EFFICIENTLY EXPLORING ARCHITECTURAL DESIGN SPACES VIA PREDICTIVE MODELING

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Engin İpek

August 2007

ABSTRACT

Computer architects rely on cycle-by-cycle simulation to evaluate the impact of design choices and to understand tradeoffs and interactions among design parameters. Although several techniques reduce time per individual simulation, efficiently exploring exponential-size design spaces spanned by several interacting parameters remains an open problem: the sheer number of experiments renders detailed simulation intractable. We attack this via an automated approach for building highly accurate and confident predictive models of design spaces. We collect simulation data incrementally, giving reliable estimates of model error on the full parameter space at each step of the building process. As validation, we perform sensitivity studies on memory system and microprocessor design spaces (conducting over 300K detailed simulations). Our models generally predict IPC with less than 1-2% error, even when trained on as little as 2% of the full design space. Further, our mechanism is orthogonal to techniques that reduce simulation runtimes. SimPoint [23] reduces the number of simulated instructions per experiment by 8-62$\times$. We reduce the total number of simulated instructions by 50-200$\times$. *Combining our approach with SimPoint reduces the number of simulated instructions required to complete thorough design-space explorations by 1000-13,000*$\times$. Our approach has potential to quantitatively and qualitatively transform computer architecture research, enabling studies heretofore beyond our computational abilities.

BIOGRAPHICAL SKETCH

Engin Ipek received his B.S. in Elecetrical and Computer Engineering from Cornell University in May 2003. He is currently a Ph.D. student at Cornell's Computer Systems Laboratory.

To my lovely wife, Zeynep

ACKNOWLEDGEMENTS

I would like to acknowledge the contribution of several individuals to the completion of this thesis. First, I thank my advisor Sally McKee for being the best advisor I could hope for, for teaching me how to do research in computer architecture and guiding my academic career. In the two years that I have been a graduate student at Cornell, she has made me truly feel like a family member within our research group. This thesis would certainly not be possible without her dedication and hard work, and I will never forget her influence on my career.

I thank Professor Rich Caruana for guiding me in my exploration to apply machine learning techniques to problems in computer architecture, and for teaching me how to reason pragmatically about complex problems. In the two years that I have known him, he and I have spent countless hours engaging in technical discussions, devising research ideas and analyzing results. I consider myself very fortunate to have the opportunity of working with him on this project.

I thank my mentor Bronis de Supinski at Lawrence Livermore National Laboratory for his technical contributions to the project, for encouraging me to pursue this idea from the day I proposed it to him, and for showing me problems faced by practitioners and researchers in high performance computing. I thank Martin Schulz for his contributions to the project, especially for his help on running the simulations at the laboratory.

I thank professor José F. Martínez for useful comments and discussions on the topic, and for not minding me constantly taking his time with technical discussions on this and other projects. I thank professor David Albonesi for his useful feedback on this work, and Professor Martin Burtscher for introducing me to computer architecture and for being an excellent teacher.

I thank my colleagues Pete Schzwed, Brian White, Vince Weaver, Jian Li, Meyrem and Nevin Kırman and Amy Henning for useful discussions and comments.

I thank my father and mother for their constant support throughout my academic career, and for pushing me to pursue my ambitions. Finally, I thank my wife Zeynep for her love and support. She is the best person I have ever met, and the reason behind everything I do. This thesis is dedicated to her.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vii
List of Tables . . . viii
List of Figures . . . ix

1 Introduction 1

2 Design Space Exploration 3

3 Modeling Parameter Spaces with Artificial Neural Networks 6
3.1 Training Artifical Neural Networks . . . 8
3.2 Cross Validation . . . 9
3.3 Modeling Architectural Design Spaces . . . 11

4 Experimental Setup 15

5 Evaluation 18
5.1 Learning Curves . . . 19
5.2 Error Estimation . . . 21
5.3 Integration with Existing Schemes . . . 24
5.4 Training Times . . . 30

6 Related Work 32

7 Conclusions and Future Work 36

A 39

Bibliography 43

LIST OF TABLES

4.1 Variable (left) and constant (right) simulation parameters and their values in the memory system study. 16

4.2 Variable (left) and constant (right) simulation parameters and their values in the processor study. 16

5.1 Results for all studies. 19

LIST OF FIGURES

3.1 Simplified diagrams of fully connected, feed-forward neural networks. 7
3.2 Example of a hidden unit with a sigmoid activation function (borrowed from Mitchell [19]). 7
3.3 Example of a 10-fold cross-validation ensemble on 1K training points. The top bar shows the distribution of data to folds. Train/ES/Test indicate training, early-stopping and test sets, respectively. 11
3.4 Example for encoding nominal and cardinal parameters. 12

5.1 Error rates of the models on the design space. The columns on the left and right show results for the memory system and processor studies, respectively. 20
5.2 Estimated and true means and standard deviations for percentage error on the memory system study. 22
5.3 Estimated and true means and standard deviations for percentage error on the processor study. 23
5.4 Error rates when ANN modeling and SimPoint are combined. 27
5.5 Estimated and true means and standard deviations for percentage error when ANN modeling is combined with SimPoint. 28
5.6 Gains from combining ANN+SimPoint. 29
5.7 Contributions of SimPoint and ANN to total gains. 29
5.8 Training times. 30

A.1 Error rates of the models on the design space. The columns on the left and right show results for the memory system and processor studies, respectively. 40
A.2 Estimated and true means and standard deviations for percentage error on the memory system study. 41
A.3 Estimated and true means and standard deviations for percentage error on the processor study. 42

Chapter 1
Introduction

Quantifying the impact of design parameters on evaluation metrics and understanding tradeoffs and interactions among such parameters permeates the foundation of computer architecture. Architects rely on this understanding to perform cost-benefit analyses among alternative design options and to propose solutions to open research problems. We usually study tradeoffs and interactions via cycle-by-cycle simulation of a target machine. Several factors have unacceptably increased the time and resources required for this approach, including the desire to model more demanding, realistic workloads; the ever increasing complexity of the architectures we model; and the exponential size of the design spaces spanned by many independent parameters. Thorough study of even relatively modest design spaces —even with substantial, dedicated computing resources— becomes challenging, if not infeasible.

