceeded; otherwise, it stops after  $|\mathcal{D}|$  iterations with the global optimum. For algorithms other than RS, it would be difficult in attributing observed benefits only to the proposed approach, since the search uses previous evaluations to decide which configurations to evaluate next. Furthermore, RS has been shown to be effective on a number of performance-tuning tasks [30, 18].

## III. PROPOSED APPROACH

We let  $\mathcal{A}$  denote a set of architectures and/or compilation types (to which we refer collectively as a set of “machines”). Given an application  $\alpha$  with a fixed input size, the autotuning problem for machine  $\gamma_a \in \mathcal{A}$  is thus to minimize  $f(x; \alpha, \beta_a, \gamma_a)$  as a function of  $x$ . We make the important assumption that the set of allowable configurations does not vary across the set  $\mathcal{A}$ , that is,  $\mathcal{D} = \mathcal{D}(\alpha) = \mathcal{D}(\alpha, \beta_a, \gamma_a)$  for all  $\gamma_a \in \mathcal{A}$ .

We take  $\mathcal{T}_a = \{(x_1, y_1), \dots, (x_l, y_l)\}$ , where  $x_i \in \mathcal{D}$  and  $y_i = f(x_i; \alpha, \beta_a, \gamma_a)$  are parameter configurations and the corresponding run times obtained on the machine  $\gamma_a \in \mathcal{A}$ . Given the same application  $\alpha$  to be tuned with the same input size, our approach consists of using  $\mathcal{T}_a$  to develop a surrogate performance model of  $f(\cdot; \alpha, \beta_b, \gamma_b)$  for some other machine  $\gamma_b \in \mathcal{A}$  and to use this model to accelerate the search for an optimal  $x \in \mathcal{D}$  on machine  $\gamma_b$ . When possible, it may be desirable to keep some hyperparameter settings the same on  $\gamma_a$  and  $\gamma_b$  (e.g., as we do with compiler type in our experiments). In our formulation, this corresponds to partitioning  $\beta$  as  $\{\beta^1, \beta^2\}$  so that  $\beta_b = \{\beta_a^1, \beta_b^2\}$ , where  $\beta_a^1$  denotes the set of hyperparameters kept constant.

To develop the surrogate performance model, we adopt a supervised machine-learning approach that seeks a surrogate function  $\mathcal{M}$  for the expensive  $f$  such that the difference between  $f(x)$  and  $\mathcal{M}(x)$  is minimal for all  $x \in \{x_1, \dots, x_l\}$  (and, ideally, for all  $x \in \mathcal{D}$ ). The function  $\mathcal{M}$ , which is called an empirical performance model, can be used to predict the run times of all  $x \in \mathcal{D}$  (not just those  $x_i$  for which  $f(x_i)$  has been evaluated).

We investigate two ways of using the surrogate model obtained from  $\mathcal{T}_a$  to accelerate the search on machine  $\gamma_b$ : a pruning strategy, where we hypothesize that the poor configurations on machines  $\gamma_a$  and  $\gamma_b$  will be highly correlated, and a biasing strategy, where we hypothesize that the

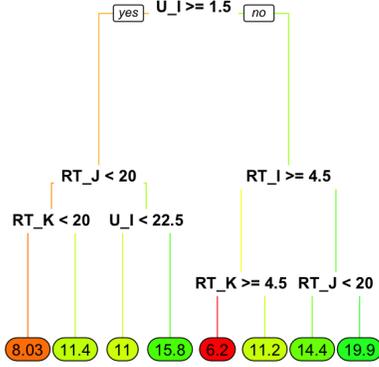


Figure 2. Decision tree obtained from matrix-multiplication kernel data on Intel Sandybridge. The input parameters are unrolls (U\_I, U\_J, U\_K) and register tilings (RT\_I, RT\_J, RT\_K) for the loops  $i$ ,  $j$ , and  $k$ .

high-performing configurations on the two machines will be highly correlated.

In the remainder of this section, we describe the supervised-learning approach for obtaining a surrogate performance model and how the pruning and biasing strategies are implemented in RS using the surrogate model.

#### A. Surrogate performance model

The choice of the supervised-learning algorithm for building the surrogate performance model is crucial. Often this choice should be driven by an exploratory analysis of the relationship between the parameter configurations and their corresponding run times. Our previous studies in performance modeling and tuning [5, 4], shows that recursive partitioning approaches are well suited for the surrogate performance modeling. Here, we employ random forest (RF), a state-of-the-art recursive partitioning method [9].

Given a set of training points  $\mathcal{T}_a$ , RF proceeds as follows. The multidimensional input space  $\mathcal{D}$  is partitioned into a number of hyperrectangles using the run times of the parameter configurations. On each hyperrectangle, further partitioning takes place recursively until the run times within the partition are the same or when partitioning does not improve certain criteria. Typically, partitions are described by a decision tree with *if-else* rules as shown in Figure 2. The paths from the root to each leaf define the set of hyperrectangles. The value in each leaf corresponds to the mean run times of the training configurations that fall within the associated hyperrectangle. Given an unseen input configuration, the decision tree is used to identify the hyperrectangle to which this input belongs and returns the corresponding leaf value as the predicted value. The power of RF comes from the fact that it uses a collection of decision trees, each of which is built on a different random subsample of points from the entire training set  $\mathcal{T}_a$ . Consequently, RF can model nonlinear interactions and relationships between the inputs and their corresponding output.

