# Scale-Model Architectural Simulation

Wenjie Liu* Wim Heirman† Stijn Eyerman† Shoaib Akram‡ Lieven Eeckhout*
*Ghent University, Belgium †Intel Corporation, Belgium ‡Australian National University
*{wenjie.liu, lieven.eeckhout}@ugent.be †{wim.heirman, stijn.eyerman}@intel.com ‡shoaib.akram@anu.edu.au

***Abstract*—Computer architects extensively use simulation to steer future processor research and development. Simulating large-scale multicore processors is extremely time-consuming and is sometimes impossible because of simulation infrastructure constraints and/or simulation host compute and memory limitations.**

**This paper proposes scale-model simulation, a novel methodology to predict large-scale multicore system performance. Scale-model simulation first constructs and simulates a scale model of the target system with reduced core count and shared resources. Target system performance is then predicted through machine-learning (ML) based extrapolation. Scale-model simulation predicts 32-core target system performance based on a single-core scale model with an average error of 8.0% and 15.8% for homogeneous and heterogeneous multiprogram workloads, respectively, while yielding a 28× simulation speedup.**

***Index Terms*—Architectural simulation, performance prediction, multi-core, machine learning, scale model**

## I. Introduction

Predicting performance for a future computer system is a challenging and critical problem. The traditional approach is to employ detailed architectural simulation. Unfortunately, simulation is extremely time-consuming. In addition, simulation infrastructures have their limitations and may not be able to simulate a future large-scale system because of excessive memory consumption, simulator infrastructure limitations, or insufficient compute capability and/or memory capacity in the simulation host system when simulating large numbers of cores. Researchers and practitioners employ a variety of techniques to tackle the simulation challenge. A widely used solution is sampled simulation [1], [2]. Unfortunately, this approach does not solve the simulation problem when it comes to simulating increasingly large target systems. In particular, we observe that simulating an 8-core, 16-core and 32-core target system using Sniper [3], a fast and state-of-the-art parallel multicore simulator, takes 8, 17 and 43 hours, respectively, on a powerful 36-core simulation host when running multiprogram SPEC CPU workloads with (only) one billion instructions per benchmark. The super-linear increase in simulation time and complexity as a function of system size is a major challenge for computer architects in academia and industry.

In this paper, we propose *scale-model simulation*, a novel paradigm to predict future system performance [4]. Scale-model simulation combines architectural simulation with machine learning to predict performance for large-scale systems based on detailed simulation of a scaled-down configuration of the target system, called the *scale model*. Scale-model simulation first simulates a scale model of the target system. Performance for the target system is then predicted through extrapolation. Scale models solve the two problems aforementioned: (1) scale models speed up the simulation of large-scale systems: scale models are small enough to simulate in reasonable amount of time while performance extrapolation is instantaneous; and (2) scale models make simulation feasible for large-scale systems that cannot be simulated on existing infrastructure because of limitations in memory and compute capacity.

Scale models are widely used in a variety of engineering disciplines, including civil engineering (e.g., construction, fluid dynamics), mechanical engineering (e.g., aerodynamics, engine design), construction (e.g., architectural design, city development), etc. The most familiar scale models are miniatures, i.e., scaled-down versions of an original object. A key property of a scale model is that it accurately maintains relationships between various important aspects, but not necessarily all aspects, of the original object. Scale models enable demonstrating or studying some behavior of the original object. To the best of our knowledge, scale models have not been applied to the field of general-purpose computer architecture. While building an exact miniature of a target system may be hard in the context of processor architectures, if at all possible, we leverage the idea of scale models to predict future computer system performance.

The scale-model simulation paradigm can be decomposed into two sub-objectives: (1) scale-model construction and (2) scale-model extrapolation. The first objective relates to how to construct a scale model of a (much) larger target multicore system, so that it takes substantially less time to simulate than the target system, yet enables an accurate prediction of the performance of the large-scale target system. A scale model is a scaled-down version of the target multicore system by featuring a reduced number of cores, say by a factor $F$, relative to the target system. The question is what to do with the shared resources, in particular the last-level cache (LLC), NoC and memory bandwidth. One option may be to not scale the shared resources. Assuming no shared resource contention, the performance of a single core in the scale model would be similar to the performance of an individual core in the target system. Of course, in reality, the actual performance will be less because of shared resource contention. We find for our suite of SPEC CPU2017 workloads, that not scaling shared resources leads to largely inaccurate scale models with an average 60% prediction error (and up to 94%) for a single-core scale model versus a 32-core target system. The alternative option is to proportionally scale the shared

resources. In particular, when scaling the number of cores by a factor $F$ in the scale model relative to the target system, the shared resources are also reduced proportionally by a factor $F$, i.e., LLC capacity, NoC bisection bandwidth and memory bandwidth are reduced by a factor $F$. We find that proportional resource scaling leads to substantially more accurate single-core scale models, with an average prediction error of 14.7% and at most 32.2% relative to a 32-core target system.

Because the scale model is not an exact miniature of the target system, the second objective relates to how to extrapolate performance from the scale model to the target system to further improve accuracy. Shared resources lead to a variety of complex interactions at the system level, which the scale models may or may not capture to a sufficient degree. Scale-model extrapolation predicts the impact of contention effects in shared resources on target-system performance based on the simulated scale model. We propose and evaluate two extrapolation methods that leverage Machine Learning (ML) to infer prediction models that predict target-system performance based on scale-model measurements. The two methods are ML-based prediction and ML-based regression. The key difference between both methods is that ML-based regression does not require simulation runs of the target system during training, in contrast to ML-based prediction. ML-based regression can thus be deployed when it is too time-consuming or even impossible to run simulations of the target system. We explore a variety of ML-based scale-model extrapolation techniques, including decision trees, random forest and support vector machines (SVM), and we find that SVM is most accurate. In addition, we evaluate a number of regression methods (linear, power and logarithmic), and find that logarithmic regression is most accurate. Our evaluation using multiprogram SPEC CPU2017 workloads demonstrates the high accuracy of scale-model simulation. Considering a single-core scale model and a 32-core target system, we report that for homogeneous multiprogram workload mixes, SVM-based prediction yields an average prediction error of 6.4% (20.8% max error). SVM-based regression is slightly less accurate as it does not involve target-system simulations during training. SVM-based regression yields an average prediction error of 8.0% (26.4% max error).

Scale-model simulation is more challenging for heterogeneous multiprogram workload mixes because of more diverse interaction and contention effects. Nevertheless, we demonstrate that scale-model simulation is also effective and accurate for heterogeneous workload mixes. We report that SVM-based prediction achieves an average prediction error of 13.2% (max error of 27.5%) for SVM-based prediction, and 15.8% for SVM-based regression (max error of 28.7%).