Research on reducing time per experiment or identifying the most important subspaces to explore within a full parameter space has made significant advances that improve our ability to conduct more thorough studies. Nonetheless, simulation times for full design space exploration remain infeasible for most researchers. **Managing the exponential increase in simulation space size with an appropriate number of significant parameters in the general case remains an open problem, the solutions of which are fundamental to advancements in computer architecture.**

We attack this problem by applying machine learning techniques to train ensembles of artificial neural networks (ANNs) from relatively few sample simulation results. Once trained, the ANN ensemble provides estimates for simulation results on the entire parameter space under consideration. At each step of the training process the model provides an accurate estimate of its average error and variance *on the full parameter space*, allowing us to incrementally collect simulation results until we reach an acceptably low

error rate. Our approach is fully automatic, and makes no assumptions regarding the general form of the target function describing relationships among parameters and target metrics under study. Furthermore, our models deliver accurate results (including reliable estimates of the model's error on the full space) with only a very sparse sampling of the entire simulation design space. We make several contributions:

- a general mechanism to build highly accurate and confident models of design spaces in computer architecture;
- a framework to incorporate additional simulation (training) results incrementally;
- demonstrations that our mechanism and framework are orthogonal and complementary to existing techniques that reduce simulation time for sensitivity studies; and
- evaluations of the overhead of our approach demonstrating that its training times are negligible compared to even individual architectural simulations, and that it can reduce simulation times required for sensitivity studies by several orders of magnitude with almost no loss in accuracy.

Combining our models with orthogonal techniques can yield multiplicative reductions in the number of instructions simulated in an architectural design space exploration—up to factors of tens of thousands for the cases we study. Likewise, such combined mechanisms enable the detailed study of design spaces that would otherwise be outside the reach of current simulation technology.

The rest of this thesis addresses design space exploration in Chapter 2, and gives a high level overview of our approach; discusses artificial neural networks and their application to modeling architectural design spaces in Chapter 3; presents our simulation infrastructure and the design spaces we explore in Chapter 4; evaluates our mechanism in Chapter 5; details related work in Chapter 6; and presents our conclusions in Chapter 7.

Chapter 2
Design Space Exploration

Sensitivity studies evaluate the effects of a set of design parameters, where these parameter values are varied in combination through a set of simulation experiments. As such, they constitute an essential tool for computer architects. Researchers and practitioners use sensitivity studies to verify that apparent performance gains of novel architectural features are not just artifacts of the specific configuration of basic architectural characteristics (such as cache sizes, number of ROB entries or pipeline depth) or to study effects of parameter values specific to a novel feature. Since sensitivity studies compare detailed simulated performance for a range of design parameters and applications, they consume enormous amounts of CPU time: the total number of simulations required is exponential in the number of parameters explored, and a single, detailed simulation experiment may take days or even weeks. For instance, Jacob [14] reports over six months of simulation time just to study a small portion of the memory system design space.

We have therefore developed an approach that can reduce the number of simulations required for a full sensitivity study of M parameters by 50-200$\times$ with almost no loss in accuracy. We view the simulator as a (potentially) highly nonlinear function of its parameter configuration and the input application A: $SIM(p_0, p_1, ...p_M, A)$. Instead of sampling this function at every point (i.e., parameter vector) of interest, we employ powerful, non-linear regression to approximate it.

We use an ensemble of artificial neural networks (ANNs) as our non-linear regression technique. Our approach requires that we first sample a small number of parameter configurations (chosen at random) through simulation, on which we train the ANN ensemble to generate an initial approximation of the simulator function. We use cross validation (discussed in Section 3) to obtain highly accurate estimates of the error of this approximation for the full parameter space. We refine our approximation by further

sampling the parameter space until the error estimate is sufficiently low. Our results demonstrate that in almost all cases this technique provides highly accurate approximations (errors of less than 2%) when sampling only 2-5% (or less) of the full parameter space.

A more thorough treatment of related work is presented in Chapter 6. Here we briefly introduce prior research upon which we build. Partial simulation techniques, in which only certain application intervals or "simulation points" are modeled, address time required per experiment. Generating accurate statistics requires that these simulation points be chosen carefully: Sherwood et al. [23] find that simulating the first million instructions yields 85% average error for SPEC CINT 2000, and fast-forwarding one billion and then simulating 100 million yields 51% average error. To address this error, they develop SimPoint, which uses *Basic Block Distribution Analysis* combined with clustering to summarize the behavior of an arbitrary section of program execution. This information is used to select representative samples to simulate in detail, greatly reducing simulation time without sacrificing significant statistical accuracy. Combining our approach with theirs yields multiplicative reductions in instructions simulated without significantly increasing modeling error: for our applications and workloads, we observe savings by factors of 1,000 to 13,000.