---

#### Algorithm 1 Random search with pruning strategy.

---

**Input:**  $\mathcal{T}_a$  from machine  $\gamma_a$ , max evaluations  $n_{\max}$ , configuration pool size  $N \gg n_{\max}$ , cutoff parameter  $\delta$

```

1  $\mathcal{M}_a \leftarrow \text{fit}(\mathcal{T}_a)$ 
2  $\mathcal{X}_p \leftarrow \text{sample } N \text{ distinct configurations from } \mathcal{D}$ 
3  $\mathcal{Y}_p \leftarrow \text{predict}(\mathcal{M}_a, \mathcal{X}_p)$ 
4  $\Delta \leftarrow \delta\% \text{ quantile of } \mathcal{Y}_p$ 
5  $\mathcal{X}_{\text{out}}, \mathcal{Y}_{\text{out}} \leftarrow \{\}$ 
6  $k \leftarrow 1$ 
7 while  $k \leq n_{\max}$  do
8    $x_k \leftarrow \text{sample from } \mathcal{D} \cap (\mathcal{X}_{\text{out}})^c$ 
9   if  $\text{predict}(\mathcal{M}_a, x_k) \leq \Delta$  then
10     $y_k \leftarrow \text{Evaluate}(x_k)$ 
11     $\mathcal{X}_{\text{out}} \leftarrow \mathcal{X}_{\text{out}} \cup x_k; \mathcal{Y}_{\text{out}} \leftarrow \mathcal{Y}_{\text{out}} \cup y_k$ 
12     $k \leftarrow k + 1$ 

```

**Output:** Best parameter configuration from  $\mathcal{X}_{\text{out}}$

---

#### Algorithm 2 Random search with biasing strategy.

---

**Input:**  $\mathcal{T}_a$  from machine  $\gamma_a$ , max evaluations  $n_{\max}$ , configuration pool size  $N \gg n_{\max}$

```

1  $\mathcal{M}_a \leftarrow \text{fit}(\mathcal{T}_a)$ 
2  $\mathcal{X}_p \leftarrow \text{sample } N \text{ distinct configurations from } \mathcal{D}$ 
3  $\mathcal{Y}_p \leftarrow \text{predict}(\mathcal{M}_a, \mathcal{X}_p)$ 
4  $\mathcal{X}_{\text{out}}, \mathcal{Y}_{\text{out}} \leftarrow \{\}$ 
5  $k \leftarrow 1$ 
6 while  $k \leq n_{\max}$  do
7    $x_k \leftarrow \arg \min_{x \in \mathcal{X}_p} \text{predict}(\mathcal{M}_a, x)$ 
8    $y_k \leftarrow \text{Evaluate}(x_k)$ 
9    $\mathcal{X}_p \leftarrow \mathcal{X}_p \setminus x_k$ 
10   $\mathcal{X}_{\text{out}} \leftarrow \mathcal{X}_{\text{out}} \cup x_k; \mathcal{Y}_{\text{out}} \leftarrow \mathcal{Y}_{\text{out}} \cup y_k$ 
11   $k \leftarrow k + 1$ 

```

**Output:** Best parameter configuration from  $\mathcal{X}_{\text{out}}$

---

#### B. Pruning and biasing strategies in random search

In the pruning strategy, we use the surrogate performance model to identify potentially poor configurations and avoid evaluating them; in the biasing strategy, we restrict the search to configurations that the model predicts to have better run times. The pruning and biasing strategies in RS are shown in Algorithms 1 and 2, respectively.

Given a maximum number of evaluations  $n_{\max}$  as a budget, data  $\mathcal{T}_a$  from machine  $\gamma_a$ , and a cutoff parameter  $0 < \delta < 100$ , RS with pruning proceeds in two phases. In the first phase, the algorithm fits a surrogate performance model  $\mathcal{M}_a$  using  $\mathcal{T}_a$ . It then samples  $N \gg n_{\max}$  configurations at random and predicts the corresponding run times  $\mathcal{Y}_p$ . A cutoff value  $\Delta$  is estimated by computing the  $\delta\%$  quantile of  $\mathcal{Y}_p$ . The iterative phase of the algorithm consists of sampling an unevaluated configuration at random, predicting its run time using  $\mathcal{M}_a$ , and evaluating the configuration on the target machine(s) only when its predicted run time is smaller than  $\Delta$ .

RS with biasing also proceeds in two phases. The only difference in the first phase is that the algorithm does not require an estimate of the cutoff. The key difference is in the second phase: at each iteration an unevaluated configuration

Table I  
ORIO TRANSFORMATIONS CONSIDERED.

Transformation	Description	Range
Loop unrolling	data reuse	$1, \dots, 31, 32$
Cache tiling	cache hits	$2^0, \dots, 2^{10}, 2^{11}$
Register tiling	cache to register loads	$2^0, \dots, 2^4, 2^5$

with the smallest predicted run time according to  $\mathcal{M}_a$  is selected for evaluation on the target machine(s).

#### IV. EXPERIMENTAL SETUP

In this section we describe the experimental setup, test problems, and machines used for the experiments.

##### A. Autotuners

In our experiments, we use two autotuning frameworks: Orio [17] and OpenTuner [2]. They are extensible and portable frameworks for empirical performance tuning. Orio takes as input an annotated C or Fortran code specifying potential code transformations (see Table I for the transformations used in our experiments), their possible parameter values, and a search strategy. The search algorithm in Orio generates multiple versions of the source code based on the different code transformations and runs the resulting configurations on a target machine in order to find the one with the best run time. We refer the reader to [17] for details about the annotation parsing and code-generation schemes in Orio. OpenTuner is a framework for building domain-specific program autotuners. It provides extensible infrastructure to support complex and user-defined data types and custom search heuristics. It runs a number of search techniques at the same time; those that perform well are allocated larger budgets systematically using optimal budget allocation methods. Unlike Orio, it does not provide automatic code transformation recipes; typically the user has to write the code-specific transformation modules. Nonetheless, it is well suited for tuning compiler and command line parameters of full applications.

##### B. Machines

Experiments were run on Intel Westmere, Intel Sandybridge, Intel Xeon Phi, IBM Power 7, and AppliedMicro X-Gene ARM 64-bit at Argonne’s Joint Laboratory for System Evaluation. Table II shows the machine specifications used in our study. As a default, we use the GNU compiler (v4.4.7) with the `-O3` the optimization flag, since this is

supported on all the tested architectures. Moreover, we also conducted experiments with Intel compiler (v15.0.1 with `-O3` optimization) involving Intel machines.