Scale-model simulation leads to substantial simulation speedups. Training the prediction model is a one-time cost that can be amortized across many predictions. Once the prediction model has been trained, scale-model simulation is fast. It only requires running a simulation of the application of interest on the single-core scale model, which is substantially faster than running a simulation of the target system, i.e., in our experimental setup in which we use Sniper [3] on a high-end 36-core Intel PowerEdge R440 server, we find that simulating a single-core scale-model is 28× faster than simulating the 32-core target system.

In summary, we make the following key contributions:

- We propose scale-model simulation, a novel methodology to predict target-system performance based on scale-model performance simulations.
- We find that shared resources are best proportionally scaled in the scale model relative to the target system.
- We demonstrate that extrapolation can significantly improve scale-model simulation accuracy.
- We propose and evaluate two ML-based extrapolation techniques that do or do not rely on target-system simulations during training.
- We evaluate scale-model simulation and demonstrate high accuracy and simulation speed improvements for both homogeneous and heterogeneous multiprogram workload mixes for a 32-core target system based on single-core scale-model simulations.
- We find that ML-based regression is almost equally accurate as ML-based prediction while not requiring target-system simulations during training, making ML-based regression a more practical approach.

## II. Scale Model Construction

Scale-model architectural simulation involves two key concerns: (1) how to construct the scale models, and (2) how to build an accurate extrapolation model based on the scale model predictions. We discuss the former in this section and the latter in the next section.

A scale model is a scaled-down version of the large-scale target system such that its performance is a (relatively) accurate representation of the target system. More precisely, the scale model needs to be configured such that its per-core performance is similar to per-core performance in the target system. The challenge when constructing scale models for multicore processors is how to deal with shared resources.

One option is to simply scale the number of cores in the scale model while keeping the shared resources as in the target system — we refer to this approach as *No Resource Scaling (NRS)*. For example, a scale model consisting of a single core would have access to the fully sized LLC capacity as well as the same NoC and memory bandwidth as in the target system.

Another, more accurate, option is to proportionally scale the shared resources with core count — we refer to this approach as *Proportional Resource Scaling (PRS)*. The intuition behind PRS is to provide balanced scale models that exhibit similar degrees of resource contention as in the target system. In particular, when scaling the number of cores by a factor $F$, we scale LLC capacity, NoC bandwidth and memory bandwidth by the same factor $F$. In other words, we keep LLC capacity per core constant and we keep interconnection and memory bandwidth per core constant. In our setup, we assume 1 MB of LLC per core, 4 GB/s NoC bisection bandwidth per core, and 4 GB/s memory bandwidth per core. See Table I for how

| #cores | LLC | NoC | DRAM |
|---|---|---|---|
| 32 | 32 MB: 32 slices | 128 GB/s: 4 CSLs, 32 GB/s per CSL | 128 GB/s: 8 MCs, 16 GB/s per MC |
| 16 | 16 MB: 16 slices | 64 GB/s: 4 CSLs, 16 GB/s per CSL | 64 GB/s: 4 MCs, 16 GB/s per MC |
| 8 | 8 MB: 8 slices | 32 GB/s: 2 CSLs, 16 GB/s per CSL | 32 GB/s: 2 MCs, 16 GB/s per MC |
| 4 | 4 MB: 4 slices | 16 GB/s: 2 CSLs, 8 GB/s per CSL | 16 GB/s: 1 MC, 16 GB/s per MC |
| 2 | 2 MB: 2 slices | 8 GB/s: 1 CSL, 8 GB/s per CSL | 8 GB/s: 1 MC, 8 GB/s per MC |
| 1 | 1 MB: 1 slice | 4 GB/s: 1 CSL, 4 GB/s per CSL | 4 GB/s: 1 MC, 4 GB/s per MC |

TABLE I: Constructing scale models through *Proportional Resource Scaling*: LLC capacity in MB; on-chip interconnection network in GB/s: number of cross-section links (CSLs) and bandwidth per CSL; main memory bandwidth in GB/s: number of memory controllers (MCs) and bandwidth per MC.

we scale shared resources in our setup. Since we assume a NUCA LLC with a 1 MB slice attached to each core in our setup, we proportionally scale down LLC capacity as we consider fewer cores in the scale model. Scaling bandwidth is more complicated. We scale DRAM bandwidth by changing both the number of memory controllers and bandwidth per memory controller. Starting from the target system, we first scale down the number of memory controllers from 8 (at 32 cores) to 1 (at 4 cores), and then scale down the amount of bandwidth per memory controller. First scaling the number of memory controllers and then scaling bandwidth per memory controller once there is only single memory controller left, enables more accurate scale models compared to first scaling memory bandwidth per memory controller and then scaling the number of memory controllers (as we will quantify in the evaluation section). For the interconnection network, we scale link bandwidth as the number of cross-section links reduces with core count. In particular, scaling down from 32 to 16 cores, the number of cross-section links remains unchanged, hence we have to halve bandwidth per link from 32 GB/s to 16 GB/s. In contrast, when moving from 16 to 8 cores, the number of cross-section links halves from 4 to 2, hence we maintain the per-link bandwidth at 16 GB/s.

## III. Scale Model Extrapolation

Scale model construction is only a first step. We need scale model extrapolation to yield even more accurate target system performance predictions. Scale-model extrapolation considers scale-model simulation results to predict target-system performance. We consider two extrapolation models in this work: no extrapolation and ML-based prediction and regression.

### A. No Extrapolation

The simplest way to predict target system performance is to use the per-core performance observed in the scale model as a prediction for per-core performance in the target system. This approach implicitly assumes that the interference observed in the shared resources in the scale model is similar to (or the same as in) the target system. The scale model that we assume is a single-core system with the shared resources proportionally scaled following the PRS approach. The performance measured for this single-core scale model then is the prediction for per-core performance in the target system.

While we primarily focus on a single-core scale model in this paper, it might be worth considering a two-core scale model or a four-core scale model (again, with the shared resources proportionally scaled). This typically leads to higher accuracy. On the flip side, simulating a scale model with more cores and larger shared resources takes longer. In other words, increasing the size of the scale model leads to an accuracy versus speed trade-off. The larger the scale model, the higher the accuracy but the longer simulation takes. While we will primarily focus on the results with a single-core scale model — as it yields the highest possible simulation speedup — we will also explore the accuracy versus simulation speed trade-off by considering larger scale models in the results section.