Yi et al. [29] use Plackett and Burman fractional factorial design to prioritize design parameters for sensitivity studies. We employ their method to verify our choice of design parameters to vary in our studies.

Architectural simulations using the SPEC CPU2000 suite [24] generally execute much faster using MinneSPEC reduced input sets [16]. SPEC codes exhibit different behaviors with the different input sets, but MinneSPEC permits exploring large parameter spaces more efficiently, aiding architects in choosing configurations to simulate in detail with official SPEC workloads. Given that we run 300K simulations, we

choose MinneSPEC to validate our approach. Results for official SPEC reference inputs would be even more impressive with respect to time saved, and combining native fast-forwarding [25], SimPoint, and our modeling approach constitutes an interesting future study. Likewise, combining our approach with the SMARTS framework is another interesting future work.

Chapter 3
Modeling Parameter Spaces with Artificial Neural Networks

An *artificial neural network* (ANN) is a machine learning model that automatically learns to predict targets (here, simulation results) from a set of input values. ANNs can be considered a powerful form of non-linear regression. Figure 3.1 shows the basic organization of simple *fully connected, feed-forward* ANNs. The network consists of an *input layer*, *output layer*, and one or more *hidden layers*. Input values are presented at the input layer, and predictions are obtained from the output layer. Layers contain processing *units* (nodes), each of which operates on its inputs to produce an output that is passed to units in the next layer. The ANN in Figure 3.1(a) has a single hidden layer with four hidden units and a single output unit; the ANN in Figure 3.1(b) has two hidden layers with four hidden units each and two output units.

In fully connected feed-forward ANNs, weighted edges connect every unit in each layer to all units in the next layer. Edges communicate a unit's computed value to other units downstream. Every edge has a weight. To compute its output, a unit calculates the weighted sum (based on edge weights) of all its inputs, applies its *activation function* to this sum, and passes the result to the next network layer through the outgoing edges. Figure 3.2 depicts the basic computation performed by a hidden unit using a sigmoid activation function. Other activation functions are possible. The main requirements for an activation function are that it be non-linear, monotonic, and differentiable.

Although there are other predictive modeling methods (such as linear or polynomial regression, Support Vector Machines [SVMs], and decision trees), several qualities make ANNs a better choice for modeling parameter spaces in computer architecture. Specifically, ANNs:

- represent a mature and already commercialized technology;

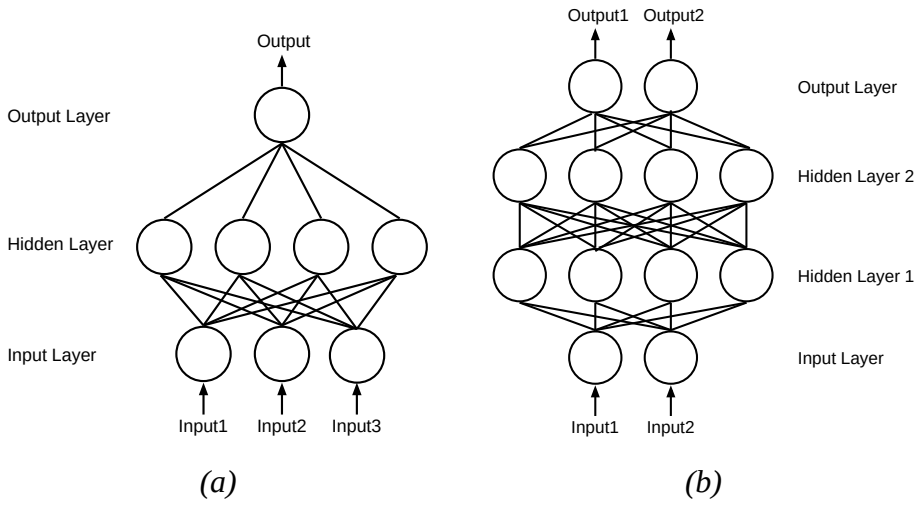

Figure 3.1: Simplified diagrams of fully connected, feed-forward neural networks.

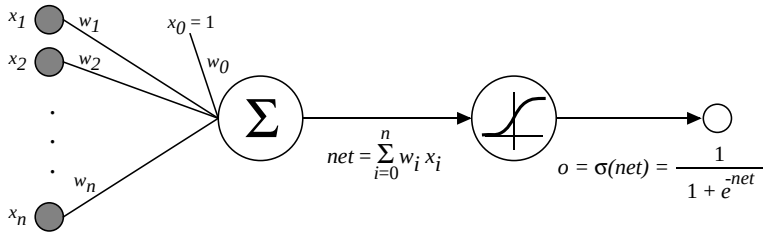

Figure 3.2: Example of a hidden unit with a sigmoid activation function (borrowed from Mitchell [19]).

- do not require that the form of the functional relationship between inputs and target values be known;
- handle real-, discrete-, cardinal-, and boolean-valued inputs and outputs, and thus are capable of representing the different parameters of interest to an architect;
- work well with noisy data, and thus can be combined successfully with existing mechanisms that reduce the time for a single simulation experiment at the expense of introducing noise.

ANNs represent one of the most powerful, flexible methods known for performing generalized nonlinear regression. Their representational power is rich enough to express complex, non-linear interactions among multiple variables. Any function can be

approximated to arbitrary precision by an ANN with three layers [5].