##### C. Kernels and mini-applications

For kernels, we use problems from the SPAPT test suite [7], which are implemented in an annotation-based language that can be readily processed by Orio and are used primarily for testing the effectiveness of search algorithms. Each search problem is a particular combination of a kernel, an input size, a set of tunable decision parameters, a feasible set of possible parameter values, and a default/initial configuration of these parameters for use by search algorithms. We selected the following kernels to cover all these groups with sufficient complexity and problem size to capture interaction among all levels of the memory hierarchy. **Matrix Multiply (MM)** is used in many scientific applications. It is usually bounded by the compute capability [33], implying that performance improvements are limited by the architecture’s speed in executing floating-point operations.  **$A^T Ax$  (ATAX)** is a matrix transpose and matrix multiplication followed by a vector multiplication operation. ATAX is commonly found in statistical applications and is inherently bounded by memory bandwidth because of the low floating-point-operation intensity. **Correlation (COR)** measures how two data sets are related. This is a common operation used in data analysis. This kernel is usually memory bounded because of the low number of floating-point operations. **LU Decomposition (LU)** is a common matrix operation that factorizes a matrix  $A$  into a lower  $L$  and an upper  $U$  triangular matrix such that  $A = LU$ . It is used to solve a system of linear equations. This kernel is also usually memory bound because of the low ratio of floating-point operations to memory operations. The number of parameters, search space size, and input size for each of the four SPAPT problems considered are shown in Table III. Note that each parameter applies to each loop level hence we will have a total of 9 independent parameters to tune for 3 nested loops. Large parameter search spaces are chosen in order to find non-intuitive optimal points. For instance, we observed that one of the faster code variant for MM kernel on Intel Westmere is obtained with a loop unroll factor of 25.

We consider two mini-applications and use Opentuner infrastructure for autotuning. **High Performance LINPACK (HPL)** is a widely used benchmark to evaluate performance

Table II  
DESCRIPTION OF ARCHITECTURE SET CONSIDERED.

Name	Processor	Cores	Clock Speed (GHz)	L1 (KB)	L2 (KB)	L3 (MB)	Memory (GB)
Sandybridge	Intel E5-2687W	8	3.4	32	256	20 (shared)	64
Westmere	Intel E5645	6	2.4	32	256	12 (shared)	48
Xeon Phi	Intel Xeon Phi 7120a	61	1.24	32	512	-	16
Power 7	IBM Power7+	6	4.2	32	256	10 (per core)	128
X-Gene	APM883208-X1	8	2.4	32	256	8 (shared)	16

Table III  
COLLECTION OF TEST KERNELS CONSIDERED.

Kernel	$n_i$	Search Space Size	Input Size
MM	12	8.58e+10	2000×2000
ATAX	13	2.57e+12	10000
COR	12	8.57e+10	2000×2000
LU	9	5.83e+08	2000×2000

of various Top 500 computing platforms [27]. It solves randomly generated dense linear systems using distributed memory architectures. The benchmark comprises of 15 tunable parameters to achieve optimal performance on a given machine. Some of the tuning parameters include mapping of block size, mapping row or column major to process and other algorithmic parameters. **Raytracer** (RT) is a C++ code to render images of 3D objects [1]. For this mini-application, we focused on tuning g++ flags. We extracted all the supported g++ flags and parameters for each machine and then found the common set that are supported across our test platforms. This resulted in 143 flags and 104 parameters. More information on g++/gcc flag tuning can be found in [2].

#### D. Run setup and metrics

For a given kernel  $\alpha$ , we run RS on the machine  $\gamma_a$ ; collect  $\mathcal{T}_a$ ; and run RS,  $RS_p$ , and  $RS_b$  on the machine  $\gamma_b$ . The controllable factors, such as input size, compiler type, and compiler flags, are kept unchanged on  $\gamma_a$  and  $\gamma_b$ . The maximum number of evaluations,  $n_{\max}$ , and configuration pool size,  $N$ , are set to 100 and 10,000, respectively. The  $n_{\max}$  value corresponds to short autotuning computation budget for SPAPT problems [6]. The  $N$  value can be set to any large arbitrary values but the time to generate  $N$  random configuration (without evaluation) should be within few seconds. In  $RS_p$ , the cutoff parameter  $\delta$  is set to 20%.

To assess the effectiveness of the model in the proposed approach, we include in our comparison model-free variants of  $RS_p$  and  $RS_b$ . The model-free variant of  $RS_p$  (denoted by  $RS_{pf}$ ) computes its value of  $\Delta$  from  $\mathcal{T}_a$  instead of using any model. Then, it evaluates the configurations on  $\gamma_b$  in the same order as on  $\gamma_a$  but avoids evaluating configurations according to the bound  $\Delta$ . The model-free variant of  $RS_b$  (denoted by  $RS_{bf}$ ) sorts the configurations in  $\mathcal{T}_a$  in ascending order with respect to the run times on  $\gamma_a$  and evaluates them in the sorted order on  $\gamma_b$ . Consequently, in both  $RS_{pf}$  and  $RS_{bf}$ , the search on  $\gamma_b$  is restricted to the evaluated configurations  $\mathcal{T}_a$  from  $\gamma_a$ .

Typically, randomized algorithms require multiple independent runs to minimize the variance in their behavior. Due to computationally expensive nature of our experiments and absence of heterogenous parallel test beds, we perform single run of all the algorithms but with the following setting to reduce variance. First we run RS on machine  $\gamma_a$ ; RS on machine  $\gamma_b$  evaluates configurations in the same order as RS on  $\gamma_a$ .  $RS_p$  on  $\gamma_b$  evaluates configurations in

the same order as RS but avoids evaluating configurations according to the bound  $\Delta$ .  $RS_b$  builds model from  $\mathcal{T}_a$ . This setting is essentially the so-called method of common random numbers, a well-known variance reduction technique [29].

For comparison, we compute **performance and search time speedups** with respect to RS. As a defining example, suppose that RS takes 100 s to find its best configuration (with a run time of 5 s) and that  $RS_b$  takes 80 s to find its best configuration (with a run time of 3 s), but requires only 50 s to find a configuration with a run time of 5 s. Then, the performance and search time speedups of  $RS_b$  over RS are 1.6X and 2X, respectively. We consider a variant to be successful on a particular problem when its performance speedup is at least 1.0X and its search time speedup is greater than 1.0X.