### B. Machine Learning-based Prediction and Regression

Leveraging Machine Learning (ML) enables achieving higher accuracy compared to the No Extrapolation method. We consider two ML-based approaches: *ML-based Prediction* and *ML-based Regression*. Both methods involve a training phase during which a performance model is trained. The training phase incurs a one-time cost. The key difference between both approaches is that ML-based Regression does not require simulation runs of the target system during training, in contrast to ML-based Prediction. This has important implications in practice. In case it is impossible to simulate the target system for some reason (too long simulation time or other simulator limitations), one has to resort to ML-based Regression. Higher accuracy is typically obtained through ML-based Prediction, although that requires access to the target system. We now explain both approaches.

*1) ML-Based Prediction:* ML-based prediction involves a training phase in which a set of training benchmarks are run on both the scale model and the target system, see also Figure 1. We consider $N$ benchmark mixes in our training set, with each mix $i$ ($1 \leq i \leq N$) consisting of $T$ benchmarks $B_j, 1 \leq j \leq T$. There are as many benchmarks per mix as there are cores in the target system, namely $T$. We denote a performance number $P$ obtained on the single-core scale model with superscript $ss$ ($P^{ss}$), on the multi-core scale models with superscript $msX$ ($P^{msX}$), and on the target system with superscript $t$ ($P^t$).

On the single-core scale model, we measure performance (i.e., IPC) and memory bandwidth utilization. The latter is a function of the number of LLC misses per unit of time and has a significant impact on resource contention in the

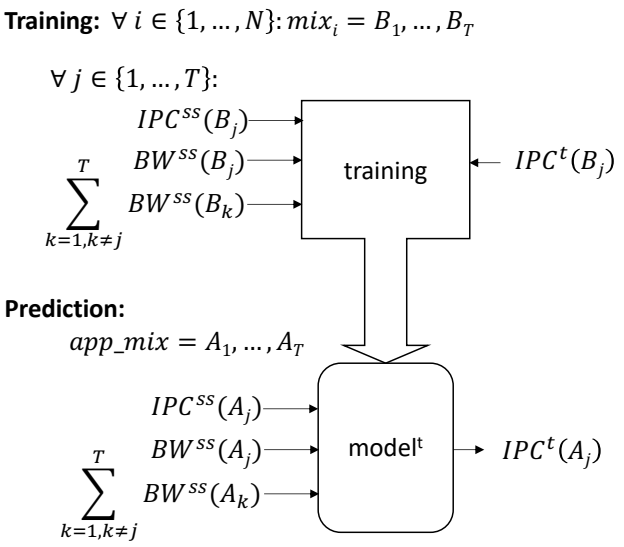


Fig. 1: ML-based prediction involves a training and prediction phase. *The training phase requires simulation results for the target system.*

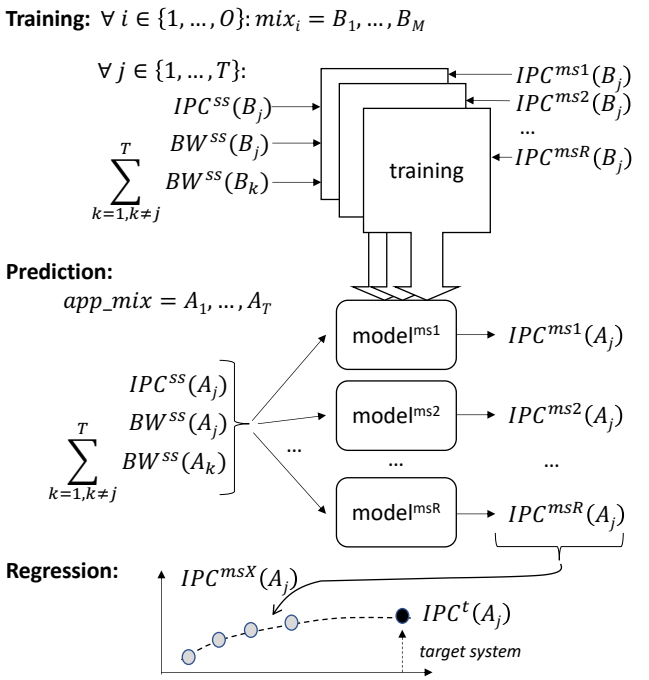


Fig. 2: ML-based regression involves a training, prediction and regression phase. *The training phase requires simulation results obtained for a number of multi-core scale models, but not the target system.*

memory subsystem during co-execution with other benchmarks. In other words, it provides a measure for how much contention the particular benchmark is going to create on the shared resources when co-executed with other benchmarks. Our results confirm that considering both performance and memory bandwidth utilization improves accuracy (as we will quantitatively demonstrate in the evaluation section). The performance and bandwidth utilization numbers on the single-core scale model serve as independent variables to the ML technique. More precisely, the input variables to the ML model are per-core performance for each of the benchmarks in the training mix ($IPC^{ss}(B_j)$), alongside the per-core bandwidth utilization for the given benchmark ($BW^{ss}(B_j)$) as well as the sum of the per-core bandwidth utilization numbers for the co-running applications in the workload mix ($\sum_{k=1,k\neq j}^{T} BW^{ss}(B_k)$). On the target system, we measure performance for each of the benchmarks in the multi-program workload mix ($IPC^{t}(B_j)$). Target system performance for each of the benchmarks in the training mix serves as dependent variables to the ML technique. Overall, the input to the ML training phase consists of $N \times T$ data points as there are $N$ mixes and $T$ benchmarks per mix. The end result of the training phase is a performance model, denoted as $model^t$, that predicts target-system performance of an application when co-run with $T-1$ other applications.

The prediction, or inference, phase involves simulating a previously unseen application $A_j$ (i.e., the workload of interest) on the single-core scale model. The measured performance and bandwidth utilization numbers serve as input to the prediction model which then yields a prediction for performance of the application of interest on the target system. More specifically, the prediction model takes the IPC of the application of interest on the single-core scale model as input ($IPC^{ss}(A_j)$), alongside its bandwidth utilization on the scale model ($BW^{ss}(A_j)$) as well as the total bandwidth consumption of the co-running applications in the workload mix. The latter is computed as the sum of the bandwidth consumption for each of the applications in the workload mix as observed in the single-core scale model, i.e., $\sum_{k=1,k\neq j}^{T} BW^{ss}(A_k)$. The model $model^t$ then predicts performance for application $A_j$ on the target system.

