Efficiently Exploring Architectural Design Spaces via Predictive Modeling

Engin İpek Sally A. McKee
Cornell University
{engin,sam}@csl.cornell.edu

Bronis R. de Supinski
Martin Schulz
Lawrence Livermore National Laboratory
{bronis,schulzm}@llnl.gov

Rich Caruana
Cornell University
caruana@cs.cornell.edu

Abstract

Architects use cycle-by-cycle simulation to evaluate design choices and understand tradeoffs and interactions among design parameters. Efficiently exploring exponential-size design spaces with many interacting parameters remains an open problem: the sheer number of experiments renders detailed simulation intractable. We attack this problem via an automated approach that builds accurate, confident predictive design-space models. We simulate sampled points, using the results to teach our models the function describing relationships among design parameters. The models produce highly accurate performance estimates for other points in the space, can be queried to predict performance impacts of architectural changes, and are very fast compared to simulation, enabling efficient discovery of tradeoffs among parameters in different regions. We validate our approach via sensitivity studies on memory hierarchy and CPU design spaces: our models generally predict IPC with only 1-2% error and reduce required simulation by two orders of magnitude. We also show the efficacy of our technique for exploring chip multiprocessor (CMP) design spaces: when trained on a 1% sample drawn from a CMP design space with 250K points and up to 55× performance swings among different system configurations, our models predict performance with only 4-5% error on average. Our approach combines with techniques to reduce time per simulation, achieving net time savings of three-four orders of magnitude.

Categories and Subject Descriptors I.6.5 Computing Methodologies [*Simulation and Modeling*]: Model Development; B.8.2 Hardware [*Performance and Reliability*]: Performance Analysis and Design Aids

General Terms Design, Experimentation, Measurement

Keywords design space exploration, sensitivity studies, artificial neural networks, performance prediction

1. Introduction

Architects quantify the impact of design parameters on evaluation metrics to understand tradeoffs and interactions among those parameters. Such analyses usually employ cycle-by-cycle simulation of a target machine either to predict performance impacts of architectural changes, or to find promising design subspaces satisfying different performance/cost/complexity/power constraints. Several factors have unacceptably increased the time and resources required for the latter task, including the desire to model more realistic workloads, the increasing complexity of modeled architectures, and the exponential design spaces spanned by many independent parameters. Thorough study of even relatively modest design spaces becomes challenging, if not infeasible [22, 16, 5].

Nonetheless, sensitivity studies of large design spaces are often essential to making good choices: for instance, Kumar et al. [20] find that design decisions not accounting for interactions with the interconnect in a CMP are often opposite to those indicated when such factors are considered. Research on reducing time per experiment or identifying the most important subspaces to explore within a full parameter space has significantly improved our ability to conduct more thorough studies. Even so, simulation times for thorough design space exploration remain intractable for most researchers.

We attack this problem by using artificial neural networks (ANNs) to predict performance for most points in the design space. We view the simulator as a nonlinear function of its M-parameter configuration: $SIM(p_0, p_1, ...p_M)$. Instead of sampling this function at every point (parameter vector) of interest, we employ nonlinear regression to approximate it. We repeatedly sample small numbers of points in the design space, simulate them, and use the results to teach the ANNs to approximate the function. At each teaching (training) step, we obtain highly accurate error estimates of our approximation for the full space. We continue refining the approximation by training the ANNs on further sample points until error estimates drop sufficiently low.

By training the ANNs on 1-2% of a design space, we predict results for other design points with 98-99% accuracy. The ANNs are extremely fast compared to simulation (training typically takes few minutes), and our approach is fully automated. Combining our models with SimPoint [34] reduces required CPU time by three-four orders of magnitude, enabling detailed study of architectural design spaces previously beyond the reach of current simulation technology. This allows the architect to purge most of the uninteresting design points quickly and focus detailed simulation on promising design regions.

Most importantly, our approach fundamentally differs from heuristic search algorithms in scope and use. It can certainly be used for optimization (predicted optimum with 1% sampling is within 3% of global optimum performance for applications in our processor and memory system studies), but we provide a superset of the capabilities of heuristics that intelligently search design spaces to optimize an objective function (e.g., those studied by Eyerman et al. [10]). Specifically, our technique:

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

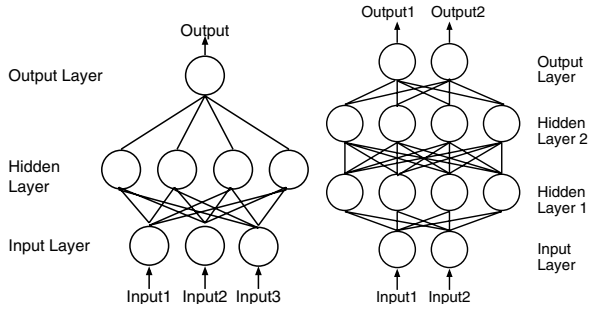

Figure 1. Simplified diagrams of fully connected, feed-forward ANN

- generates accurate predictions *for all points in the design space*. Unlike heuristic search techniques, our models can be queried to predict performance impacts of architectural changes, enabling efficient discovery of tradeoffs among parameters in different regions.
- provides highly accurate (typically within 1-2%) error estimates. These increase confidence in results, and provide a well crafted knob for the architect to control the accuracy-speed tradeoff inherent in architectural modeling.
- verifies that apparent performance gains from a novel proposal are not mere artifacts of other parameters chosen.
- allows architects to observe the sensitivity of a proposal to interacting parameters. This allows more thorough evaluation, increasing confidence in novel architectural ideas.

After training on 2% of our design spaces, querying our models to identify design points within >10% of the predicted optimal IPC purges over 80% of design points. Querying again to identify points within a given power budget, for instance, could eliminate comparable portions of remaining subspaces. Inspection of these spaces can then provide insight to guide subsequent design decisions.

2. ANN Modeling of Design Spaces

Artificial neural networks (ANNs) are machine learning models that automatically learn to predict targets (here, simulation results) from a set of inputs. ANNs constitute a powerful, flexible method for generalized nonlinear regression, and deliver accurate results in the presence of noisy input data. They have been used in research and commercially to guide autonomous vehicles [30], to play backgammon [36] and to predict weather [23], stock prices, medical outcomes, and horse races. The representational power of ANNs is rich enough to express complex interactions among variables: any function can be approximated to arbitrary precision by a three-layer ANN [24]. We choose ANNs over other predictive models (such as linear or polynomial regression and Support Vector Machines [SVMs]) for modeling parameter spaces in computer architecture because:

1. they represent a mature, already commercialized technology;
2. they do not require the form of the functional relationship between inputs and target values to be known;
3. they operate with real-, discrete-, cardinal-, and boolean-valued inputs and outputs, and thus can represent parameters of interest to an architect; and
4. they work well with noisy data, and thus can successfully be combined with existing mechanisms that reduce the time simulation experiments take at the expense of introducing noise.

Figure 1 shows the basic organization of simple *fully connected, feed-forward* ANNs. Networks consist of an *input layer*, *output layer*, and one or more *hidden layers*. Input values are presented at the input layer; predictions are obtained from the output layer. Each unit operates on its inputs to produce an output that it passes to the next layer. In fully connected feed-forward ANNs, weighted edges connect every unit in each layer to all units in the next layer, communicating outputs to other units downstream. A unit calculates its output by applying its *activation function* to the weighted sum (based on edge weights) of all inputs. Figure 2 depicts a hidden unit using a sigmoid activation function. In general, activation functions need not be sigmoid, but must be nonlinear, monotonic, and differentiable. Our models use sigmoid activation functions.

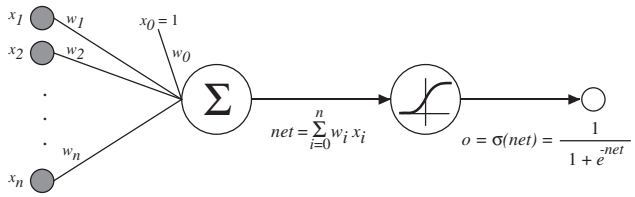

Figure 2. Example of a hidden unit with a sigmoid activation function (borrowed from Mitchell [24]).