## V. RESULTS

First, we focus on results obtained using the GNU compiler (v4.4.7) with the -O3 optimization flag. Figure 3 shows the results from two Intel machines, where we use data from RS on Westemere ( $\gamma_a$ ) to improve search on Sandybridge ( $\gamma_b$ ). The first column shows the results of the model-based RS variants,  $RS_p$ , and  $RS_b$ . Each plot shows the run times of the best-found code version and the elapsed search time for RS and its variants. Table IV summarizes the observed speedups in search time and performance.

**Model-based and model-free RS variants are better than RS:** The plots in Figure 3 show a general trend that the RS variants dominate RS and that  $RS_b$  outperforms  $RS_p$  primarily with respect to search time speedups, which are between 1.6X and 130X. The performance speedups are not as significant as the search time speedups and they range between 1.0X and 1.3X. The second column shows the results of the model-free RS variants,  $RS_{pf}$ , and  $RS_{bf}$ . We observe that  $RS_{bf}$  is significantly better than  $RS_{pf}$  and RS (see also Table IV). However, there are no performance speedups because  $RS_{pf}$  and  $RS_{bf}$  are restricted to the same 100 configurations of RS.

**Biasing is better than pruning:** The superior performance of biasing ( $RS_b$  and  $RS_{bf}$ ) over pruning ( $RS_p$  and  $RS_{pf}$ ) strategies can be explained by observing the correlation plots of RS run times obtained on the two machines, which are shown in the third column of Figure 3. Except for HPL, the plots exhibit a high correlation across various problems. We observe that a large fraction of the high-performing code versions remain the same across Westemere and Sandybridge. Consequently,  $RS_b$  and  $RS_{bf}$  benefit from this strong correlation by identifying the promising configurations and evaluating them first. On the other hand, the conservative pruning strategy does not result in significant speedups, which can be attributed to the cutoff parameter  $\delta$  (= 20%).

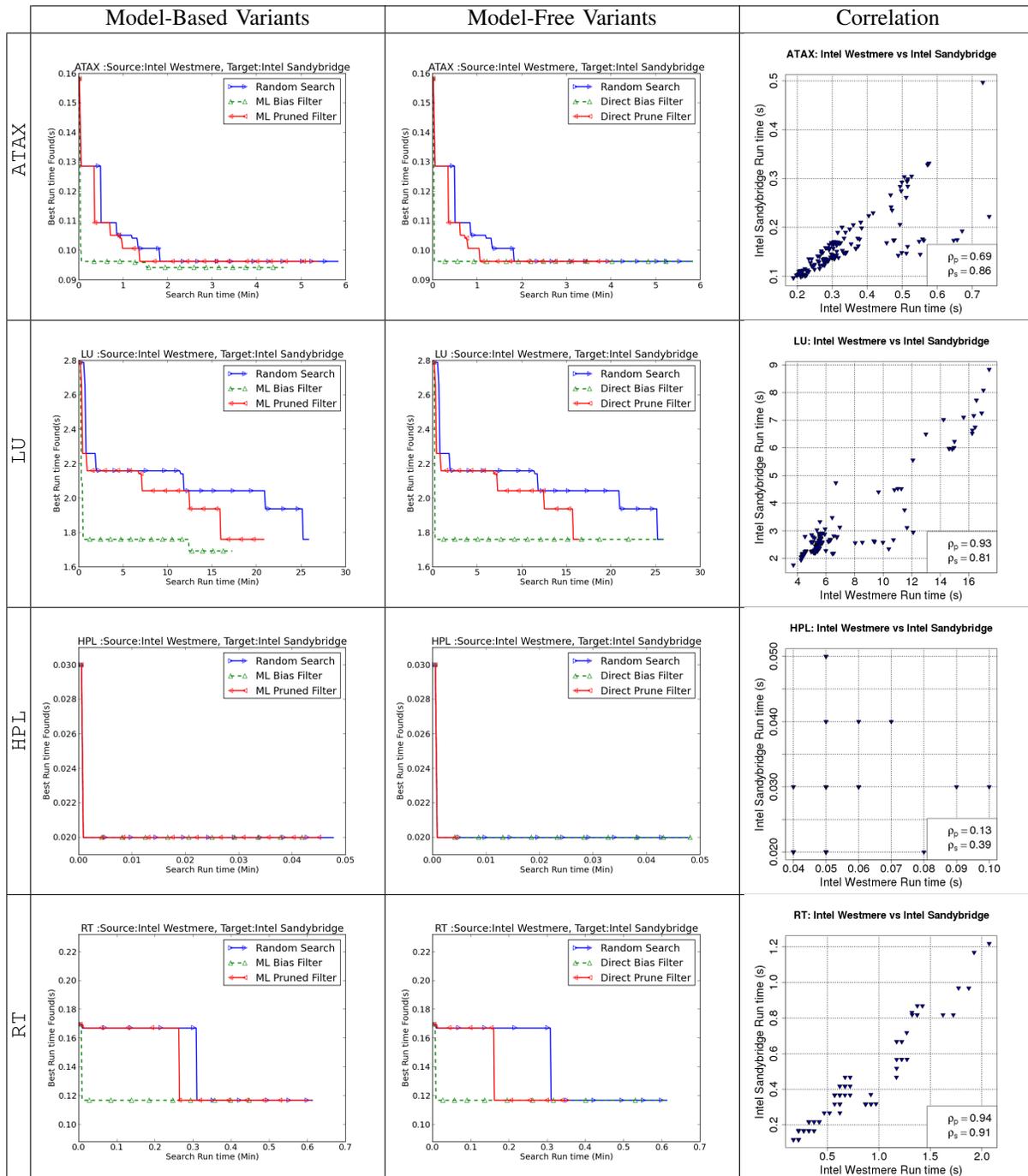


Figure 3. Using Intel Westmere to speed the search on Intel Sandybridge.