We consider different ML techniques in this work to construct the prediction model, namely decision tree (DT), random forest (RF) and support vector machines (SVM) using the scikit-learn v1.0.1 framework.[1] The DT algorithm is an optimized version of the CART (Classification and Regression Trees) algorithm which constructs binary trees by seeking for the largest information gain at each node using Iterative Dichotomiser. RF includes a diverse set of decision trees to avoid overfitting; this is done through two levels of randomization. First, each tree in the ensemble is built for a random subset from the training set. Second, when splitting a node during the construction of a tree, the best split is found for either all input features or for a random subset of features. We use the radial basis function (RBF) as the SVM kernel to capture non-linear performance scaling trends.

*2) ML-Based Regression:* As mentioned above, ML-based prediction requires simulation runs of the target system during

[1] `http://scikit-learn.org/`

training, which may be a significant impediment in practice. ML-based regression overcomes this drawback by relying on simulation runs of a variety of scale models instead, which is typically easier to achieve in practice. ML-based regression consists of three steps, see also Figure 2. In the first step, ML-based regression leverages the ML-based prediction method discussed above to train a number of prediction models. These prediction models do not predict performance for the target system, as under ML-based prediction, but they predict performance for a number of multi-core scale models $ms1, ms2, \ldots, msR$. Note that these scale models feature multiple cores. The training phase involves measuring performance and bandwidth utilization on the single-core scale model, and measuring performance for the multi-core scale models for each of the benchmarks in the training workload mixes. The input to the training phase thus includes, as independent variables, the performance ($IPC^{ss}(B_j)$) and bandwidth utilization ($BW^{ss}(B_j)$) of each benchmark in the mix alongside the aggregate bandwidth utilization of the co-running benchmarks ($\sum_{k=1,k\neq j}^{T} BW^{ss}(B_k)$). The dependent variables are the performance numbers for each benchmark for the scale models $ms1, ms2, \ldots, msR$, or $IPC^{ms1}(B_j), IPC^{ms2}(B_j), \ldots, IPC^{msR}(B_j)$. The ML-based prediction method is used to train the prediction models for the various multi-core scale models.

As a second step, once these prediction models have been trained, we predict performance for a previously unseen application of interest $A_j$ on the multi-core scale models $ms1, ms2, \ldots, msR$. The input to the models includes the application's scale-model performance ($IPC^{ss}(A_j)$), its bandwidth utilization ($BW^{ss}(A_j)$) and the aggregate bandwidth utilization of the co-runners ($\sum_{k=1,k\neq j}^{T} BW^{ss}(A_k)$). The models then predict performance for application $A_j$ on the scale models, namely $IPC^{ms1}(A_j), IPC^{ms2}(A_j), \ldots, IPC^{msR}(A_j)$.

The third step involves regression to predict performance for the target system based on the predicted performance numbers for the multi-core scale models. We consider a number of regression techniques, including linear, power-law and logarithmic regression, to predict target-system performance. We find that logarithmic regression yields the highest accuracy (a quantitative evaluation is reported in the evaluation section).

## IV. Experimental Setup

*1) Simulation Setup:* We use Sniper v6.0, a parallel and high-speed cycle-level x86 simulator for multicore systems, using its most detailed cycle-level hardware-validated core model [3]. Our target system is a 32-core processor, see Table II. We simulate 4-wide out-of-order cores with a 3-level cache hierarchy. The LLC is a 32 MB NUCA cache, and we assume a 128 GB/s bisection bandwidth mesh NoC and 128 GB/s main memory system with 8 memory controllers.

Our simulation speed numbers are obtained by running Sniper on a 36-core Intel PowerEdge R440 server. This server is dual-socket machine with 18 cores per socket, 24 MB LLC per socket, 384 GB of memory.

| | |
|---|---|
| **Processor** | |
| Number of cores | 32 cores |
| Core frequency | 4.0 GHz |
| Issue width | 4-wide |
| ROB size | 128 entries |
| Branch predictor | hybrid local/global predictor |
| Max. outstanding | 48 loads, 32 stores, 10 L1-D misses |
| **Cache Hierarchy** | |
| L1-I | 32 KB, 4 way, 4 cycle access time |
| L1-D | 32 KB, 8 way, 4 cycle access time |
| L2 | 256 KB per core, 8 way, 8 cycle |
| LLC | shared 32 MB, 64 way, 30 cycle<br>NUCA, 32 slices, 1 MB/slice, 1 slice/core |
| **NoC** | |
| Mesh topology | 4×8 |
| Bandwidth | 128 GB/s bisection bandwidth |
| **DRAM** | |
| Memory controllers | 8 |
| Bandwidth | 128 GB/s aggregate bandwidth |

TABLE II: Target system.

*2) Workloads:* We consider both homogeneous and heterogeneous multiprogram workload mixes in the evaluation. The benchmarks are taken from SPEC CPU2017 and we consider 1B-instruction simulation points per benchmark as identified by the SimPoint methodology [1]. The homogeneous workloads assume co-running instances of the same benchmark, all starting at (slightly) different offsets. The heterogeneous workload mixes are randomly composed. We finish the simulation and measure performance when the first benchmark in the workload mix has reached the end of its simulation point.

We make sure that the training set is completely disjoint from the evaluation set in all of our experiments. For the homogeneous workload mixes, we use a cross-validation setup in which we use $N-1$ benchmarks for training the models when evaluating prediction accuracy for the $N$th benchmark, with $N = 29$ for SPEC CPU2017. There are hence 28 training benchmarks to train a model to predict performance for the 29th benchmark. We use the prediction and extrapolation models to predict performance for the previously unseen application of interest on the target system when co-run with additional $(T-1)$ copies of the application of interest.

For the heterogeneous workload mixes, we consider 8 randomly chosen benchmarks in the evaluation set while using the 21 remaining benchmarks in the training set. The reason for limiting the evaluation to 8 benchmarks is to limit overall simulation time for training. The number of training mixes is chosen such that the total amount of training data is constant, i.e., we consider a total of 320 training results to train an ML model. For ML-based prediction, this means that we consider $N = 10$ training mixes with $T = 32$ benchmarks each, yielding a total of $N \times T = 320$ training results. For ML-based regression, when training an ML model for an $M$-core scale model, we consider $O$ training mixes, so that we have a total of $O \times M = 320$ training mixes. In particular, when training