Our model's edge weights are updated via backpropagation, using gradient descent in weight space to minimize squared error between simulation results and model predictions. Weights are initialized near zero, causing the network to act like a linear model. During training, examples are repeatedly presented at the inputs, differences between network outputs and target values are calculated, and weights are updated by taking a small step in the direction of steepest decrease in error. As weights grow, the ANN becomes increasingly nonlinear. Every network weight $w_{i,j}$ (where i and j correspond to processing units) is updated according to Equation 1, where E stands for squared-error and η is a small *learning rate* constant (effectively the gradient descent step size).

$$w_{i,j} \leftarrow w_{i,j} - \eta \frac{\partial E}{\partial w_{i,j}} \tag{1}$$

$$w_{i,j} \leftarrow w_{i,j} - (\eta \frac{\partial E}{\partial w_{i,j}} + \alpha \Delta w_{i,j}(n-1)) \tag{2}$$

To speed search and help avoid backpropagation's getting "stuck" in local minima, a *momentum* term in the update rule of Equation 2 causes a weight's update in the current gradient descent iteration to include a fraction of the previous iteration's update. This allows the search to continue "rolling downhill" past inferior local minima. Momentum accelerates gradient descent in low-gradient regions and damps oscillations in highly nonlinear regions.

2.1 Training

Machine learning models require some type of training experience from which to learn. Here we use *direct* training examples consisting of the design space parameter vector to the simulator function along with IPC from the simulation results. Training an ANN involves learning edge weights from these examples. The weights associated with each edge in an ANN define the functional relationship between input and output values. In order to predict IPC from L1 and L2 cache sizes and front-side bus bandwidth, the architect runs a number of cycle-by-cycle simulations for combinations of these parameters, collecting the parameters and resulting IPCs into a *training* dataset. The weights are adjusted based on these data until the ANN accurately predicts IPC from the input parameters. Obviously, a good model must make accurate predictions for parameter combinations on which it was not trained.

The ANN parameters that most impact learning are number of hidden layers, number of hidden units per layer, learning rate (gradient descent step size), momentum, and distribution of initial weights. Finding settings that perform well is typically straightforward. We use one 16-unit hidden layer, a learning rate of 0.001,

and a momentum value of 0.5, initializing weights uniformly on [-0.01,+0.01]. These parameters can be tuned automatically.

2.2 Cross Validation

In polynomial curve fitting, polynomials of high degree yield models that have excellent fit to the training samples yet interpolate poorly; likewise, ANNs may *overfit* to training data. Overfitting yields models that generalize poorly to new data even though they perform well on training data. In contrast to polynomial curve fitting, where model complexity is reduced by decreasing polynomial degree, larger networks for which training is halted *before* gradient descent reaches the minimum error on the training set generally make better predictions [2]. We reserve a portion of the training set and prevent overfitting by stopping gradient descent when squared error on this unbiased sample stops improving. If 25% of the data are used as this *early stopping* set, the training set is 25% smaller, and as with other regression methods, ANNs learn less accurate models from reduced training samples. *Cross validation* both avoids this problem and allows us to estimate model accuracy.

In cross validation, the training sample is split into multiple subsets or *folds*. In our case, we divide training samples into 10 folds, each containing 10% of the training data. Folds 1-8 (80% of the data) are used for training an ANN; fold 9 (10% of the data) is used for early stopping; and fold 10 (also 10% of the data) is used for estimating performance of the trained model. We train another ANN on folds 2-9; use fold 10 for early stopping; and use fold 1 to estimate accuracy. This process is repeated to use the data in each fold successively as early stopping sets and test sets.

We combine the 10 networks into an ensemble, averaging their predictions. Each ANN is trained on 80% of the data, but all data are used to train models in the final ensemble. This ensemble performs similarly to models trained on all data, yet held-aside data are available for early stopping and unbiased error estimation. Averaging multiple models often performs better than using one model. Means and standard deviations of model error on the test folds are used to estimate ensemble accuracy, allowing the architect to determine when the models are accurate enough to be useful.

2.3 Modeling Architectural Design Spaces

Parameters in architectural design spaces can be grouped into a few broad categories. *Cardinal* parameters indicate quantitative relationships (e.g., cache sizes, or number of ROB entries). *Nominal* parameters identify choices, but lack quantifiable properties among their values (e.g., front-end fetch policy in SMTs, or type of coherence protocol in CMPs). Continuous (e.g., frequency) and boolean (e.g., on/off states of power-saving optimizations) parameters are also possible. The encoding of these parameters and how they are presented to ANNs as inputs significantly impact model accuracy. We encode each cardinal or continuous parameter as a real number in [0,1], normalizing with minimax scaling via minimum and maximum values over the design space. Using a single input facilitates learning functional relationships involving different regions in the parameter's range, and normalization prevents placing more emphasis on parameters with broader ranges. We represent nominal parameters with one-hot encoding: we allocate an input unit for each parameter setting, making the input corresponding to the desired setting 1 and those corresponding to other settings 0. This avoids erroneous encoding of range information where none exists. We represent boolean parameters as single inputs with 0/1 values. Target values (simulation results) for model training are encoded like inputs. We scale normalized IPC predictions back to the actual range. When reporting error rates, we perform calculations based on actual (not normalized) values.

When exploring a design space, absolute value of the model error is of little use. For instance, in predicting execution time, erring by one second is negligible for actual time of an hour but significant for actual time of two seconds. When absolute errors have differing costs, ANNs can be trained by presenting points with higher costs more often. *Stratification* replicates each point in the dataset by a factor proportional to the inverse of its target value so that the network sees training points with small target values many more times than it sees those with large absolute values. As a result, the training algorithm puts varying amounts of emphasis on different regions of the search space, making the right tradeoffs when setting weights to minimize percentage error. We perform early stopping based on percentage error.

2.4 Intelligent Sampling

Sample points for training models affect how quickly they learn to approximate the simulator function accurately. Randomly sampling simulation data from the design space yields good performance, but choosing sample points intelligently yields better predictions. *Active learning* [33] is a general class of algorithms that aim for a given accuracy with the fewest possible samples by selecting the points from which the model is likely to derive the most benefit. We seek to identify samples on which the model makes the greatest error, since learning these points is most likely to improve model accuracy. Unfortunately, assessing model error on any point requires knowing simulation results for that point. Since results are unavailable before simulation, identifying points with highest error requires an alternative strategy. We instead measure the variance of the predictions for all ANNs in our cross validation ensemble. After each training round, we query all ANNs for predictions on every point in the design space (testing takes under 10 seconds for 20K+ points). We calculate the variance of model predictions, the mean prediction, and the prediction's coefficient of variance (CoV, or the ratio of the standard deviation to mean). Points with high prediction CoV values are those for which disagreement among the ANNs is largest. Since the ensemble's ANNs differ primarily by the samples (folds) on which they are trained, high disagreement indicates that including the point in the sample set can lower model variance (and thus error). In effect, active learning assigns confidence values based on predictions of the 10 models from our 10-fold cross validation, and then samples the least confident points.

Impacts of different points on model accuracy are dependent on one another. Sorting and sampling strictly according to CoV values can yield a dataset with redundant points. One point may improve the variance of another if these points lie close together in the design space, and adding either point increases model accuracy over the whole region. Sampling both such points is inefficient, since the second presents little additional information. We address this by iteratively choosing samples from the sorted points. We include the least confident point first. We include the next least confident point if its distance (in the design space) to the first point is above a certain threshold. We include each point considered if its distance to current points is above threshold. If we do not find enough points satisfying this constraint, we lower the threshold and reconsider rejected points. Once we have enough sample points, we simulate them, train a new model, and while our error estimate is too high we calculate a new set of points for sampling through the active learning mechanism. Figure 3 summarizes our modeling mechanism for random sampling or active learning. Figure 4 provides a different perspective on the steps in training versus using the model.

3. Experimental Setup

We conduct performance sensitivity studies on memory system, microprocessor, and multithreaded chip multiprocessor (CMP) parameters via detailed simulation of an out-of-order processor and its memory subsystem (SESC) [32]. We model contention and latency at all levels. Phansalkar et al. [29] use principal component analy-

1. Identify important design parameters.
2. Perform a set of simulations for N combinations of parameter settings, possibly reducing the time for each simulation by using statistical simulation techniques (e.g., SimPoint).
3. Normalize inputs and outputs. Encode nominal parameters with one-hot encoding, booleans as 0-1, and others as real values in the normalized 0-1 range. Collect the results in a data set.
4. Divide data set into k folds.
5. Train k neural nets with k-fold cross validation. During training, present each data point to the ANNs at a frequency proportional to the inverse of its IPC (we assume the target to be predicted is IPC; other targets are similar). Perform early stopping based on percentage error.
6. Estimate average and standard deviation of error from cross validation.
7. Repeat 2-6 with N additional simulations if estimated error is too high.
8. Predict any point in the parameter space by placing the parameters at the input layers of all ANNs in the ensemble, and averaging predictions of all models.