**Model-based is better than model-free:** The advantage of the model-based ( $RS_b$ ) over the model-free ( $RS_{bf}$ ) variant is twofold. The adoption of the model enables  $RS_b$  to identify promising configurations from a large configuration pool, which results in performance speedups up to 1.3X. The total search time required by  $RS_b$  is shorter than that of  $RS_{bf}$  because the former tries to evaluate only code versions with shorter run times.

We note that the run time correlation between machines is also affected by the problem being tested. Different problems stress different hardware components that may or may not be similar across machines. This dependence is evident in the correlation plot of ATAX, where there is a loss of correlation between poor performing variants. However, this lack of correlation alone does not affect our  $RS_b$  and  $RS_{bf}$  variants because the high-performing versions are strongly correlated.

From Table IV, we can also observe that data from Sandybridge can speed the search on Westmere. The search time and performance speedups range from 1.8X to 129X and 1X to 1.05X, respectively.

**Sandybridge performance data can be used to speedup search on Power 7:** Figure 4 shows the results on Intel Sandybridge and IBM Power 7. Despite the architectural differences, we observe a similar trend:  $RS_b$  and  $RS_{bf}$  are better than RS,  $RS_p$ , and  $RS_{pf}$  and the effectiveness of RS variants depends on the tested problems.  $RS_b$  obtains search speedups between 15X and 109X and performance speedups between 1.0X and 1.3X (see Table IV). The dissimilarity between the two architectures is evident in the correlation plots. Despite low  $\rho_p$  and  $\rho_s$  values, the  $RS_b$  variant performs well because the high-performing code versions are correlated across different platforms.

**Approach fails on dissimilar machines:** We used RS data from Intel Sandybridge to speedup search on ARM X-Gene for ATAX and LU. We were not able to collect data for all the problems since their run times or compilation times were too high on the ARM X-Gene given the same input size as on other architectures. We found that RS variants do not achieve any significant search time and performance speedups over RS.

We next focus on tuning on the Xeon Phi with Westmere and Sandybridge as source machines, where we use the Intel compiler (v15.0.1) with the -O3 optimization flag. For testing we used the MM, LU, and COR kernels compiled by Intel’s compiler with added OpenMP pragmas to take full advantage of the Xeon Phi. We set 8 threads for Sandybridge and Westmere, respectively, and 60 threads for the Xeon Phi.

The results are shown in Figure 5 and Table V. On MM, we did not get a clear trend because out of all the evaluated code versions, default one without any code transformation is the best on the Xeon Phi. It seems that the Intel compiler is performing all the required transformations and any additional transformations are detrimental to performance. The results

Table IV  
SEARCH PERFORMANCE AND RUN TIME SPEEDUP FOR BIASED, MODEL VARIANT.

				Source					
				Westmere		Sandybridge		Power 7	
				Prf.Imp	Srh.Imp	Prf.Imp	Srh.Imp	Prf.Imp	Srh.Imp
Target	MM	Westmere	-	-	<b>1.05</b>	<b>5.33</b>	<b>1.09</b>	<b>9.60</b>	
		Sandybridge	<b>1.04</b>	<b>28.92</b>	-	-	<b>1.19</b>	<b>7.95</b>	
		Power 7	1.00	<b>1.66</b>	1.00	<b>16.18</b>	-	-	
		X-Gene	-	-	-	-	-	-	
	ATAX	Westmere	-	-	<b>1.00</b>	<b>1.85</b>	<b>1.01</b>	<b>14.25</b>	
		Sandybridge	<b>1.02</b>	<b>29.91</b>	-	-	<b>1.03</b>	<b>17.84</b>	
		Power 7	<i>0.96</i>	<i>0.00</i>	<i>0.98</i>	<i>0.00</i>	-	-	
		X-Gene	<i>0.88</i>	<i>0.00</i>	<i>0.79</i>	<i>0.00</i>	<b>1.11</b>	<b>4.52</b>	
	LU	Westmere	-	-	<b>1.03</b>	<b>129.31</b>	<b>1.03</b>	<b>129.31</b>	
		Sandybridge	<b>1.04</b>	<b>52.56</b>	-	-	<b>1.04</b>	<b>99.90</b>	
		Power 7	<b>1.32</b>	<b>20.67</b>	<b>1.32</b>	<b>109.82</b>	-	-	
		X-Gene	1.00	1.00	1.00	1.00	1.00	1.00	
COR	Westmere	-	-	1.00	<b>4.94</b>	<i>0.97</i>	<i>0.00</i>		
	Sandybridge	1.00	<b>1.76</b>	-	-	<i>0.90</i>	<i>0.00</i>		
	Power 7	<i>0.84</i>	<i>0.00</i>	<b>1.00</b>	<b>25.75</b>	-	-		
	X-Gene	-	-	-	-	-	-		
HPL	Westmere	-	-	1.00	<b>4.78</b>	1.00	<b>1.79</b>		
	Sandybridge	1.00	1.00	-	-	1.00	1.00		
	Power 7	1.00	<i>0.45</i>	1.00	<b>2.90</b>	-	-		
	X-Gene	<i>0.88</i>	<i>0.00</i>	<i>0.88</i>	<i>0.00</i>	1.00	<b>2.42</b>		
RT	Westmere	-	-	<b>1.00</b>	<b>4.60</b>	<i>0.77</i>	<i>0.00</i>		
	Sandybridge	1.00	<b>29.96</b>	-	-	1.00	<i>0.00</i>		
	Power 7	<b>1.00</b>	<b>30.04</b>	<b>1.00</b>	<b>3.68</b>	-	-		
	X-Gene	1.00	<i>0.00</i>	<b>1.00</b>	<i>0.19</i>	<b>1.12</b>	<b>10.71</b>		

Note: Prf.Imp and Srh.Imp denote performance and search time speedups respectively. A value is typesetted in bold when  $RS_b$  obtains better code variant in shorter search time than RS.

Table V  
SEARCH PERFORMANCE AND RUN TIME SPEEDUP FOR BIASED, MODEL-BASED VARIANT FOR XEON PHI EXPERIMENTS.