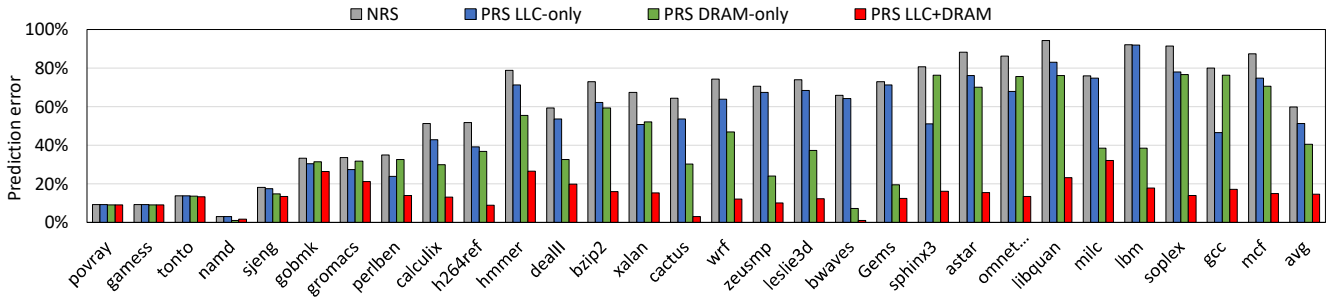


Fig. 3: Evaluating scale model construction using homogeneous workload mixes: NRS versus PRS with scaled LLC capacity, scaled DRAM bandwidth, and both. *Proportional Resource Scaling (PRS) in which all shared resources are scaled proportionally leads to the most accurate scale models.*

a model for a two-core scale model, we consider 160 training mixes, yielding 320 training results; when training a model for a quad-core scale model, we consider 80 training mixes, again yielding 320 training results; etc. Prediction is done for a previously unseen application of interest which we simulate on the single-core scale model. We predict performance for the application of interest on the target system when co-run with 10 random heterogeneous mixes of previously unseen applications from the evaluation set; we report the average prediction error across these 10 mixes for each of application of interest.

## V. Evaluation

We now evaluate scale model simulation. We first evaluate scale-model construction, and then evaluate scale-model extrapolation. We quantify accuracy using the following absolute prediction error metric: $error = \left|\frac{IPC_{predicted} - IPC_{actual}}{IPC_{actual}}\right|$. $IPC_{actual}$ is the IPC of the application of interest on the target system — in our setup, this is the IPC of a single benchmark instance in a 32-instance multi-program workload. $IPC_{predicted}$ is the predicted IPC of the application of interest on the target system based on measurements obtained through simulation of the scale model. In case of No Extrapolation, the predicted IPC on the target system *is* the IPC obtained on the scale model. In case of ML-based Prediction and Extrapolation, the predicted IPC is provided by the ML model when given the performance metrics for the scale model as input. We assume a single-core scale model in all of our experiments unless mentioned otherwise.

### A. Scale Model Construction

We consider the following four scale-model construction techniques: (1) No Resource Scaling (NRS), i.e., the shared resources in the scale model are sized identically to the target system, (2) Proportional Resource Scaling (PRS) in which we only scale the LLC in the scale model (i.e., DRAM bandwidth in the scale model is the same as in the target system), (3) PRS with scaled DRAM bandwidth only (i.e., LLC capacity is the same in the scale model and target system), and (4) PRS with scaled LLC size *and* DRAM bandwidth. (We evaluated NoC scaling as well but found it to have (virtually) no effect for the workloads considered in this work, hence we exclude it from the discussion.)

Figure 3 reports prediction error for the single-core scale model, i.e., we consider a scale model with a single core to predict per-core performance in the 32-core target system. The benchmarks are sorted by their LLC MPKI from left to right. The benchmarks on the left-hand side are thus compute-intensive for which NRS and PRS perform equally well. However, memory-intensive benchmarks appearing on the right-hand side experience contention in the shared resources and hence require that the scale models feature proportionally scaled-down shared resources. Overall, NRS is generally inaccurate with an average absolute error of 60% and up to 94%. PRS is more accurate, especially for memory-intensive workloads: scaling the LLC brings down the average error to 51.3%, while scaling DRAM bandwidth reduces the average error to 40.5%. Scaling both LLC capacity and DRAM bandwidth has synergistic effects, bringing down the prediction error to 14.7% on average and at most 32.2% (milc). Proportionally scaling all shared resources leads to a scale model that is a relatively accurate representation for per-core performance in the target system.

### B. Scale Model Extrapolation

While PRS leads to relatively accurate scale models, we can do even better through scale model extrapolation. No Extrapolation uses performance obtained for the scale model as a prediction for per-core performance in the target system — this is effectively PRS with scaled resources from the previous section. We further consider ML-based Prediction and ML-based Regression; we consider three ML techniques — Decision Tree (DT), Random Forest (RF) and Support Vector Machines (SVM) — and we use logarithmic regression for the Regression approach.

Figure 4 reports the prediction error for these techniques assuming homogeneous workload mixes. ML-based Prediction brings down the average absolute prediction error by a significant margin compared to No Extrapolation (average error of 14.7% and up to 32.2%). SVM is the most accurate ML-based Prediction technique with an average error of 6.4% (maximum error of 20.8%). DT yields an average absolute prediction error of 9.3% (and up to 29.1%), whereas RF leads to an average

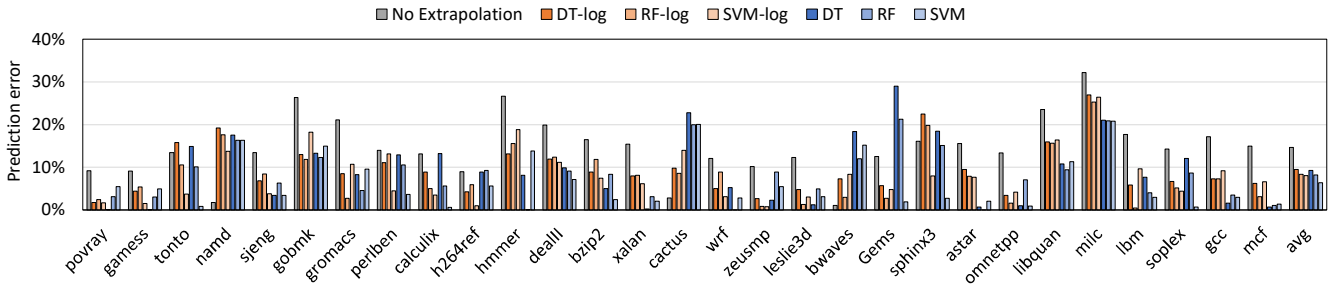


Fig. 4: Evaluating scale model extrapolation using homogeneous workload mixes: No Extrapolation versus ML-based Prediction and Regression. *SVM-based prediction yields the highest accuracy (6.4% average absolute prediction error), while SVM-based regression (SVM-log) is only slightly less accurate (8.0% average absolute prediction error).*