Figure 3. Summary of steps in modeling mechanism

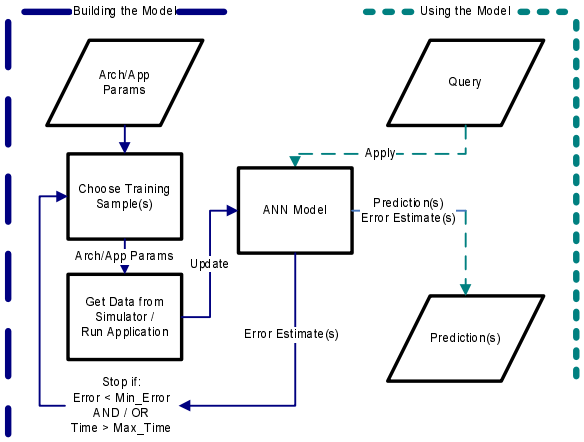

Figure 4. Pictorial representation of modeling mechanism

sis to group SPEC CPU 2000 [35] applications according to metric similarities like instruction mix, branch direction, and data locality. For the memory system and processor studies, we choose codes (bzip2, crafty, gcc, mcf, vortex, twolf, art, mgrid, applu, mesa, equake, and swim) to cover the clusters they identify, ensuring that we model the diverse program behaviors represented by the suite. Since we run over 500K simulations, we choose MinneSPEC [19] reduced reference inputs to validate our approach (results for SPEC reference inputs would be even more impressive in terms of time saved). We run all simulations, then sample these incrementally to train the ANNs, predicting remaining points in the space at each round, and validating predictions against the simulation results. For the CMP study, we use two applications (swim and art) from the SPEC OMP V3.0 suite and one application (mg) from the parallel NAS benchmarks. We use MinneSPEC and Class-S inputs for SPEC OMP and NAS benchmarks, respectively. We train our models on 1.03% of the design space (2500 simulations), and report accuracies obtained on an independently sampled set of 500 points.

We assume a 90nm technology for the processor and memory system studies and a 65nm technology for the CMP study. We derive all cache latencies with CACTI3.2 [39]. Table 1 shows parameters in the memory hierarchy study, the design space for which spans the cross product of all parameter values and requires 23,040 simulations per benchmark. Core frequency is 4GHz. The L2 bus runs at core frequency and the front-side bus is 64 bits. Table 2 shows parameters in the microprocessor study, which requires 20,736 simulations per benchmark. We use core frequencies of 2GHz and 4GHz, and calculate cache and SDRAM latencies and branch misprediction penalties based on these. We use 11- and 20-cycle minimum latencies for branch misprediction penalties in the 2GHz and 4GHz cases, respectively. For register files,

Variable Parameters	Values
L1 DCache Size	8,16,32,64 KB
L1 DCache Block Size	32,64 B
L1 DCache Associativity	1,2,4,8 Way
L1 Write Policy	WT,WB
L2 Cache Size	256,512,1024,2048 KB
L2 Cache Block Size	64,128 B
L2 Cache Associativity	1,2,4,8,16 Way
L2 Bus Width	8,16,32 B
Front Side Bus Frequency	0.533,0.8,1.4 GHz
Fixed Parameters	**Value**
Frequency	4GHz
Fetch/Issue/Commit Width	4
LD/ST Units	2/2
ROB Size	128 Entries
Register File	96 Integer/96 FP
LSQ Entries	48/48
SDRAM 100 ns	64 bit FSB
L1 ICache	32 KB/2 Cycles
Branch Predictor	Tournament (21264)

Table 1. Parameter values in memory system study

we choose two of the four sizes in Table 2 based on ROB size (e.g., a 96 entry ROB makes little sense with 112 integer/fp registers). When choosing the number of functional units, we choose two sizes from Table 2 based on issue width. The number of load, store and branch units is the same as the number of floating point units. SDRAM latency is 100ns, and we simulate a 64-bit front-side bus at 800MHz. The design space for the CMP study spans the cross product of all parameters in Table 3, requiring 241,920 simulations per benchmark for a full sensitivity study. We vary number of cores and SMT contexts, microarchitectural parameters, and parameters of the shared-memory subsystem.

Partial simulation techniques (in which only certain application intervals or simulation points are modeled) reduce time per experiment at the expense of slight losses in accuracy. In SimPoint, Sherwood et al. [34] combine *basic block distribution analysis* with clustering to summarize behavior of sections of program execution; this guides selection of representative samples to simulate in detail. Errors induced by such techniques vary across the parameter space. When combining techniques like SimPoint with our predictive models, the ANNs see noisy results where the precise amount of error depends on the simulation technique, its parameters, and design space parameters. To evaluate our approach with noisy (but quickly obtained) simulation results, we repeat the processor study with SimPoint and four long-running applications (mesa, equake, mcf, crafty). We scale SimPoint intervals from 100M to 10M dynamic instructions to adjust for shorter MinneSPEC runtimes. Otherwise, we run SimPoint out-of-the-box.

4. Evaluation

We address four questions:

1. How much of the design space must we simulate to train our models?
2. How accurate and robust are our predictions versus full simulation?
3. How fast can we train the models?
4. How well does our approach integrate with other approaches to reduce time per simulation?

We answer these via four sensitivity studies: the memory hierarchy and processor studies with random sampling; the memory study with active sampling; and the processor study with SimPoint. We

Variable Parameters	Values
Fetch/Issue/Commit Width	4,6,8 Instructions
Frequency	2,4 GHz (affects Cache/DRAM/ Branch Misprediction Latencies)
Max Branches	8,32
Branch Predictor	1K,2K,4K Entries (21264)
Branch Target Buffer	1K,2K Sets (2-Way)
ALUs/FPUs	2/1,4/2,3/1,6/3,4/2,8/4 (2 choices per Issue Width)
ROB Size	96,128,160
Register File	64,80,96,112 (2 choices per ROB Size)
Ld/St Queue	16/16,24/24,32/32
L1 ICache	8,32KB
L1 DCache	8,32KB
L2 Cache	256,1024KB
Fixed Parameters	**Value**
L1 DCache Associativity	1,2 Way (depends on L1 DCache Size)
L1 DCache Block Size	32B
L1 DCache Write Policy	WB
L1 ICache Associativity	1,2 Way (depends on L1 ICache Size)
L1 ICache Block Size	32B
L2 Cache Associativity	4,8 Way (depends on L2 Cache Size)
L2 Cache Block Size	64B
L2 Cache Write Policy	WB
Replacement Policies	LRU
L2 Bus	32B/Core Frequency
FSB	64bits/800 MHz
SDRAM	100ns

Table 2. Parameter values in the processor study

Variable Parameters	Values
Core Configuration	In-Order, Out-of-Order
Issue Width	1,2,4
Number of Cores	1,2,4,8
SMT Contexts per Core	1,2,4
Off-Chip Bandwidth	8,16,24,32,40,48,56,64 GB/s
Frequency	1,1.5,2,2.5,3,3.5,4 GHz (affects Cache/DRAM/ Branch Misprediction Latencies)
L2 Cache Size	1,2,4,8MB
L2 Cache Block Size	32,64,128B
L2 Cache Associativity	1,2,4,8,16 Way
Fixed Parameters	**Value**
ROB Size	24,48,96 (depends on Issue Width)
Int/FP Issue Queue Sizes	12/12,24/24,48/48 (depends on Issue Width)
LD/ST Queue Sizes	6/6,12/12,24/24 (depends on Issue Width)
Int/FP Rename Registers	24/24,48/48,96/96 (depends on Issue Width)
L2 Bus	64B/Core Frequency
L1 ICache	32KB,2 Way,32B,LRU (latency depends on Core Frequency)
L1 DCache	32KB,2 Way,64B,LRU (latency depends on Core Frequency)
L2 Cache	Shared,Unified,8 Banks (latency depends on Core Frequency and L2 Parameters)
Branch Predictor	Tournament
SMT Fetch Policy	Round Robin
SDRAM	80ns Uncontended

Table 3. Parameter values in the CMP study

also apply our approach to the CMP design space described in Section 3. Here we predict performance because it's well understood, but our approach is general enough to apply to other metrics, even multiple metrics at once.