				Source					
				Westmere		Sandybridge		Xeon Phi	
				Perf.Imp	Srh.Imp	Perf.Imp	Srh.Imp	Perf.Imp	Srh.Imp
Target	MM	Westmere	-	-	1.00	<b>165.49</b>	<i>0.92</i>	<i>0.00</i>	
		Sandybridge	1.00	1.00	-	-	1.00	1.00	
		Xeon Phi	1.00	1.00	1.00	1.00	-	-	
	LU	Westmere	-	-	<b>1.09</b>	<b>41.45</b>	<b>1.10</b>	<b>168.89</b>	
		Sandybridge	<b>1.34</b>	<b>514.49</b>	-	-	<b>1.17</b>	<b>120.67</b>	
		Xeon Phi	<b>1.63</b>	<b>850.53</b>	<b>1.61</b>	<b>850.53</b>	-	-	
	COR	Westmere	-	-	<b>1.29</b>	<b>24.95</b>	<b>1.06</b>	<b>4.12</b>	
		Sandybridge	<b>1.17</b>	<b>248.02</b>	-	-	<b>1.20</b>	<b>5.90</b>	
		Xeon Phi	<b>1.44</b>	<i>0.52</i>	<i>0.49</i>	<i>0.00</i>	-	-	

See Note in Table IV for header and typeset explanation.

on LU show that  $RS_b$  dominates RS and  $RS_p$ . It obtains search-time speedups of 850X and performance speedup of 1.6X. However, on COR, although  $RS_b$  quickly identifies promising configurations, it eventually fails to outperform RS and  $RS_p$  in performance speedup. Furthermore, because of the adoption of the model, the overall search time of  $RS_b$  is shorter than that of all other variants. We observe a similar trend even when Westmere is used as a source.

## VI. RELATED WORK

Using cross-architecture performance data to improve autotuning can be seen as a warm-start approach, a well-known technique to speedup numerical optimization algorithms (see, e.g., [35]). Any cross-architecture performance-projection approaches can be deployed within our proposed framework to speed search algorithms. GROPHECY [20] is a performance-projection framework that can estimate the

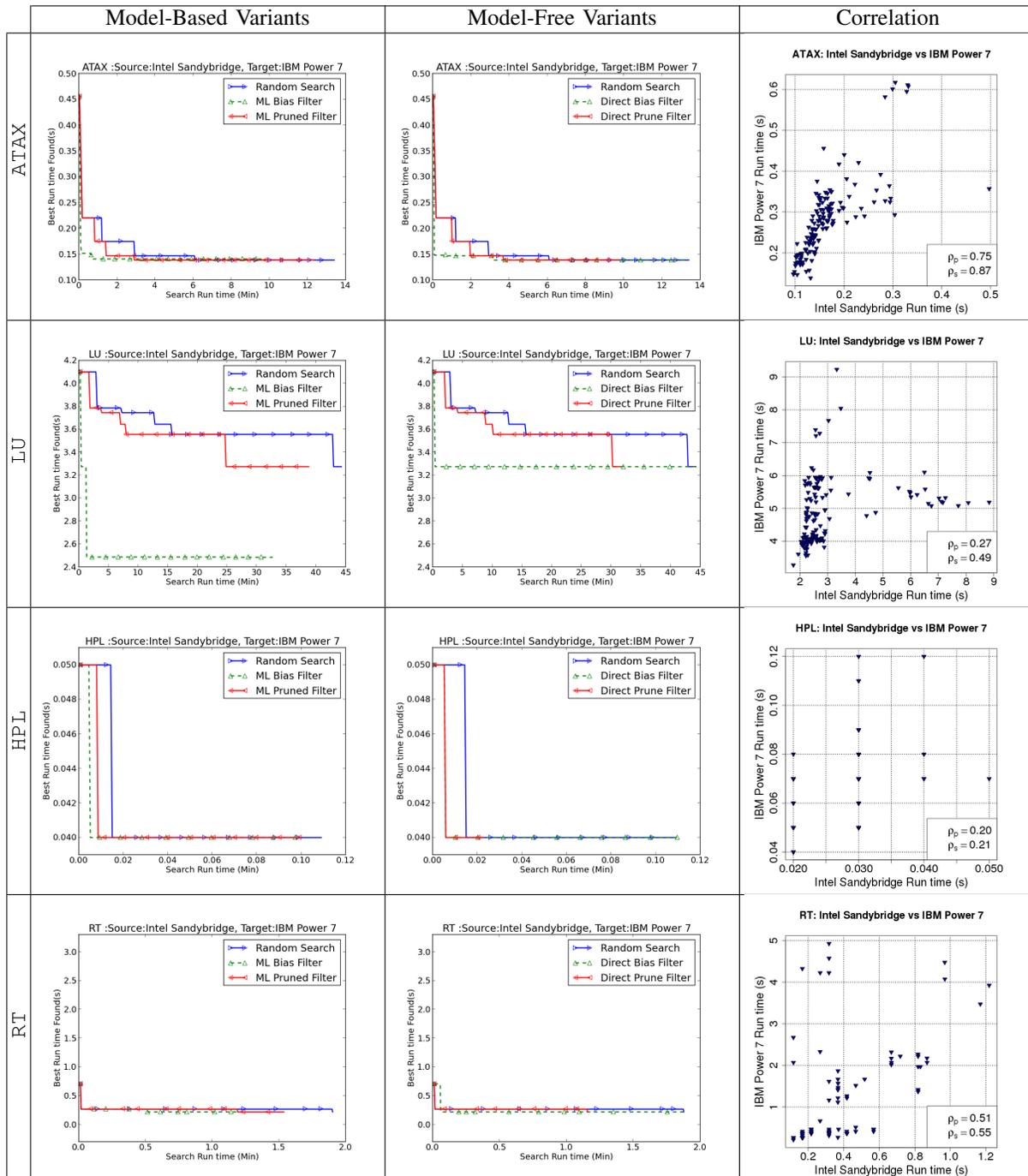


Figure 4. Intel Sandybridge used to speed the search on IBM Power 7.