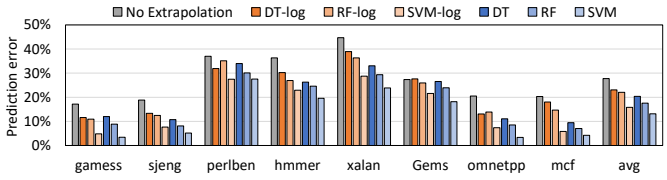


Fig. 5: Evaluating scale model extrapolation using heterogeneous workload mixes. *The SVM-based Prediction method yields the highest accuracy (13.2% average prediction error), while SVM-based Regression (SVM-log) is only slightly less accurate (15.8% average prediction error).*

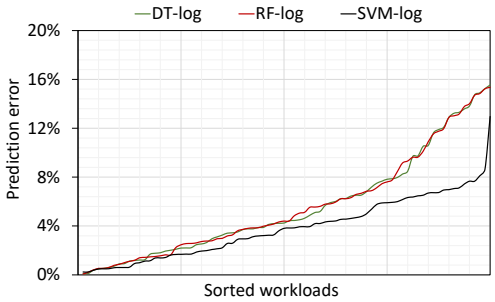


Fig. 6: STP prediction error for ML-based regression across a total of 80 heterogeneous workload mixes. *SVM-log predicts system throughput (STP) with an average prediction error of 3.8% and at most 13.0%.*

error of 8.3% (and up to 21.3%). ML-based Regression is slightly less accurate than ML-based Prediction as it does not require simulating the target system during training. Yet, accuracy is still high and SVM with logarithmic regression (SVM-log) yields the highest accuracy among the ML-based Regression techniques with an average absolute prediction error of 8.0% (and at most 26.4%). While ML-based prediction outperforms ML-based prediction in general, the inverse is true for some benchmarks. This is the case when performance across scale models (with 2, 4, 8 and 16 cores) follows a predictive trend line — favoring regression. If on the other hand, the relative performance delta between the one-core scale model and the 32-core target system is relatively easy to predict, i.e., the relative delta is fairly similar to previously seen training examples, then prediction is most accurate.

### C. Heterogeneous Workload Mixes

So far, we considered homogeneous workload mixes. Figure 5 reports prediction error for the various prediction techniques under heterogeneous workload mixes. The results are consistent with the homogeneous workload mixes, i.e., ML-based Prediction is slightly more accurate than ML-based Regression, and SVM is the most accurate ML approach. We do note higher prediction errors for the heterogeneous workload mixes compared to the homogeneous workload mixes due to more complex and diverse interactions between co-running applications: average prediction error of 15.8% (max 28.7%) for SVM-log versus 13.2% (max 27.5%) for SVM, versus 27.8% (max 44.7%) for No Extrapolation.

These per-application performance predictions can be used to predict system throughput (STP) on the 32-core target system. STP is computed as the sum of normalized IPC values (IPC on the target system divided by IPC on the single-core scale model) across all applications in the workload mix [5]. Figure 6 reports the STP prediction error (sorted) for ML-based regression for a total of 80 heterogeneous mixes. SVM-log is the most accurate regression approach with an average error of 3.8% versus 5.6% for DT-log and RF-log. Interestingly, the STP prediction errors are lower than the per-application prediction errors reported above. The reason is that STP is computed as the sum of normalized IPC values, hence over- and underestimations offset each other.

### D. Simulation Speedup

Scale-model simulation yields a substantial simulation speedup because simulating a scale model takes considerably less time than simulating the target system. And in some cases, it might not even be possible to simulate the target system, due to simulator infrastructure and/or simulation host constraints. Once scale-model simulation results are available, predicting target-system performance is almost instantaneous provided that the ML model has been trained offline.

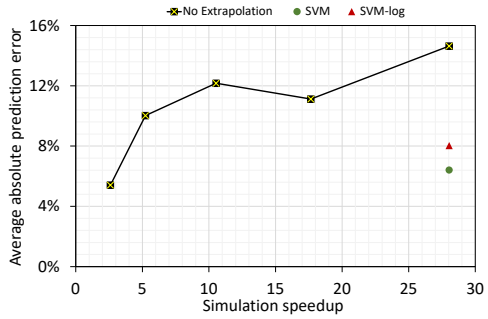


Fig. 7: Prediction error versus simulation speedup. *SVM-based prediction and regression achieve high prediction accuracy while yielding high simulation speedups.*

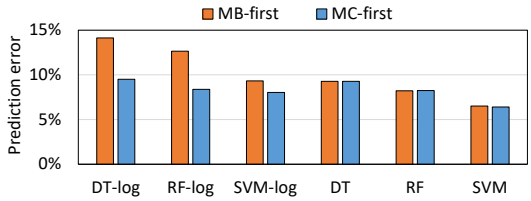


Fig. 8: Evaluating memory bandwidth scaling alternatives under PRS. *ML-based regression achieves higher accuracy by first scaling the number of memory controllers ('MC-first') compared to first scaling memory bandwidth per memory controller ('MB-first').*

Figure 7 reports prediction error versus simulation speedup compared to simulating the 32-core target system. The No Extrapolation curve consists of 5 data points. The data point on the far right refers to the case where the scale model is a single-core system. Moving to the left, we have a dual-core, quad-core, octo-core and finally a 16-core scale model. Prediction accuracy generally improves as we move towards larger scale models[2], while simulation speedup decreases considerably. The ML-based prediction techniques, SVM and SVM-log, rely on a single-core scale model simulation only, and hence yield the highest possible simulation speedup, namely 28×. Overall, the conclusion is that ML-based prediction and regression is accurate while yielding high simulation speedups.

### E. Sensitivity Analyses

We now perform a couple analyses to evaluate the sensitivity of the proposed scale-model simulation methodology. We consider the homogeneous workload mixes throughout.

*1) Memory bandwidth scaling:* Recall from Section II that we explored two options for how to proportionally scale down memory bandwidth from the target system to the scale model. One option ('MC-first', our default) is to first scale the number of memory controllers (while keeping memory bandwidth per memory controller constant) and then scale memory bandwidth per memory controller when there is only single memory controller left. An alternative option ('MB-first') is to first scale down memory bandwidth per memory controller from 16 to 4 GB/s while keeping the number of memory controllers constant, and then scale down the number of memory controllers from 8 to 1. Figure 8 reports prediction error for the various scale models under MC-first and MB-first. We find that first scaling the number of memory controllers yields the highest accuracy, especially for the ML-based regression techniques. In particular, for SVM-log, the average prediction error reduces from 9.3% to 8.0%; the improvement in accuracy is even more substantial for DT-log: reduction in average prediction error from 14.1% to 9.5%.