Several mechanisms demonstrate that our design spaces are complex and nonlinear. First, linear regression yields high error (15-20%) for even dense (8%) samplings of the space: simple linear models are inadequate. Second, we investigate what the learned model "looks like": edge weights deviate significantly from their initial values near zero, indicating that the model is a highly nonlinear function of the design parameters. Third, we perform multidimensional scaling (MDS) analysis on our dataset [14], finding numerous local maxima in different regions, many of which are $\geq$90% of the globally optimal IPC.

4.1 Training Set Size

Like other regression methods, ANNs typically predict better when trained on more data. However, data collection in architecture design space exploration is expensive, and a tradeoff exists between number of simulations and model accuracy. For the processor and memory system studies, we increment our datasets by 50 simulations at each training step. After each training round, we test the ANNs on remaining points in the design space, recording mean percentage error and standard deviation of error and tracking the cross-validation estimates for these metrics. Space constraints limit us to presenting graphs for a subset of representative applications.

Table 4 summarizes results for all applications with randomly selected training samples. We show mean and standard deviation of error along with cross-validation estimates for training sets corresponding roughly to 1%, 2%, and 4% of the full space. Table 6 shows the reduction factor in number of required sample points when using active learning versus random sampling and training the ANNs within error rates of 2-3% for nine applications. For tighter accuracy requirements, we expect active sampling to outperform random sampling: the latter generally requires more samples to deliver comparable results. In most cases we reduce the number of samples, but sometimes there is no change. Nonetheless, active learning never increases sampling requirements, and further study of intelligent sampling strategies is an avenue of ongoing research.

4.2 Accuracy

Learning curves in Figure 5 illustrate percentage error rate decreases in the memory system and processor studies as training set sizes increase (via random sampling). The x axes show portions of full parameter spaces simulated to form training sets, and the y axes show percentage error across the design spaces. Error bars indicate ± 1 standard deviation of the averages.

For the memory system study, training on 0.22% of the full design space (50 training examples) yields average error between 5-10%, with standard deviation of error typically between 10-15% across all applications. These are unacceptably high. The small training set includes insufficient information to capture the functional relationship between design parameters and performance. Standard deviation of error is high, and model accuracy varies significantly from one region of the design space to another, indicating that sampling is too sparse. For the applications in Figure 5, error rates improve dramatically as more data are added to the training sets. When training on roughly 1% of the full space, average error and standard deviation drop to about 0.7-6.7%. Randomly sampling 1% more brings error rates down to 0.6-4.7%. Rates reach an asymptote at sample sizes of about 4%: models for these applications exhibit $<$2% average error.

	Memory System Study											
	1.08% Sample				2.17% Sample				4.12% Sample			
	Mean Error		SD of Error		Mean Error		SD of Error		Mean Error		SD of Error	
Application	True	Estimated	True	Estimated	True	Estimated	True	Estimated	True	Estimated	True	Estimated
equake	2.32%	2.47%	3.28%	4.58%	1.40%	1.39%	1.81%	1.61%	0.92%	0.92%	0.97%	0.98%
applu	3.11%	2.97%	2.74%	2.79%	2.35%	2.57%	1.90%	2.32%	1.28%	1.31%	1.04%	1.21%
mcf	4.61%	4.53%	5.6%	5.73%	2.84%	3.06%	2.94%	3.61%	1.74%	1.77%	1.59%	1.68%
mesa	2.85%	2.8%	4.27%	5.24%	2.69%	2.73%	4.16%	4.77%	1.97%	2.15%	2.87%	3.79%
twolf	4.13%	3.70%	6.23%	5.80%	3.78%	3.30%	5.61%	4.84%	3.67%	3.44%	5.50%	4.95%
crafty	2.16%	2.45%	2.10%	2.38%	1.17%	1.29%	1.10%	1.33%	0.87%	0.96%	0.77%	0.91%
mgrid	4.96%	5.19%	6.12%	6.43%	1.53%	1.52%	1.40%	1.79%	0.83%	0.85%	0.74%	0.75%
art	6.63%	6.83%	5.23%	5.99%	4.69%	4.82%	4.29%	4.45%	2.92%	3.05%	2.86%	3.09%
gcc	3.69%	4.13%	4.02%	5.46%	1.50%	1.49%	1.44%	1.30%	1.13%	1.14%	0.97%	1.09%
bzip2	1.95%	1.90%	1.84%	1.82%	0.97%	0.94%	0.86%	0.81%	0.59%	0.61%	0.48%	0.52%
vortex	4.53%	4.65%	4.63%	5.19%	2.90%	2.96%	3.07%	2.77%	1.90%	2.07%	1.99%	2.14%
swim	0.66%	0.75%	0.52%	0.55%	0.57%	0.55%	1.50%	0.48%	0.54%	0.54%	0.45%	0.46%

	Processor Study											
	0.96% Sample				1.93% Sample				4.10% Sample			
	Mean Error		SD of Error		Mean Error		SD of Error		Mean Error		SD of Error	
Application	True	Estimated	True	Estimated	True	Estimated	True	Estimated	True	Estimated	True	Estimated
equake	1.80%	1.89%	1.39%	1.47%	1.15%	1.99%	0.94%	1.00%	0.72%	0.73%	0.59%	0.64%
applu	1.94%	1.85%	1.45%	1.43%	1.30%	1.29%	0.99%	1.04%	0.87%	0.89%	0.72%	0.82%
mcf	1.67%	1.71%	1.38%	1.51%	1.20%	1.26%	0.99%	1.13%	0.94%	0.99%	0.83%	0.90%
mesa	2.57%	2.851%	1.96%	2.06%	1.27%	1.33%	0.99%	1.05%	0.87%	0.94%	0.69%	0.79%
twolf	4.85%	5.26%	4.76%	5.78%	4.32%	4.28%	4.39%	4.63%	4.34%	4.05%	4.05%	4.29%
crafty	2.65%	2.75%	2.03%	2.07%	1.53%	1.61%	1.25%	1.34%	0.78%	0.81%	0.66%	0.65%
mgrid	1.39%	1.16%	1.13%	0.89%	0.99%	1.00%	0.75%	0.78%	0.74%	0.80%	0.59%	0.66%
art	2.41%	2.34%	1.91%	2.00%	1.67%	1.74%	1.45%	1.53%	1.29%	1.29%	1.12%	1.18%
gcc	1.88%	1.97%	1.48%	1.31%	1.09%	1.05%	0.88%	0.95%	0.59%	0.59%	0.49%	0.52%
vortex	2.90%	3.46%	2.17%	2.68%	1.39%	1.52%	1.11%	1.34%	0.88%	0.93%	0.71%	0.78%
swim	2.65%	1.49%	2.05%	1.88%	1.22%	1.45%	0.94%	1.31%	0.59%	0.57%	0.49%	0.49%
bzip2	1.30%	1.50%	0.95%	1.16%	0.79%	1.86%	0.61%	0.66%	0.56%	0.57%	0.44%	0.46%

	CMP Study (1.03% Sample)			
	Mean Error		SD of Error	
Application	True	Estimated	True	Estimated
art-OMP	4.66%	5.03%	6.05%	5.50%
swim-OMP	4.44%	4.87%	4.65%	3.90%
mg	5.27%	6.41%	4.76%	4.39%

Table 4. Memory hierarchy, processor and CMP study results (random sampling)

Learning curves for the processor parameter study follow similar trends. When simulating only 0.24% of the full design space (50 training examples), the data contain too little information to train accurate models. Depending on the application, average error rate varies between 2.5-6.4%, while standard deviation falls in the 1.8-5.2% range. As more data are sampled, model accuracy improves rapidly. When training set size reaches 1% of the full space, models for the applications in Figure 5 reach average error rates $\leq$2.9% and standard deviations $<$2.2%. Table 4 shows that models for other applications (art, bzip2, equake, gcc, swim, twolf) maintain average error rates of 1.3-4.9% with standard deviations of 1.0-4.8% at this sampling rate. When training set size increases to 2% of the full space, models for all applications except twolf achieve error rates $<$2%. At a 4% sample size, all other models exhibit error rates $\leq$1.3%; the twolf model's error rate drops to 4.3% on average, and at an approximately 8% sample size, to roughly 3.9%. This is not problematic: since cross validation yields accurate error estimates, the architect can continue simulations until error rates become acceptable for the respective study.