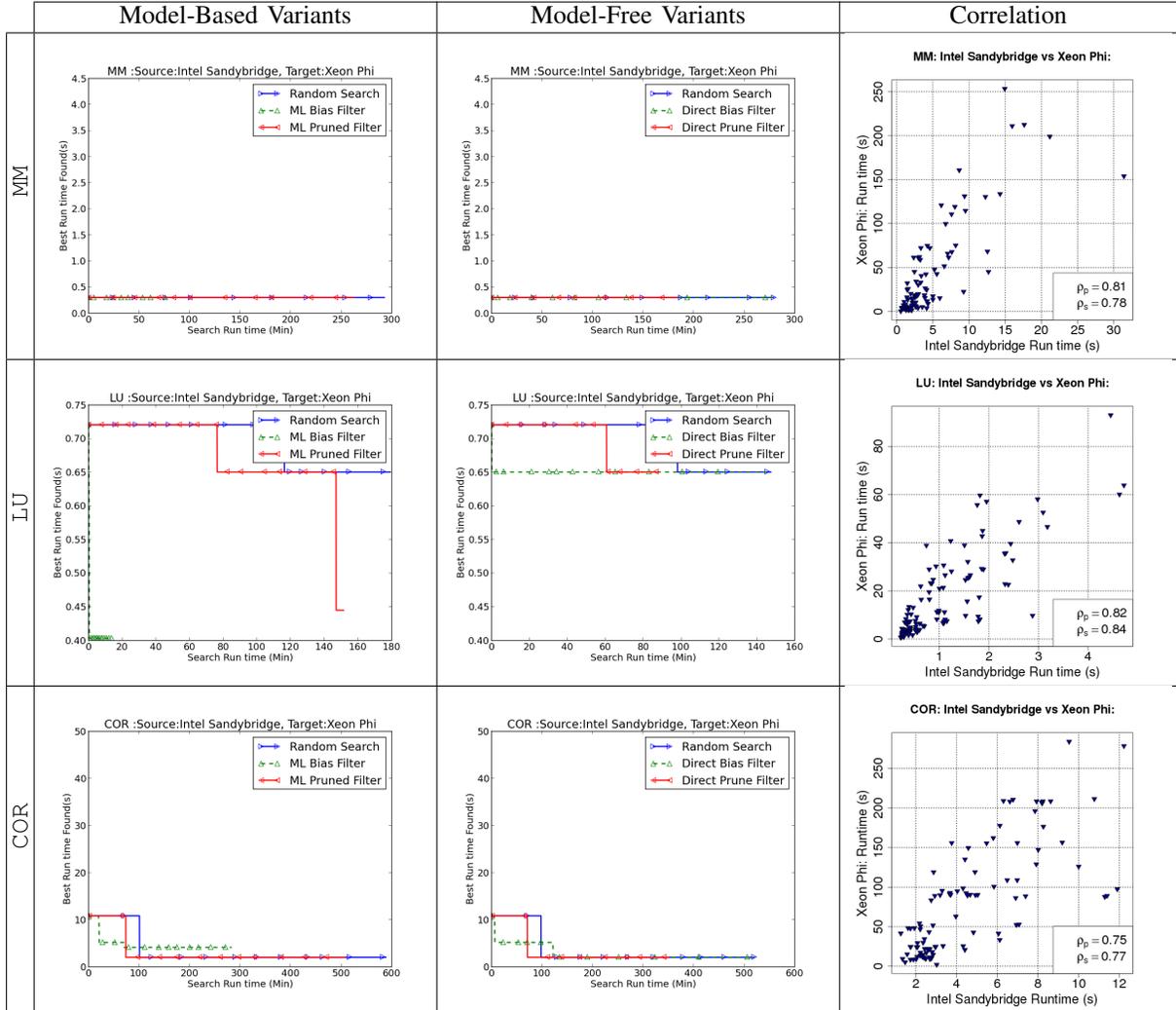


Figure 5. Intel Sandybridge used to speed search on Xeon Phi.

performance of GPU implementations by analyzing their corresponding CPU code skeletons. Although the CPU to GPU code transformation is automatic, it relies on user-supplied code skeletons and analytical models of the target GPU. SKOPE [21] is an extension of GROPHECY to any type of architecture for which a user can supply a specific form of analytical model. PACE [25] is a performance-projection framework designed to guide and speedup code design and program scheduling for a targeted architecture. PACE uses a collection of analytical models for various components of an architecture’s hierarchy; these models can also be deployed within our framework. Other work in performance projection, such as [10, 8], can also be used within our framework.

The adoption of machine learning to improve the auto-tuning search has received much attention in recent years. Whether in an offline or an online setting, the focus in all these works is to use machine learning to search faster within

a single machine. Examples include [13, 15, 11, 26, 16, 24, 4, 23, 19, 31].

A number of systems support the tuning of optimization parameters by means of search, but again they are restricted to single targeted machine. Such systems include Active Harmony [32] (integrated with the CHiLL loop transformation framework [12] to generate configurations), POET [34], Orio [17], Sequoia [28], the X-Language [14], and the approach in [22].

## VII. CONCLUSION

We proposed a machine-learning-based approach to use performance data obtained from one machine to speed auto-tuning on another machine. The key aspect of the proposed approach consists of building a surrogate performance model from the performance data from one machine to bias the auto-tuning search toward promising parameter configurations on the target machine. The experiments on various machines

and kernels showed that the proposed approach resulted in significant search time speedups.

Although Sandybridge and Power 7 are from different vendors, the promising configurations are similar. However, we did not observe such a trend on ARM. Quantification of the dissimilarity between source and target machines requires further investigation, and the proposed approach will greatly benefit from empirical methods that can assess the dissimilarity. Although the observed search time speedups are significant, the performance speedups are small because of the adoption of random search. We will test the proposed approach with other sophisticated search algorithms in order to achieve performance improvements. We will also investigate whether the proposed approach can be generalized for different input sizes, compiler settings, and applications.

#### ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under contract number DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation at Argonne National Laboratory. We are grateful to Mary Hall for useful discussions.