3.1 Training Artificial Neural Networks

The weights associated with each edge in an ANN define the functional relationship between input and output values. Training an ANN involves learning the edge weights from a set of sample data points (tuples of input and output values, corresponding to design space parameters and simulation results in our case). For example, to learn 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 various combinations of these architectural parameters, and collects the parameters and resulting IPCs into a training set. This data is then used to adjust the weights in the ANN until it accurately predicts IPC from the input parameters. A good model must accurately predict IPC for parameter combinations on which it was not trained.

We use the *backpropagation* algorithm to train edge weights. Backpropagation uses gradient descent in the weight space to minimize the squared error between simulation results and model predictions. Edge weights are initialized near zero, causing the network initially to act like a simple linear model. During training, examples are repeatedly presented at the inputs, differences between network outputs and target values are calculated, and backpropagation updates all weights by taking a small step in the direction of steepest decrease in error. Every network weight $w_{i,j}$ (where i and j correspond to processing units) is updated according to Equation 3.1, where E stands for squared-error and η is a small *learning rate* constant (effectively the gradient descent step size). As the weights grow, the ANN becomes increasingly non-linear.

$$w_{i,j} \leftarrow w_{i,j} - \eta \frac{\partial E}{\partial w_{i,j}} \tag{3.1}$$

$$w_{i,j} \leftarrow w_{i,j} - (\eta \frac{\partial E}{\partial w_{i,j}} + \alpha \Delta w_{i,j}(n-1)) \tag{3.2}$$

Like all gradient descent methods on complex surfaces, backpropagation can get "stuck" in local minima. To combat this, a *momentum* term α is often added to the update rule, as shown in Equation 3.2. With momentum, updates to the weights during the current iteration of gradient descent partially depend on updates during the previous iteration. This allows the search to continue "rolling downhill" past inferior local minima by giving the search sufficient momentum to overcome "small hills." Momentum also speeds convergence by accelerating gradient descent in regions with a low gradient and by damping oscillations in highly non-linear regions.

Main parameters affecting ANN learning are the number of hidden layers, number of hidden units in each layer, learning rate, momentum, and distribution of the initial weights. Tuning these may be necessary for some problems, but typically it is not difficult to find reasonable settings that yield good performance. We use networks with one hidden layer consisting of 16 hidden units, a learning rate of 0.001, momentum equal to 0.5, and initialize weights uniformly on [-0.01,+0.01]. These parameters can be set automatically by gauging the adequacy of different settings through our error estimation, which we discuss next.

3.2 Cross Validation

An ANN with a large enough hidden layer can approximate any continuous function. As in polynomial curve fitting, where using a polynomial of high degree results in models that have excellent fit to the training samples yet interpolate poorly, ANNs also may *overfit* to the training samples. However, unlike polynomial curve fitting, where model complexity is reduced by decreasing the degree of the polynomial, experience

with ANNs has shown that better predictions usually are made by large networks with excess capacity where training is halted before gradient descent reaches the minimum error on the training set [19, 2]. For this purpose, a portion of the training set (called the *early stopping set*) is held aside; gradient descent is stopped when the model's squared error on this unbiased sample stops improving.

Holding aside a fraction of the training data to determine when to halt training is an effective way of preventing overfitting in ANNs. Unfortunately, if 25% of the data is used as the early stopping set, the training set used for gradient descent is 25% smaller. As with other regression methods, ANNs learn less accurate models if the training sample is reduced. Cross validation is a method that permits the use of early stopping sets with minimal impact on model performance due to this reduction in training set size. As we show in Section 5.2, cross validation also allows us to estimate model accuracy.

In cross validation, the training sample is split into multiple subsets, or *folds*. For example, 10-fold cross validation splits the training sample into 10 equal-sized folds, each containing 10% of the training data. An ANN is then trained on the samples in folds 1-8 (80% of the data); fold 9 (10% of the data) is used for early stopping; and fold 10 (also 10% of the data) is used to estimate the performance of the trained model. A second ANN is then trained on folds 2-9; fold 10 is used for early stopping; and fold 1 is used to estimate accuracy. This process is repeated 10 times, with the data in each fold being used successively as early stopping sets and test sets (see Figure 3.3).

The 10 networks that result from 10-fold cross validation are then combined into an *ensemble* by averaging the predictions made by each ANN. Although each ANN is trained on only 80% of the training data, all data are eventually used to train some models in the final ensemble. As a result, the ensemble performs similarly to a model trained on all data, yet held-aside data is always available for early stopping and unbiased error estimation. Further, experience has shown that averaging multiple models (an approach

	1−100	101−200	201−300	301−400	401−500	501−600	601−700	701−800	801−900	901−1000
	Fold1	Fold2	Fold3	Fold4	Fold5	Fold6	Fold7	Fold8	Fold9	Fold10
Model 1	Train	Train	Train	Train	Train	Train	Train	Train	ES	Test
Model 2	Test	Train	Train	Train	Train	Train	Train	Train	Train	ES
Model 3	ES	Test	Train	Train	Train	Train	Train	Train	Train	Train
Model 4	Train	ES	Test	Train	Train	Train	Train	Train	Train	Train
Model 5	Train	Train	ES	Test	Train	Train	Train	Train	Train	Train
Model 6	Train	Train	Train	ES	Test	Train	Train	Train	Train	Train
Model 7	Train	Train	Train	Train	ES	Test	Train	Train	Train	Train
Model 8	Train	Train	Train	Train	Train	ES	Test	Train	Train	Train
Model 9	Train	Train	Train	Train	Train	Train	ES	Test	Train	Train
Model 10	Train	Train	Train	Train	Train	Train	Train	ES	Test	Train

Figure 3.3: Example of a 10-fold cross-validation ensemble on 1K training points. The top bar shows the distribution of data to folds. Train/ES/Test indicate training, early-stopping and test sets, respectively.

frequently used in weather forecasting) often yields better performance than training just one model: averaging the 10 ANNs trained with cross validation often yields better accuracy than a single network trained on all the data.