Table 4 lists the mean error and the standard deviation of error obtained on the CMP study at a 1.03% sampling of the design space. Models for art, swim and mg obtain average error rates of 4.66%, 4.44%, and 5.27%, and standard deviations of 5.03%, 4.87% and 6.41%, respectively. Note that system performance varies widely across the CMP design space we study (by 25, 41, and 55$\times$ for art, swim, and mg, respectively), and hence these error rates are negligible compared to the performance impact of the parameters we vary. The empirical cumulative distribution function (CDF) plots in Figure 7 show the distribution of error. The x axes show percentage error, and the y axes show the percentage of data points that achieve error less than each x value. 90% of the points are predicted with less than 12.2-9.2% error, and 75% of the points are predicted with less than 5.5-6.5% error. 65-72% of the points are predicted with less than 5% error. These results suggest that ANNs can handle design spaces with widely varying target values and can still deliver low percentage error when trained for the right metric (as explained in Section 2.3).

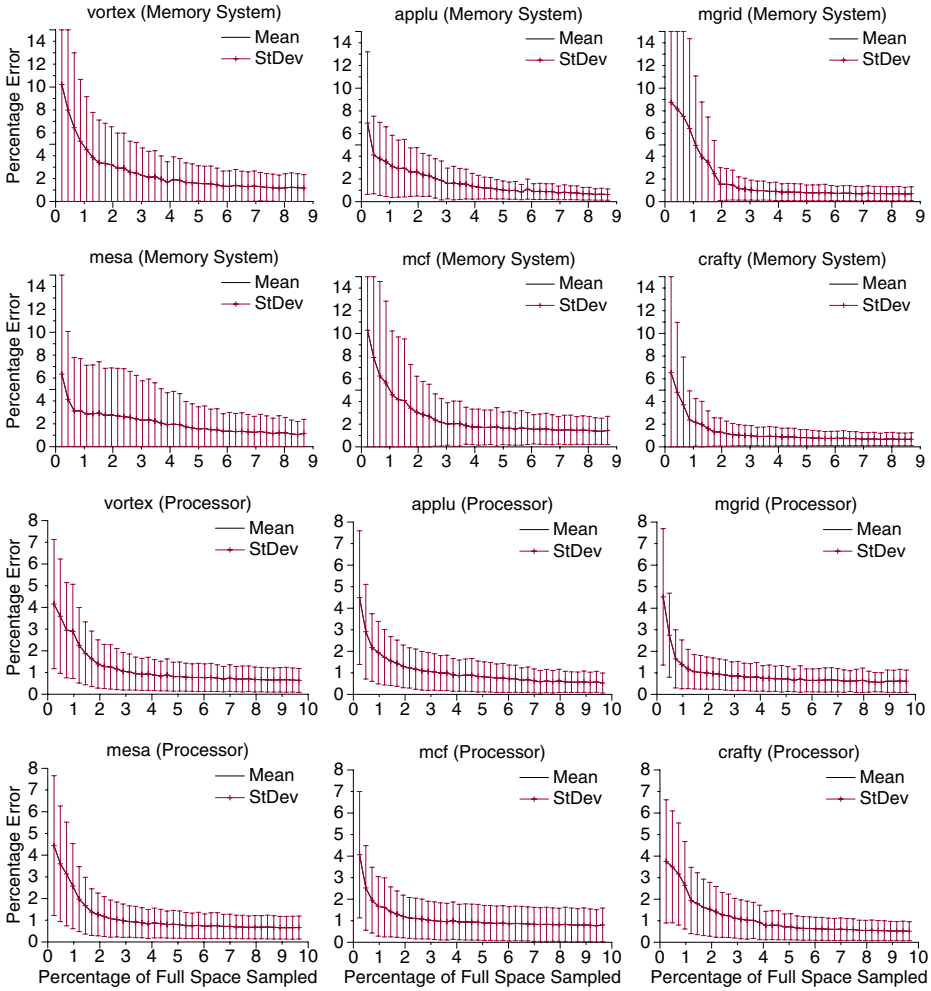

Figure 5. Accuracy of the models on the design spaces (random sampling)

To evaluate the robustness of our error estimation, we compare estimated and true mean error and standard deviation. Figure 6 illustrates these metrics on the memory and processor design spaces as a function of training set size. For almost all codes, cross validation estimates are within 0.5% of actual values for sample sizes $\geq$1%. For sample size $\leq$1%, differences between estimated and actual errors vary between 0.5-4%, but estimates are conservative in this regime. Cross validation estimates error rates from individual ANNs in the ensemble. Final predictions, however, average predictions of all ANNs, typically yielding lower error rates. Cross validation thus slightly *overestimates* actual error, providing a conservative estimate of average prediction accuracy and standard deviation. With sample size above 1%, differences between true and estimated error rates become negligible. Accuracy of these estimates allows the architect to stop collecting simulation results as soon as error rates become acceptable. In our experiments, cross validation almost never underestimates error.

The top of Table 5 shows the best three configurations for bzip2 from the memory hierarchy study. The bottom shows the best predicted configurations for a 2% sampling of the design space. The predicted configurations point to the same narrow area of the design space as the simulation results: L1 cache size, write policy, and L2 size match, and L2 block size differs in only one configuration. For other parameters, the model chooses slightly different tradeoffs from the optimal settings found by simulation, but these do not yield significant performance differences for this application and this particular (optimum) region of the design space. The model comes close to finding the global optimum, and predicted performance characteristics of the region it finds are similar to the actual performance characteristics. Combining this with an approach that ranks parameter importance or characterizes interactions [41, 17] guides the architect in exploring design tradeoffs in the region.

4.3 Training Times

If ANN models are to enable exploration of large design spaces with reasonable expenditures of time and computational resources, it is critical that time required to train the models be *much* smaller than architectural simulation time. Figure 8 shows time required to train our models as a function of training set size. The ANNs in the 10-fold cross validation ensemble are trained in parallel on 10 cluster nodes with 3GHz Intel Pentium 4™ CPUs and 1GB DRAM. Each point represents the average of three measurements. As training sets increase from 1-9% of the parameter space, training times scale linearly from 30 seconds to four minutes. [1] Since simulation results are collected in batches, each round of training is amortized over multiple simulations. Learning curves presented in these studies typically level off at training set sizes of 2-4% of

[1] This result is expected, since the algorithmic complexity of training a neural network with a single hidden layer, H hidden units, I inputs, and O outputs on D data points for P passes through the training set is $O(H(I+O)PD)$.