[2]The dual-core scale model is more accurate than the quad-core scale model due to how memory bandwidth is scaled down, see also Table I. Both the 'MC-first' and 'MB-first' scaling methods (discussed in Section V-E1) lead to similar trend anomalies, albeit at different core counts.

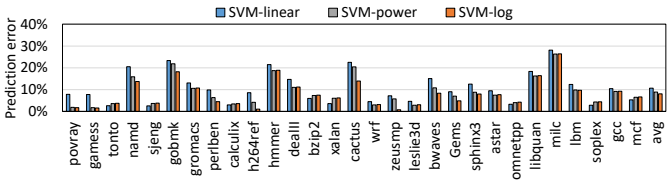


Fig. 9: Linear, power and logarithmic regression under SVM. *Logarithmic regression yields the lowest prediction error.*

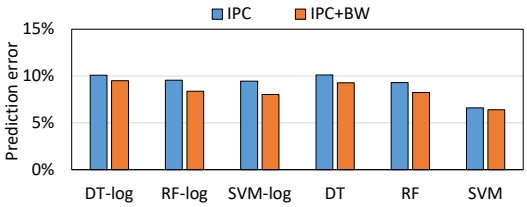


Fig. 10: Varying the input variables to the ML-based extrapolation techniques. *Considering both performance and bandwidth utilization as input variables leads to improved accuracy compared to using only performance as input.*

*2) Regression:* As aforementioned in Section III-B2, we evaluated three regression approaches following a linear model ($y = a \cdot x + b$), a power model ($y = a \cdot x^b$) and a logarithmic model ($y = a \cdot ln(x) + b$), in which $x$ is the number of cores and $y$ is performance. We use least squares regression to obtain the parameters $a$ and $b$ that yield the best fitting curve. Figure 9 reports the accuracy for these three regression techniques under SVM-based regression. Logarithmic regression outperforms the power and linear models by a significant margin for most of the benchmarks, and leads to the lowest average prediction error: 10.7% (linear), 8.9% (power) and 8.0% (logarithmic).

*3) ML model inputs:* The proposed scale-model simulation methodology uses performance (IPC) and bandwidth utilization as input to the ML models, see Section III. Figure 10 reports the average prediction error for the different ML-based prediction and regression methods when comparing using both performance and bandwidth utilization as input versus using performance only. Using both performance and bandwidth uti-

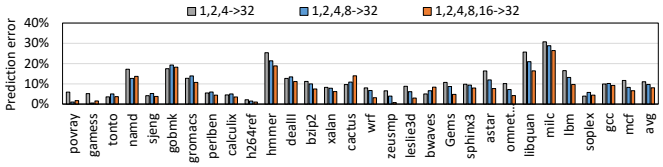


Fig. 11: Prediction error as a function of the number of multi-core scale models used for SVM-log regression. *The prediction error only slightly increases with a reduced number of multi-core scale models.*

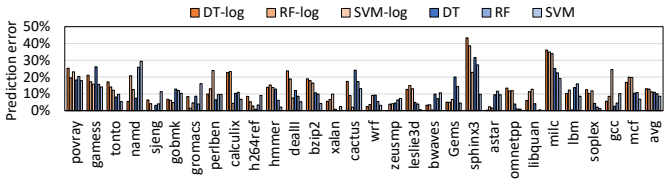


Fig. 12: Prediction error for predicting memory bandwidth utilization. *SVM and SVM-log predict memory bandwidth utilization with an average error of 8.7% and 11.3%, respectively.*

lization improves the prediction error by a significant margin compared to using only performance. In particular, for SVM-log, the average prediction error reduces from 9.5% to 8.0%.

*4) Multi-core scale-models under regression:* As discussed in Section III-B2, the ML-based regression techniques use a number of multi-core scale models to drive the regression. So far, we assumed four multi-core scale models with 2, 4, 8 and 16 cores. Figure 11 reports the prediction error when changing the number of multi-core scale models to 2 (dual- and quad-core scale models), 3 (dual-, quad- and octo-core scale models) and 4 (our default). Reducing the number of multi-core scale models might be of interest if the goal is to reduce model training time. Remarkably, the error is only slightly higher when limiting the number of multi-core scale models. The average prediction error equals 11.0% (2 and 4-core scale models) to 9.7% (2, 4 and 8-core scale models) to 8.0% (2, 4, 8 and 16-core scale models).

*5) Memory bandwidth utilization:* We focused on predicting performance throughout the result section. Scale-model simulation can also be used to predict other metrics, such as bandwidth utilization. This is done by considering bandwidth utilization (rather than performance) as the dependent variable when training the ML models, see also Section III. Figure 12 reports the prediction error for predicting memory bandwidth utilization. The result is in line with the previously reported accuracy numbers: SVM is the most accurate prediction approach (8.7% average error) and SVM-log is the most accurate regression approach (11.3% average error).

*6) Multi-threaded workloads:* While the evaluation in this work is limited to multi-program workloads consisting of single-threaded applications, applying and extending the proposed methodology to multi-threaded workloads is left for future work. We believe that scale-model simulation as proposed in this work might be easily applied to data-parallel multi-threaded workloads in which all threads execute the same code (on different data elements) and there is very little or no communication between threads. We expect scale-model simulation to perform similarly for such workloads as for the homogeneous workload mixes considered in this work. Applying scale-model simulation to multi-threaded workloads with inter-thread communication and synchronization (e.g., critical sections) requires additional research. We believe that speedup stacks [6], [7] provide an opportunity to develop a scale-model simulation methodology for general multi-threaded workloads. A speedup stack quantifies how various bottlenecks (i.e., synchronization, coherence, imbalance, on-chip network and memory bandwidth congestion, etc.) scale with system size. By quantifying how these individual bottlenecks scale (or do not scale) across a range of scale models, we might be able to make a prediction for multi-threaded application performance on the target system. This is left as part of future work.

## VI. Related Work

The most closely related work by Eyerman et al. [8] proposes scale models for an experimental Intel processor, called PIUMA (Programmable Integrated Unified Memory Architecture), that is specifically designed for the efficient execution of graph analytics workloads. The lack of resource sharing among processor cores makes the development of scale models for this type of architecture relatively easy. More specifically, the PIUMA architecture does not have shared caches; each core has a dedicated memory controller; and a highly scalable interconnection network provides high bandwidth and low latency to each individual core. In contrast, the cores in a general-purpose multi-core processor share the LLC, NoC and memory subsystem.