The mean and standard deviation of the model errors made across the 10 test folds are used to estimate the accuracy of the cross validation ensemble across the design space. This estimate allows the architect to determine when the models are accurate enough to be useful. In general, partitioning the data into more folds results in lower error rates and better estimates of the network's accuracy, at the expense of a higher computational cost to train more models. In this thesis we use 10-fold cross validation for all of our experiments.

3.3 Modeling Architectural Design Spaces

Parameters of interest in architectural design spaces can be grouped into a few broad categories. *Cardinal* parameters indicate quantitative relationships (e.g., cache sizes,

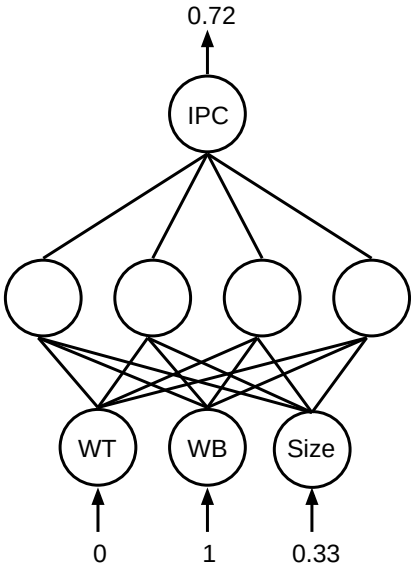

Figure 3.4: Example for encoding nominal and cardinal parameters.

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 the way they are presented to ANNs as inputs significantly impact model accuracy.

We follow a systematic approach when representing these parameter types. We encode each cardinal or continuous design parameter as a single real number in the [0,1] range. We normalize these parameters by using their minimum and maximum values over the design space with minimax scaling. Using a single input facilitates the learning of functional relationships involving different regions in the parameter's range, while normalization prevents placing more emphasis on parameters with a broader range than others. On the other hand, we do not encode nominal parameters in this manner, since they do not represent quantitative properties. Rather, we represent them using one-hot encoding by allocating a separate input unit for each setting of the parameter. For every

possible setting, the corresponding ANN input is set to 1 while those corresponding to other settings of the same parameter are 0. This avoids erroneous encoding of range information where none exists. Boolean parameters are represented as single inputs with 0-1 values. As an example, Figure 3.4 shows how an 8KB, write-back L1 cache configuration is presented to the network, where the possible settings for the write-policy and size are (WT,WB) and (4KB,8KB,16KB), respectively.

Target values (simulation results) used for training the models are encoded in the same way as inputs. In this study, we focus on predicting performance (IPC), and follow the approach described above for continuous parameters when encoding it. After the ANN ensemble provides predictions for a design point, we scale the normalized prediction back to the actual range. When reporting percentage error rates, we do all of our calculations based on these actual (not normalized) values.

One architecture-specific issue when building ANNs is training for percentage error instead of absolute or squared error. When exploring a design space with our proposed mechanism, the absolute value of the error that the model makes on any given design point has little value—rather, one is typically interested in the model's error represented as a percentage of the actual simulation result. For instance, when predicting the execution time of an application, erring by one second is negligible if the actual runtime is 60 minutes, but is significant if the true execution time is two seconds. When training ANNs, gradient descent by default takes steps in the direction of steepest decrease in absolute squared error, and considers these two error rates equal. This can result in poor percentage error across the design space, since the ANN is not trained to optimize the correct metric. A common way of training ANNs when absolute errors on different samples have differing costs is to present points with higher costs to the ANN more often during training than those with lower costs. In the case of percentage error, data points are presented at a frequency based on their target values. This effectively focuses

backpropagation's attention on different data points based on percentage error, training the ANN for the correct metric. In addition, early stopping is based on percentage error as opposed to absolute error.

The following procedure summarizes our overall modeling mechanism:

1. Identify important design parameters.
2. Perform a set of simulations for N random 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 the average and standard deviation of error from cross validation.
7. If estimated error is too high, repeat 2-6 with N additional simulations.
8. Predict any point in the parameter space by placing the parameters at the input layers of all ANNs in the ensemble, and averaging the predictions of all models.

Chapter 4
Experimental Setup

We evaluate our proposal by conducting performance (IPC) prediction sensitivity studies on memory system and microprocessor parameters. Our infrastructure is based on detailed execution-driven simulation of an out-of-order processor and its memory subsystem [22]. Contention and latency are modeled in detail at all levels. For each study, we run our simulations on four SPEC CINT2000 (gzip, mcf, crafty, and twolf) and four SPEC CFP2000 (mgrid, applu, mesa, equake) benchmarks. As stated earlier, we use the reference versions of the MinneSPEC reduced input sets. In both studies, we validate the significance of the parameters we vary through Plackett and Burman fractional factorial designs with foldover, as in Yi et al. [29].