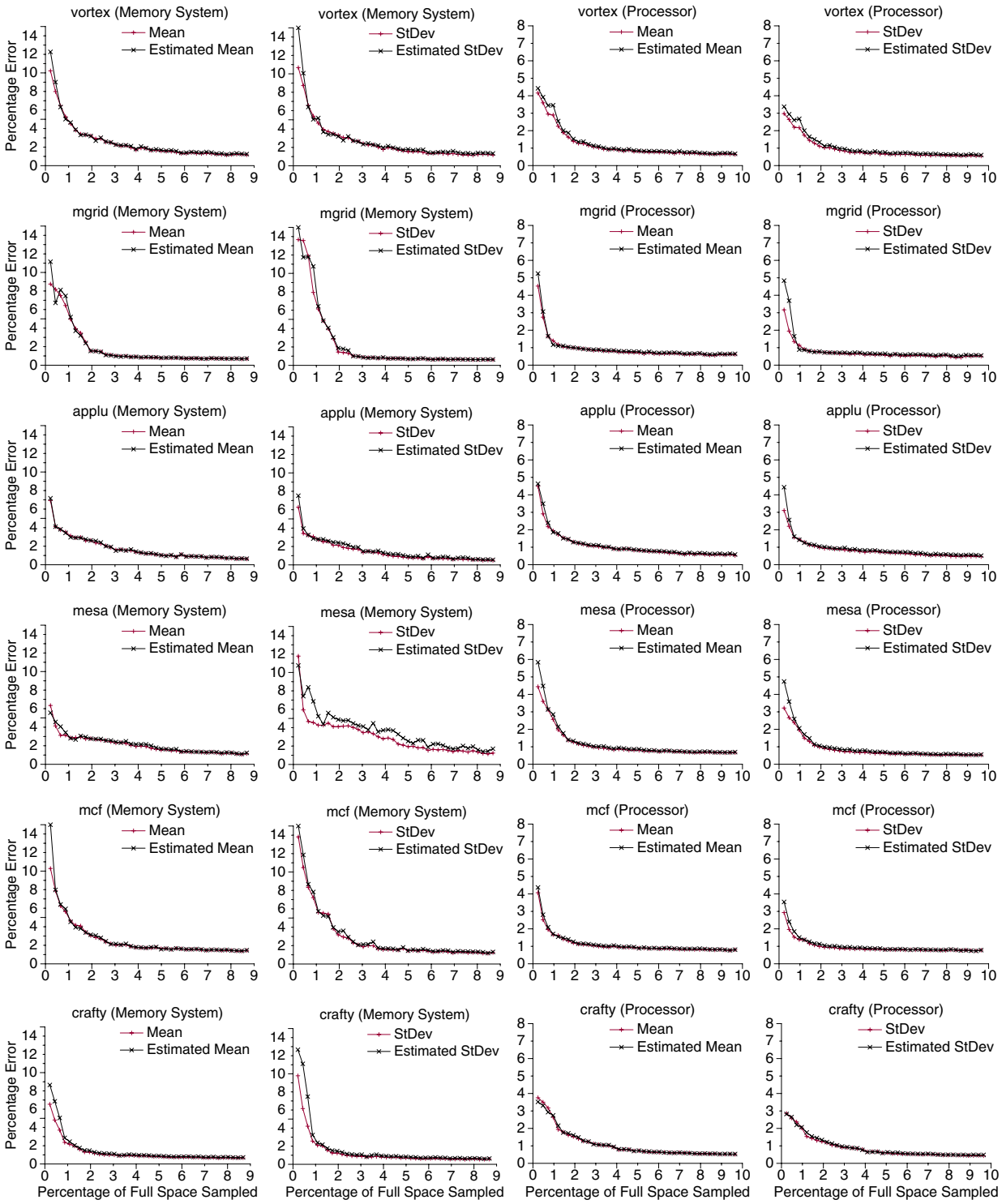

Figure 6. Estimated and true means and standard deviations for percentage error (random sampling)

the full space, requiring less than two minutes per training step (compared to hours, days, or weeks of simulation per design point).

4.4 Integration with SimPoint

Our predictive modeling directly targets large parameter spaces and is orthogonal to techniques that reduce times for single simulations. This orthogonality does not necessarily imply that multiple techniques can be combined successfully. For instance, SimPoint reduces experiment time at the expense of some loss in accuracy. If the two approaches are combined to build a predictive model based on SimPoint samples, that model must handle input imprecision (which acts like noise), and not amplify error. Fortunately, ANNs

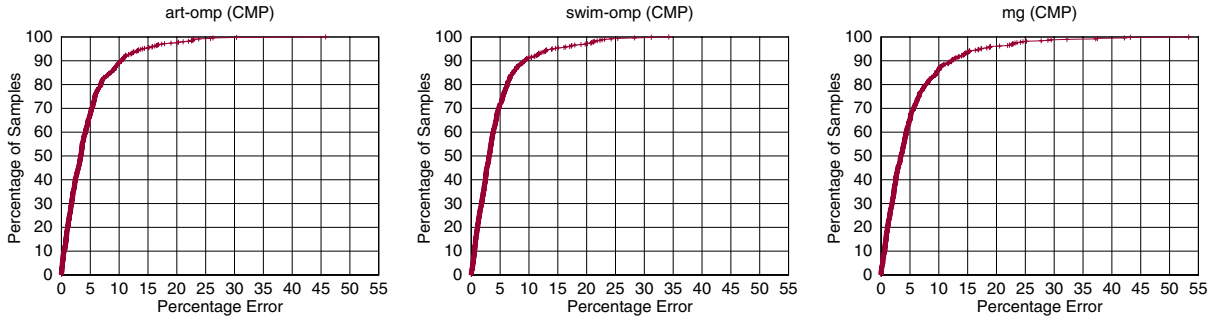

Figure 7. Empirical CDF plots of error on CMP study: x axes show percentage error, and the y axes show the percentage of data points that achieve error less than each x value

L1 size	L1 block size (B)	L1 ways	L1 write policy	L2 size (KB)	L2 block size (B)	L2 ways	L2 bus width	FSB frequency (GHz)	IPC
Best Simulation Configurations									
16	64	2	WB	1K	128	16	32	1.4	1.10
16	64	2	WB	1K	128	16	16	1.4	1.10
16	64	2	WB	1K	128	8	32	1.4	1.10
Best Predicted Configurations									
16	32	2	WB	1K	128	16	32	1.4	1.09
16	32	4	WB	1K	128	16	32	1.4	1.08
16	64	1	WB	1K	128	16	8	0.8	1.08

Table 5. Similarity in best configurations for bzip2

	Application								
Accuracy	applu	mcf	mgrid	mesa	equake	crafty	vortex	bzip2	gcc
98%	1.09	1.23	1.17	1.17	1.15	1.18	1.32	1.0	1.0
97%	1.51	1.29	1.13	2.55	1.0	1.0	1.0	1.15	1.15

Table 6. Reduction factors in number of sample points (active sampling vs. random sampling)

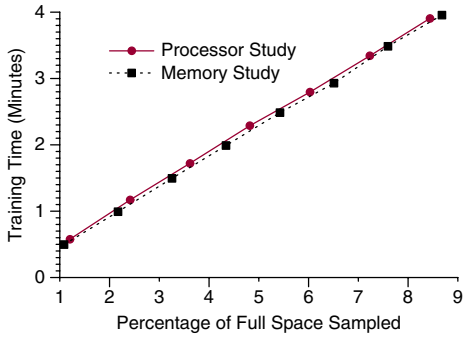

Figure 8. Training times

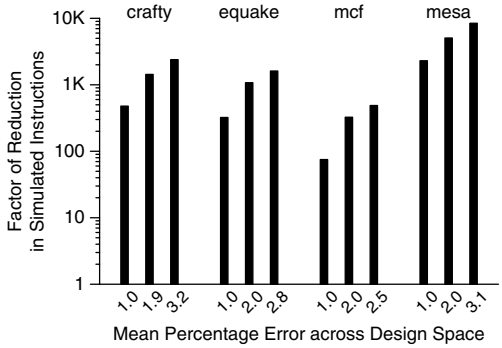

Figure 9. Gains from combining ANNs+SimPoint

work well in the presence of noise. To verify the robustness of our models, we repeat the processor study (with random sampling) using SimPoint. After deriving SimPoints and corresponding weights, we collect results for each application on every point in the space by calculating SimPoint performance estimates per run. We train our ANNs on these noisy datasets, but measure accuracy *with respect to the complete simulations*. Figure 9 shows gains from combining approaches, and Figure 10 shows details. Figure 9 shows reduction in simulated instructions at average percentage errors between 1-4%. The combined approach yields impressive reductions in number of simulated instructions for design space exploration. Even at error rates as low as 1%, the combined approach reduces number of simulated instructions by two-three orders of magnitude. If 3.2% error rates can be tolerated, reductions reach three-four orders of magnitude. Of these gains, ANN modeling contributes about 40-200×, while SimPoint contributes 10-60×.

The top of Figure 10 shows learning curves. When simulating only 0.24% of the parameter space, average error rate and standard deviation vary between 3.5-4.4% and 2.4-3.2%, respectively. Error rates steadily decrease as simulation results are added to the training sets. When 1% of the full space is simulated, average error rate drops to <2.7% and standard deviation drops to 1.4-2.0%. At this sampling rate, the models are accurate and perform consistently well in all regions of the design space, as indicated by lower standard deviation. When training sets contain 2% of the full space, average error falls between 1.1-1.4%. In this regime, standard deviation varies between 0.9-1.2%. Compared to using full simulations, training models with SimPoint results gives slightly higher error, but differences are negligible. The bottom of Figure 10 plots estimated and average error and its standard deviation as a function

of training set size. Estimates are again accurate. One difference between these and the original results is that estimates provided by cross validation beyond a 1% sampling are consistently lower than actual error (differences are small). When cross validation calculates error estimates, it does so with respect to SimPoint results, unaware of the noise in those results. Note, however, that these estimates are never off by more than 1% in this regime.