Machine learning (e.g., neural networks [9] and spline-based regression [10]) was previously proposed to explore single-core and multi-core design spaces, however, predicting performance for larger-scale target systems fell out of reach for these models. Analytical models have been proposed for multi-core processors for both multiprogram workloads [11], [12] and multi-threaded workloads [13]. An inherent challenge for such models is how to analytically model overlap effects as well as timing-sensitive events in large target systems; scale-model simulation addresses this challenge through extrapolation. Hoste et al. [14] and Piccart et al. [15] determine the optimum platform among a set of previously benchmarked platforms for an application of interest. Other prior work predicts performance across architecture paradigms: Baldini et al. [16] and Ardalani et al. [17] propose machine-learning based methodologies to predict GPU performance based on CPU implementations.

Scaling down the workloads to speed up simulation has received considerable attention in the literature. Sampling is a widely used methodology to select representative regions of execution of an unchanged workload. Prior work has proposed sampling for single-threaded workloads [1], [2] as well as

for barrier-synchronized workloads [18] and general multi-threaded workloads [19], [20]. Alameldeen et al. [21] propose a methodology for scaling down commercial workloads in both size and runtime, allowing commodity machines to simulate much more powerful server systems.

Raising the level of abstraction is yet another, complementary, way to speed up simulation. One-IPC models assume that a single instruction is executed per cycle in the absence of miss events such as cache misses and branch mispredictions [22], [23]. Interval simulation [24] models the impact of miss events on performance through mechanistic analytical modeling. ZSim [25] and Sniper [26] implement high-abstraction simulation models for superscalar processors. While these high-abstraction models significantly speed up simulation, they do not fundamentally solve the simulation challenge of large-scale target systems.

Estimating the impact of interference in shared resources in multicore processors has been an active field of research, see for example [27]–[30]. Unfortunately, this prior work does not predict system performance for a target system that is substantially larger than the scale model(s).

## VII. Conclusion and Future Work

This paper proposed scale-model simulation, a novel methodology that combines architectural simulation of scale models with machine learning to predict the performance of a larger-scale target system. We provide results that demonstrate the effectiveness of scale-model simulation using both homogeneous and heterogeneous multiprogram workload mixes to predict 32-core target system performance based on single-core scale model simulation runs. We find that it is critical to proportionally scale the shared resources when constructing scale models. Leveraging ML techniques to construct extrapolation models further improves scale-model simulation accuracy. We find that ML-based regression, which does not rely on target-system simulations during training, achieves an average prediction error of 8% for homogeneous multiprogram workload mixes and 15.8% for heterogeneous mixes. Because scale-model simulation makes these predictions based on single-core scale model simulations, scale-model simulation leads to a 28$\times$ simulation speedup compared to simulating a 32-core target system using Sniper on a high-end 36-core simulation host system.

We believe that the idea of scale-model simulation opens up a range of opportunities for future work. Extending and evaluating scale-model simulation for multi-threaded workloads requires extending the methodology to account for NoC, coherence and synchronization effects. Scale-model simulation could also be explored and applied to other architecture paradigms including throughput processors such as GPUs. Scale-model simulation might also have a potential use case for steering procurement and purchasing decisions: scale-model simulation could be used to provide performance predictions for next-generation processors and systems.

## VIII. Acknowledgments

We thank the anonymous reviewers for the valuable feedback. Wenjie Liu is supported through a CSC scholarship. Additional support is provided through the European Research Council (ERC) Advanced Grant agreement No. 741097, and Research Foundation Flanders (FWO) grant No. G018722N.

## References

[1] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002, p. 45–57.

[2] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003, pp. 84–97.

[3] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, pp. 1–25, 2014.

[4] W. Liu, W. Heirman, S. Eyerman, S. Akram, and L. Eeckhout, "Scale-model simulation," *IEEE Computer Architecture Letters (CAL)*, vol. 20, no. 2, pp. 175–178, 2021.

[5] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[6] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2012, pp. 145–155.

[7] J. B. Sartor, K. Du Bois, S. Eyerman, and L. Eeckhout, "Analyzing the scalability of managed language applications with speedup stacks," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 23–32.

[8] S. Eyerman, W. Heirman, Y. Demir, K. Du Bois, and I. Hur, "Projecting performance for PIUMA using down-scaled simulation," in *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–7.

[9] E. Ipek, S. McKee, R. Caruana, d. B. R., and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, p. 195–206.

[10] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of 12th the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 185–194.

[11] K. Van Craeynest and L. Eeckhout, "The multi-program performance model: debunking current practice in multi-core simulation," in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 26–37.

[12] R. Jongerius, A. Anghel, G. Dittmann, G. Mariani, E. Vermij, and H. Corporaal, "Analytic multi-core processor model for fast design-space exploration," *IEEE Transactions on Computers (TC)*, vol. 67, no. 6, pp. 755–770, 2018.

[13] S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout, "RPPM: Rapid performance prediction of multithreaded workloads on multicore processors," in *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 257–267.

[14] 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 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006, pp. 114–122.

[15] B. Piccart, A. Georges, H. Blockeel, and L. Eeckhout, "Ranking commercial machines through data transposition," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2011, pp. 3–14.

[16] I. Baldini, S. J. Fink, and E. Altman, "Predicting GPU performance from CPU runs using machine learning," in *Proceedings of International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2014, pp. 254–261.

[17] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015, pp. 725–737.

[18] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout, "Barrierpoint: Sampled simulation of multi-threaded applications," in *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 2–12.

[19] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 2–12.

[20] A. Sabu, H. Patil, W. Heirman, and T. E. Carlson, "LoopPoint: Checkpoint-driven sampled simulation for multi-threaded applications," in *Proceedings of the 28th International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1–15.

[21] A. R. Alameldeen, M. M. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin, "Simulating a $2 M Commercial Server on a $2 K PC," *Computer*, vol. 36, no. 2, pp. 50–57, 2003.

[22] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "CMP$im: A pin-based on-the-fly multi-core cache simulator," in *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, 2008, pp. 28–36.

[23] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, 2010, pp. 1–12.

[24] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, 2010, pp. 1–12.

[25] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013, pp. 475–486.

[26] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–12.

[27] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 248–259.

[28] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, p. 77–88.

[29] K. Du Bois, S. Eyerman, and L. Eeckhout, "Per-thread cycle accounting in multicore processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–22, 2013.

[30] M. Jahre and L. Eeckhout, "GDP: Using dataflow properties to accurately estimate interference-free performance at runtime," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 296–309.