Table 4.1 shows parameters in the memory system performance study and their corresponding values. The right side lists fixed parameters; the left half shows the parameters we vary. The resulting design space spans the cross product of all parameter values, yielding 23,040 simulations per benchmark, for a total of 184,320. We fix core clock frequency at 4GHz, assume a 90nm technology, and derive latencies of all cache configurations through CACTI3.2 [26]. The L2 bus runs at core frequency (as in the Pentium $4^{®}$), and the front-side bus is 64 bits (typical in several current-generation systems).

Table 4.2 shows parameters in the processor study and their corresponding values. Fixed parameters again appear on the right side of the table; the parameters that we vary are on the left. We again assume a 90nm technology and derive latencies of caches through CACTI 3.2. We vary core frequency between 2GHz and 4GHz, and calculate cache and SDRAM latencies as well as branch misprediction penalties based on these values. When setting branch misprediction penalties, we use 11- and 20-cycle minimum latencies for the 2GHz and 4GHz cases, respectively. When varying the size of the register files, we choose two out of the four sizes shown in Table 4.2 based on the ROB size,

Table 4.1: Variable (left) and constant (right) simulation parameters and their values in the memory system study.

Parameter	Values
L1 DCache Size	8,16,32,64KB
L1 DCache Block Size	32,64B
L1 DCache Associativity	1,2,4,8 Way
L1 Write Policy	WT,WB
L2 Cache Size	256,512,1024,2048KB
L2 Cache Block Size	64,128B
L2 Cache Associativity	1,2,4,8,16 Way
L2 Bus Width	8,16,32B
Front Side Bus Frequency	0.533,0.8,1.4 GHz

Parameter	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	32KB/2 cycles
Branch Predictor	Tournament (A21264)

Table 4.2: Variable (left) and constant (right) simulation parameters and their values in the processor study.

Parameter	Values
Fetch/Issue/Commit Width	4,6,8 Instrs
Frequency	2,4 GHz
Max Branches	16,32
Branch Predictor	1K,2K,4K Entries (21264)
Branch Target Buffer	1K,2K Sets (2-way)
Functional Units	4,8
ROB Size	96,128,160
Register File	64,80,96,112 (2 choices per ROB Size)
LSQ	32/32,48/48,64/64
L1 ICache	8,32KB
L1 DCache	8,32KB
L2 Cache	256,1024KB

Parameter	Value
L1 DCache Associativity	1,2 way (dependent on L1 DCache Size)
L1 DCache Block Size	32B
L1 Dcache Write Policy	WB
L1 ICache Associativity	1,2 way (dependent on L1 ICache Size)
L1 ICache Block Size	32B
L2 Cache Associativity	4,8 way (dependent 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

instead of taking the cross product of all four register file sizes with all other parameters (since, e.g., a 96 entry ROB+112 makes little sense with 112 integer/fp registers). We set SDRAM latency at 100ns, and simulate a 64-bit front-side bus at 800MHz. The full design space requires 20,736 simulation per benchmark, and a total of 165,888 simulations. We perform all simulations and measure our error over this full design space when validating our approach.

Chapter 5
Evaluation

Like other regression methods, ANNs typically make better predictions when they are trained on more data. However, data collection in architectural design space exploration is expensive, requiring cycle-by-cycle simulation for every data point in the training set. A tradeoff thus exists between the number of simulations performed and model accuracy. For both the memory system and processor studies, we evaluate this tradeoff by training ANN ensembles through 10-fold cross validation (as in Section 3.2) on training sets containing results of 50-2000 simulations. This corresponds to 0.22-8.70% and 0.24-9.60% of the full design spaces in the memory system and processor studies, respectively. We train the models on progressively larger sets (in increments of 50 simulations) by incorporating additional randomly sampled points from the parameter space into the existing training sets. Once trained, we test the ANN ensembles on the remaining points in the design space that were not used for training, and record average percentage error and standard deviation of error on these points. In addition, we track the cross-validation estimates for the mean and standard deviation of percentage error. We present graphs for four representative applications (mesa, mcf, equake, and crafty). Results for the remaining applications are similar, and are contained in Appendix A. Table 5.1 summarizes results for all eight applications: for each application, the table lists the average and standard deviation of error across the tested design space for training sets corresponding roughly to 1%, 2%, and 4% of the full space. Cross validation estimates for both the average and standard deviation of error are listed under the “Estimated” columns.

Table 5.1: Results for all studies.

	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	Est.	True	Est.	True	Est.	True	Est.	True	Est.	True	Est.
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%
gzip	1.82%	1.63%	1.42%	1.56%	1.03%	1.17%	0.87%	0.95%	0.81%	0.83%	0.68%	0.68%
twolf	5.63%	5.40%	6.96%	6.94%	4.73%	5.07%	6.32%	6.39%	4.16%	4.41%	6.01%	6.65%
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%

	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	Est.	True	Est.	True	Est.	True	Est.	True	Est.	True	Est.
equake	2.11%	3.21%	1.53%	2.19%	1.23%	1.38%	0.99%	1.04%	0.53%	0.54%	0.41%	0.43%
applu	3.13%	2.19%	2.34%	1.55%	0.93%	0.99%	0.80%	0.83%	0.62%	0.64%	0.59%	0.58%
mcf	2.11%	2.57%	1.57%	2.05%	1.29%	1.28%	1.06%	1.07%	0.94%	0.91%	0.87%	0.84%
mesa	1.50%	1.41%	1.24%	1.79%	0.81%	0.83%	0.61%	0.70%	0.35%	0.36%	0.27%	0.29%
gzip	1.42%	1.69%	1.23%	1.53%	1.07%	1.12%	0.89%	0.99%	0.76%	0.78%	0.62%	0.69%
twolf	6.48%	7.39%	6.94%	8.97%	5.81%	6.29%	6.42%	7.51%	4.94%	4.94%	6.49%	6.77%
crafty	2.43%	2.52%	1.82%	2.20%	1.11%	1.26%	0.87%	1.07%	0.44%	0.44%	0.37%	0.39%
mgrid	4.29%	4.25%	3.77%	4.24%	1.95%	2.09%	1.76%	2.40%	0.88%	0.96%	0.75%	0.82%