Our results indicate that ANN ensembles handle the inherent inaccuracies induced by SimPoint well. Typically, average error rates of $\leq$2% are maintained below a 1% sampling of the full design space, and a 1% error rate is obtained by sampling about 2% of the space (representing 50-100$\times$ fewer simulations).

5. Related Work

Several recent articles elucidate aspects of the design space problem. Martonosi and Skadron [22] summarize a 2001 NSF workshop's conclusions: trends towards multiple cores and increasing on-chip heterogeneity will lead us to build systems that are difficult to simulate; we require research into abstractions and evaluation methodologies that make quantitative evaluations of complex systems manageable; existing tools dictate the majority of research, resulting in light exploration of difficult-to-model parts of the design space; and the community's emphasis on simulation may cause practitioners to overlook other useful and possibly more informative modeling techniques. Jacob [16] spends six months simulation time to study a small part of a memory system design space. Davis et al. [5] struggle with massive design spaces for in-order multithreaded CMP configurations: they employ industry guidelines to prune the space, but find the remaining find 13K design points for in-order CPUs alone unmanageable.

5.1 Analytic and Statistical Models

Noonberg and Shen [27] use probability vectors to compose a set of components as linked Markov chain models solved iteratively, delivering 90-98% accuracy on a set of kernels. Karkhanis and Smith [18] construct an intuitive first-order analytic model of superscalar microprocessors that estimates performance with 87-95% accuracy compared to detailed simulation. Fields et al. [11] define *interaction costs* (icosts) to identify events affecting another event of interest. They propose efficient hardware to enable sampling execution in sufficient detail to construct statistically representative microarchitecture graphs for computing icosts.

Yi et al. demonstrate Plackett and Burman fractional factorial design [41] in prioritizing parameters for sensitivity studies. They model a high and low value for a set of N parameters, varying each independently to explore extremes of the design space in $2N$ simulations. By focusing on the most important parameters, PB analysis can reduce the simulations required to explore a large design space. Chow and Ding [3] and Cai et al. [1] apply principal components analysis and multivariate analysis to identify the most important parameters and their correlations for processor design. Eeckhout et al. [9] and Phansalkar et al. [29] use principal components analysis for workload composition and benchmark suite subsetting.

Muttreja et al. [25, 26] perform macromodeling, pre-characterizing reusable software components to construct high-level models to estimate performance and energy consumption. Symbolic regression filters irrelevant macromodel parameters, constructs macromodel functions, and derives optimal coefficient values to minimize fitting error. They apply their approach to simulation of several embedded benchmarks, yielding estimates with maximum 1.3% error. Joseph et al [17] develop linear models primarily for identifying significant parameters and their interactions. Not intended to be predictive, model prediction accuracy depends on use of appropriate input transformations.

Lee and Brooks [21] propose regression on cubic splines (piecewise polynomials) for predicting performance and power for applications executing on microprocessor configurations in a large microarchitectural design space. Their approach is not automated, and requires some statistical intuition on the part of the modeler. Like us, they observe high accuracies at sparse samplings. We expect our approaches to be complementary, and are currently collaborating with them to perform fair comparisons and gain insight into predictive modeling, in general.

5.2 Reduced Simulation Workloads

Statistical simulation [8] represents an attractive alternative to full simulation for many purposes. The technique first derives application characteristics; generates a much smaller, synthetic trace exhibiting those characteristics; and then simulates that trace. Oskin et al. [28] develop a hybrid simulator (HLS) that uses statistical profiles to model application instruction and data streams. HLS dynamically generates a code base and symbolically executes it on a superscalar microprocessor core much faster than detailed, observing average error within 5-7% of cycle-by-cycle simulation of a MIPS R10000. Iyengar et al. [15] evaluate *representativeness* of sampled, reduced traces (with respect to actual application workloads) and develop a graph-based heuristic to generate better synthetic traces. Eeckhout et al. [6] build on this to generate statistical control flow graphs characterizing program execution, attaining 1.8% average error on 10 SPEC 2000 benchmarks.

5.3 Partial Simulation Techniques

Wunderlich et al. [40] model minimal instruction stream subsets in SMARTS to achieve results within desired confidence intervals. The approach can deliver high accuracies, even with small sampling intervals. For large intervals, as in SimPoint [34], state warmup becomes less significant, but can still improve accuracy.

Conte et al. [4] and Haskins and Skadron [12] sample portions of application execution, performing *warmup* functional simulation to create correct cache and branch predictor state for portions of the application being simulated in detail. Haskins and Skadron exploit *Memory Reference Reuse Latencies* (MRRLs) to choose the number of warmup instructions to simulate functionally before a desired simulation point [13]. This selection of warm-up periods roughly halves simulation time with minimal effect on IPC accuracy. Eeckhout et al. [7] further reduce warmup periods with *Boundary Line Reuse Latencies* (BLRLs), in which they consider only reuse latencies that cross the boundary between warmup and sample. Van Biesbrouck et al. [37] investigate warmup for both SimPoint and SMARTS, storing reduced *Touched Memory Images* (TMIs) and *Load Value Sequences* (LVSs) of data to be accessed in a simulation interval in conjunction with *Memory Hierarchy State* (MHS) collected through cache simulation of the target benchmark. They find MHS+LVS to be as accurate as MRRLs with faster simulation, and to require less storage than TurboSMARTS [38] checkpoints.

Rapaka and Marculescu [31] use a hybrid simulation engine to detect code hotspots of high temporal locality and use information from these to estimate statistics for the remaining code. This approach needs no application behavior characterization before simulation. Their adaptive profiling strategy predicts application performance and power with less than 2% error while speeding simulations by up to 12$\times$. This approach is not tied to a given architecture or application/input pairs, but requires modifying the simulation model for adaptive sampling.

6. Conclusions

We demonstrate that Artificial Neural Networks can model large design spaces with high accuracy and speed (training the ANNs on

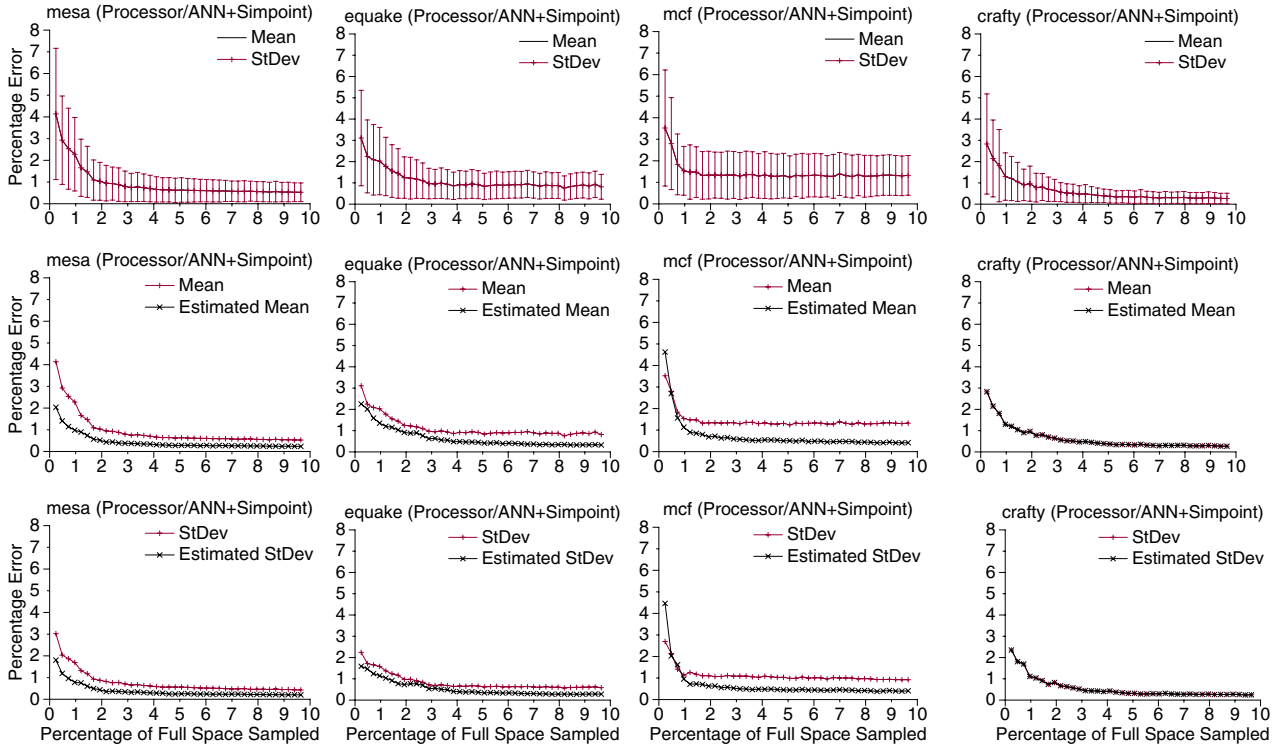

Figure 10. SimPoint processor study results (random sampling)