#### REFERENCES

- [1] Writing a simple raytracer. <http://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing>.
- [2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. OpenTuner: an extensible framework for program autotuning. In *Int. Conf. Par. Arch. Compil. Tech.*, pages 303–316, 2014.
- [3] D. Bailey, R. Lucas, and S. Williams, editors. *Performance Tuning of Scientific Applications*. Chapman & Hall/CRC Press, 2010.
- [4] P. Balaprakash, Y. Alexeev, S. Mickelson, S. Leyffer, R. Jacob, and A. Craig. Machine-learning-based load balancing for community ice code component in CESM. In M. Daydé, O. Marques, and K. Nakajima, editors, *11th Int. Conf. High Performance Computing for Computational Science (VECPAR 2014)- Revised Selected Papers*, volume 8969 of *LNCS*, pages 79–91. Springer, 2015.
- [5] P. Balaprakash, R. B. Gramacy, and S. M. Wild. Active-learning-based surrogate models for empirical performance tuning. In *IEEE Int. Conf. Cluster Comput.*, pages 1–8, 2013.
- [6] P. Balaprakash, S. M. Wild, and P. D. Hovland. An experimental study of global and local search algorithms in empirical performance tuning. In *10th Int. Conf. High Performance Computing for Computational Science (VECPAR 2012)- Revised Selected Papers*, LNCS, pages 261–269. Springer, 2013.
- [7] P. Balaprakash, S. M. Wild, and B. Norris. SPAPT: Search problems in automatic performance tuning. *Proc. Comp. Sci.*, 9:1959–1968, 2012.
- [8] S. Becker, H. Koziol, and R. Reussner. Model-based performance prediction with the palladio component model. In *Proc. 6th Int. Wksp. Softw. Perf.*, WOSP '07, pages 54–65, 2007.
- [9] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, Oct. 2001.
- [10] C. Cascaval, L. D. Rose, D. A. Padua, and D. A. Reed. Compile-time based performance prediction. In *Proc. 12th Int. Wkshp. Lang. Compilers Par. Comp.*, LCPC '99, pages 365–379, 2000.
- [11] J. Cavazos. Intelligent compilers. In *IEEE Int. Conf. Cluster Comput.*, pages 360–368, 2008.
- [12] C. Chen. *Model-guided Empirical Optimization for Memory Hierarchy*. PhD thesis, Univ. Southern California, 2007.
- [13] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. ACME: Adaptive compilation made efficient. *ACM SIGPLAN Notices*, 40(7):69–77, 2005.
- [14] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In *Proc. 18th Int. Conf. Languages Compilers Par. Comp. (LCPC '05)*, pages 136–151, 2006.
- [15] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. *ACM SIGPLAN Notices*, 40(7):78–86, 2005.
- [16] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, et al. MILEPOST GCC: Machine learning based research compiler. In *GCC Summit*, 2008.
- [17] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *IEEE Int. Sym. Parallel Distr. Proc. (IPDPS 2009)*, pages 1–11, 2009.
- [18] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *2000 Int. Conf. Parallel Arch. Compil. Tech.*, 2000.
- [19] A. Magni, C. Dubach, and M. F. P. O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *Proc. Int. Conf. High Perf. Comp. Networking Storage Anal. (SC 2013)*, pages 11:1–11:11. ACM, 2013.
- [20] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. Grophecy: GPU performance projection from CPU code skeletons. In *Int. Conf. High Perf. Comp. Networking Storage Anal. (SC 2011)*, pages 1–11, 2011.
- [21] J. Meng, X. Wu, V. Morozov, V. Vishwanath, K. Kumaran, and V. Taylor. SKOPE: A framework for modeling and exploring workload behavior. In *Proc. 11th ACM Conf. Comp. Front. (CF 2014)*, pages 6:1–6:10, 2014.
- [22] D. Merrill, M. Garland, and A. Grimshaw. Policy-based tuning for performance portability and library co-optimization. In *Proc. Innov. Parallel Comp. (InPar 2012)*, 2012.
- [23] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro. Nitro: A framework for adaptive code variant tuning. In *IEEE Int. Sym. Parallel Distr. Proc. (IPDPS 2014)*, pages 501–512, 2014.
- [24] T. Nelson, A. Rivera, P. Balaprakash, M. Hall, P. Hovland, E. Jessup, and B. Norris. Generating efficient tensor contractions for GPUs. In *2015 Int. Conf. Par. Proc.*, pages 969–978, 2015.
- [25] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace—a toolset for the performance prediction of parallel and distributed systems. *Int. J. High Perform. Comput. Appl.*, 14(3):228–251, 2000.
- [26] E. Park, S. Kulkarni, and J. Cavazos. An evaluation of different modeling techniques for iterative compilation. In *ACM Int. Conf. Compilers Arch. Synth. Embed. Sys.*, pages 65–74, 2011.
- [27] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL—a portable implementation of the high-performance linpack benchmark for distributed-memory computers, 2008.
- [28] M. Ren, J. Y. Park, M. Houston, A. Aiken, and W. J. Dally. A tuning framework for software-managed memory hierarchies. In *Proc. 17th Int. Conf. Par. Arch. Compil. Tech. (PACT '08)*, pages 280–291. ACM, 2008.
- [29] R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo Method*, volume 707. John Wiley & Sons, 2011.
- [30] K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In *IEEE Int. Conf. Cluster Comput.*, pages 421–429, 2008.
- [31] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proc. Int. Sym. Code Gener. Optimiz. (CGO '05)*, pages 123–134. IEEE, 2005.
- [32] A. Tiwari, C. Chen, C. Jacqueline, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IEEE Int. Sym. Parallel Distr. Proc. (IPDPS 2009)*, pages 1–12, 2009.
- [33] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.
- [34] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. In *IEEE Int. Sym. Parallel Distr. Proc. (IPDPS 2007)*, pages 1–8, 2007.
- [35] E. A. Yildirim and S. J. Wright. Warm-start strategies in interior-point methods for linear programming. *SIAM J. Optimiz.*, 12(3):782–810, 2002.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.