5.1 Learning Curves

Figure 5.1 shows *learning curves* illustrating how our models' percentage error rates on the parameter spaces decrease as training set sizes increase (by performing more simulations). In each graph, the x axis shows the percentages of the full parameter space simulated to form the training set, and the y axis shows the percentage error across the design space of the models trained on that set. Solid lines show average percentage error, with error bars placed at ± 1 standard deviation of the averages. Results for the memory system study are given in the left column; results for the processor study are given in the right column.

For the memory system study, when training data are 0.22% of the full design space (50 simulations), average error varies between 5-10%, while the standard deviation of error is typically between 10-15%. This is unacceptably high error for computer architecture research. The training set is so small that it includes insufficient information to capture the functional relationship between design parameters and performance. Stan-

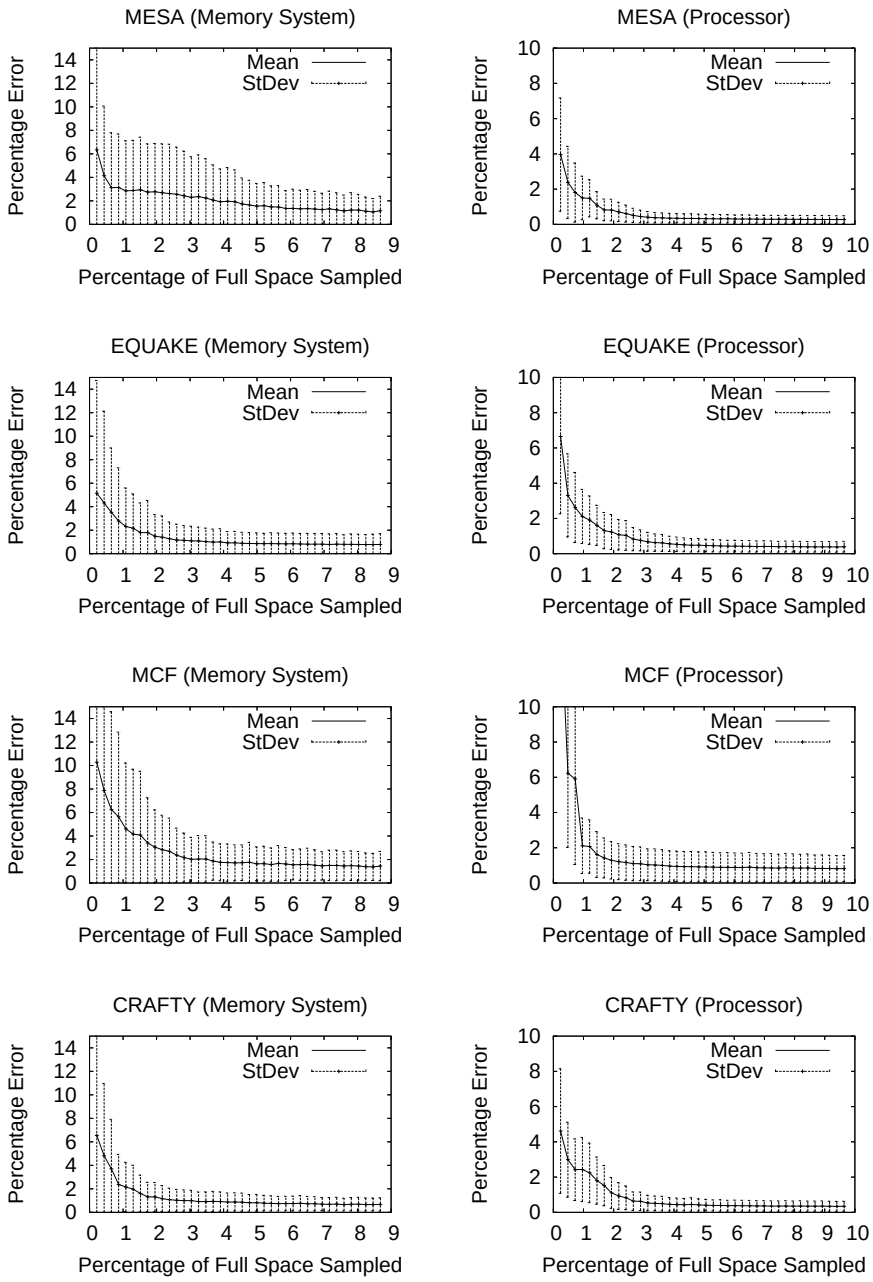

Figure 5.1: Error rates of the models on the design space. The columns on the left and right show results for the memory system and processor studies, respectively.

dard deviation of error is high, and the accuracy of the models varies significantly from one region of the design space to another, indicating that sampling is too sparse. Error rates improve dramatically as more data are added to the training sets. When the train-

ing set contains roughly 1% of the full design space, both the average errors and the standard deviations drop to 2-5%. Sampling an additional 1% brings error rates down to 1-3%. Error rates start to reach an asymptote at a sample size of 4%, at which point models for all four applications exhibit less than 2% average error.