1-2% of the space lets us predict results for other design points with 98-99% accuracy). Our approach is potentially of great value to computer architects, who rely on design space exploration to evaluate the sensitivity of a proposal to many interacting architectural parameters. We present a fully automated, general mechanism to build accurate models of architectural design spaces from limited simulation results. The approach is orthogonal to statistical techniques that reduce single simulation times or assess parameter importance, and we show that combining our models with one already widely used technique reduces number of simulated instructions by three-four orders of magnitude. We make several contributions:

- a general mechanism to build highly accurate, confident models of architectural design spaces, allowing architects to cull uninteresting design points quickly and focus on the most promising regions of the design space;
- a framework that incorporates additional simulation results incrementally and allows models to be queried to predict performance impacts of architectural changes, enabling efficient discovery of tradeoffs among parameters in different regions;
- an evaluation showing that training times are negligible compared to even individual architectural simulations; and
- an analysis of results showing that our approach can reduce simulation times for sensitivity studies by several orders of magnitude with almost no loss in accuracy.

We predict performance here, but our approach is sufficiently general to predict other statistics, even several at once. Our mechanism enables much faster exploration of design spaces of currently feasible sizes, and makes possible exploration of massive spaces outside the reach of current simulation infrastructures. We thus provide the architect with another tool to assist in the design and evaluation of systems. In so doing, we hope to increase understanding of design tradeoffs in a world of ever increasing system complexity.

Acknowledgments

Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory (LLNL) under contract W-7405-Eng-48 (UCRL-CONF-223240) and under National Science Foundation (NSF) grants CCF-0444413, OCI-0325536, and CNS-0509404. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or LLNL. The authors thank Dave Albonesi, Kai Li, José Martínez, José Moreira, Karan Singh, and the anonymous reviewers for feedback on this work. A research equipment grant from Intel Corp. provided computing resources that helped enable our design space studies.

References

[1] G. Cai, K. Chow, T. Nakanishi, J. Hall, , and M. Barany. Multivariate power/performance analysis for high performance mobile microprocessor design. In *Power Driven Microarchitecture Workshop*, June 1998.

[2] R. Caruana, S. Lawrence, and C. Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Proc. Neural Information Processing Systems Conference*, pages 402–408, Nov. 2000.

[3] K. Chow and J. Ding. Multivariate analysis of Pentium Pro processor. In *Intel Software Developers Conference*, pages 84–91, Oct. 1997.

[4] T. Conte, M. Hirsch, and K. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proc. IEEE International Conference on Computer Design*, pages 468–477, Oct. 1996.

[5] J. Davis, J. Laudon, and K. Olukotun. Maximizing CMP throughput with mediocre cores. In *Proc.IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 51–62, Oct. 2005.

[6] L. Eeckhout, R. Bell, Jr., B. Stougie, K. De Bosschere, and L. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Proc. 31st IEEE/ACM International Symposium on Computer Architecture*, pages 350–361, June 2004.

[7] L. Eeckhout, Y. Luo, L. John, and K. De Bosschere. BLRL: Accurate and efficient warmup for sampled processor simulation. *The Computer Journal*, 48(4):451–459, 2005.

[8] L. Eeckhout, S. Nussbaum, J. Smith, and K. De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003.

[9] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction Level Parallelism*, 5:http://www.jilp.org/vol5, Feb. 2003.

[10] S. Eyerman, L. Eeckhout, and K. D. Bosschere. The shape of the processor design space and its implications for early stage explorations. In *Proc. 7th WSEAS International Conference on Automatic Control, Modeling and Simulation*, pages 395–400, Mar. 2005.

[11] B. Fields, R. Bodick, M. Hill, and C. Newburn. Interaction cost and shotgun profiling. *ACM Transactions on Architecture and Code Optimization*, 1(3):272–304, 2004.

[12] J. Haskins, Jr. and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proc. IEEE International Conference on Computer Design*, pages 32–39, Sept. 2001.

[13] J. Haskins, Jr. and K. Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pages 195–203, Mar. 2003.

[14] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Verlag, 2001.

[15] V. Iyengar, L. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In *Proc. 2nd IEEE Symposium on High Performance Computer Architecture*, pages 62–73, Feb. 1996.

[16] B. Jacob. A case for studying DRAM issues at the system level. *IEEE Micro*, 23(4):44–56, 2003.

[17] P. Joseph, K. Vaswani, and M. Thazhuthaveetil. Use of linear regression models for processor performance analysis. In *Proc. 12th IEEE Symposium on High Performance Computer Architecture*, pages 99–108, Feb. 2006.

[18] T. Karkhanis and J. Smith. A 1st-order superscalar processor model. In *Proc. 31st IEEE/ACM International Symposium on Computer Architecture*, pages 338–349, June 2004.

[19] A. KleinOsowski and D. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[20] R. Kumar, V. Zyuban, and D. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proc. 32nd IEEE/ACM International Symposium on Computer Architecture*, pages 408–419, June 2005.

[21] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proc. 12th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[22] M. Martonosi and K. Skadron. NSF computer performance evaluation workshop `http://www.princeton.edu/~mrm/nsf_sim_final.pdf`, Dec. 2001.

[23] C. Marzban. A neural network for tornado diagnosis. *Neural Computing and Applications*, 9(2):133–141, 2000.

[24] T. Mitchell. *Machine Learning*. WCB/McGraw Hill, Boston, MA, 1997.

[25] A. Muttreja, A. Raghunathan, S. Ravi, and N. Jha. Automated energy/performance macromodeling of embedded software. In *Proc. 41st ACM/IEEE Design Automation Conference*, pages 99–102, June 2004.

[26] A. Muttreja, A. Raghunathan, S. Ravi, and N. Jha. Hybrid simulation for embedded software energy estimation. In *Proc. 42nd ACM/IEEE Design Automation Conference*, pages 23–26, July 2005.

[27] D. Noonburg and J. Shen. Theoretical modeling of superscalar processor performance. In *Proc. IEEE/ACM 27th International Symposium on Microarchitecture*, pages 53–62, Nov. 1994.

[28] M. Oskin, F. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *Proc. 27th IEEE/ACM International Symposium on Computer Architecture*, pages 71–82, June 2000.

[29] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pages 10–20, Mar. 2005.

[30] D. Pomerleau. Knowledge-based training of artificial neural networks for autonomous robot driving. In J. Connell and S. Mahadevan, editors, *Robot Learning*, pages 19–43. Kluwer Academic Press, Boston, 1993.

[31] V. Rapaka and D. Marculescu. Pre-characterization free, efficient power/performance analysis of embedded and general purpose software applications. In *Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition*, pages 10504–10509, Mar. 2003.

[32] J. Renau. SESC. `http://sesc.sourceforge.net/index.html`.

[33] M. Saar-Tsechansky and F. Provost. Active learning for class probability estimation and ranking. In *Proc. 17th International Joint Conference on Artificial Intelligence*, pages 911–920, Aug. 2001.

[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.

[35] Standard Performance Evaluation Corporation. SPEC CPU benchmark suite. `http://www.specbench.org/osg/cpu2000/`, 2000.

[36] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, Mar. 1995.

[37] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *Proc. 1st International Conference on High Performance Embedded Architectures and Compilers*, pages 47–67, Nov. 2005.

[38] T. Wenisch, R. Wunderlich, B. Falsafi, and J. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. *SIGMETRICS Performance Evaluation Review*, 33(1):408–409, 2005.

[39] S. Wilton and N. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[40] R. Wunderlich, T. Wenish, B. Falsafi, and J. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proc. 30th IEEE/ACM International Symposium on Computer Architecture*, pages 84–95, June 2003.

[41] J. Yi, D. Lilja, and D. Hawkins. A statistically-rigorous approach for improving simulation methodology. In *Proc. 9th IEEE Symposium on High Performance Computer Architecture*, pages 281–291, June 2003.