Learning curves for the processor parameter study follow similar trends. When only 0.24% (50 simulations) of the full design space is simulated, the data contain too little information to train accurate models. In this regime, depending on the application, average error rates across all benchmarks vary between 2-15%, while the standard deviations fall in the 3-10% range. As more data are sampled from the parameter space via simulation, accuracy of the ANN ensembles improves rapidly. When training set size reaches 0.96% of the full space, models for the four applications in Figure 5.1 reach average error rates below 2.5% and standard deviations below 2%. However, as Table 5.1 indicates, models for three applications (applu, twolf, and mgrid) maintain average error rates in the 3-6.5% with standard deviations between 2-7% at this point. When training set size increases to 1.93% of the full space, models for all applications except twolf and mgrid achieve error rates lower than 1%. At this point, the model for mgrid yields roughly 2% average error. The model for twolf yields higher error rates than the other models, and its error rates drop more slowly with increasing training set size. At a 4% sample size, the twolf model's error rate drops to 4.9% on average, and at an approximately 16% sample size, to roughly 3%. This difference in behavior is not problematic: cross validation yields excellent error estimates, and the architect can continue simulations until acceptable error rates are attained, as we show next.

5.2 Error Estimation

Figure 5.2 and Figure 5.3 illustrate the estimated and true mean errors and standard deviations on the design spaces as a function of the training set size for the memory

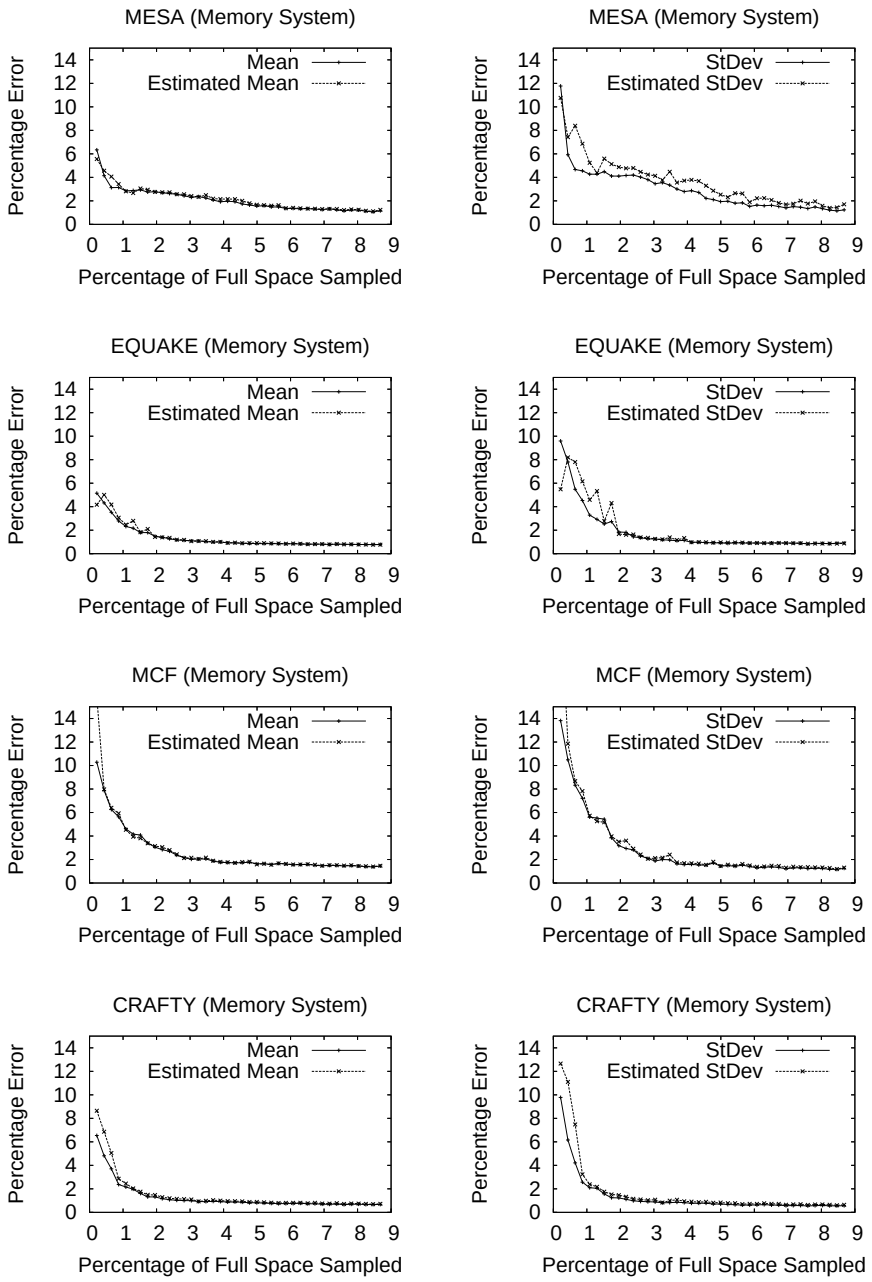

Figure 5.2: Estimated and true means and standard deviations for percentage error on the memory system study.

system and processor studies. For each graph, the x axis shows the size of the training set as a percentage of the full design space, and the y axis shows percentage error. For most applications, the estimates provided by cross validation are within 0.5% of the

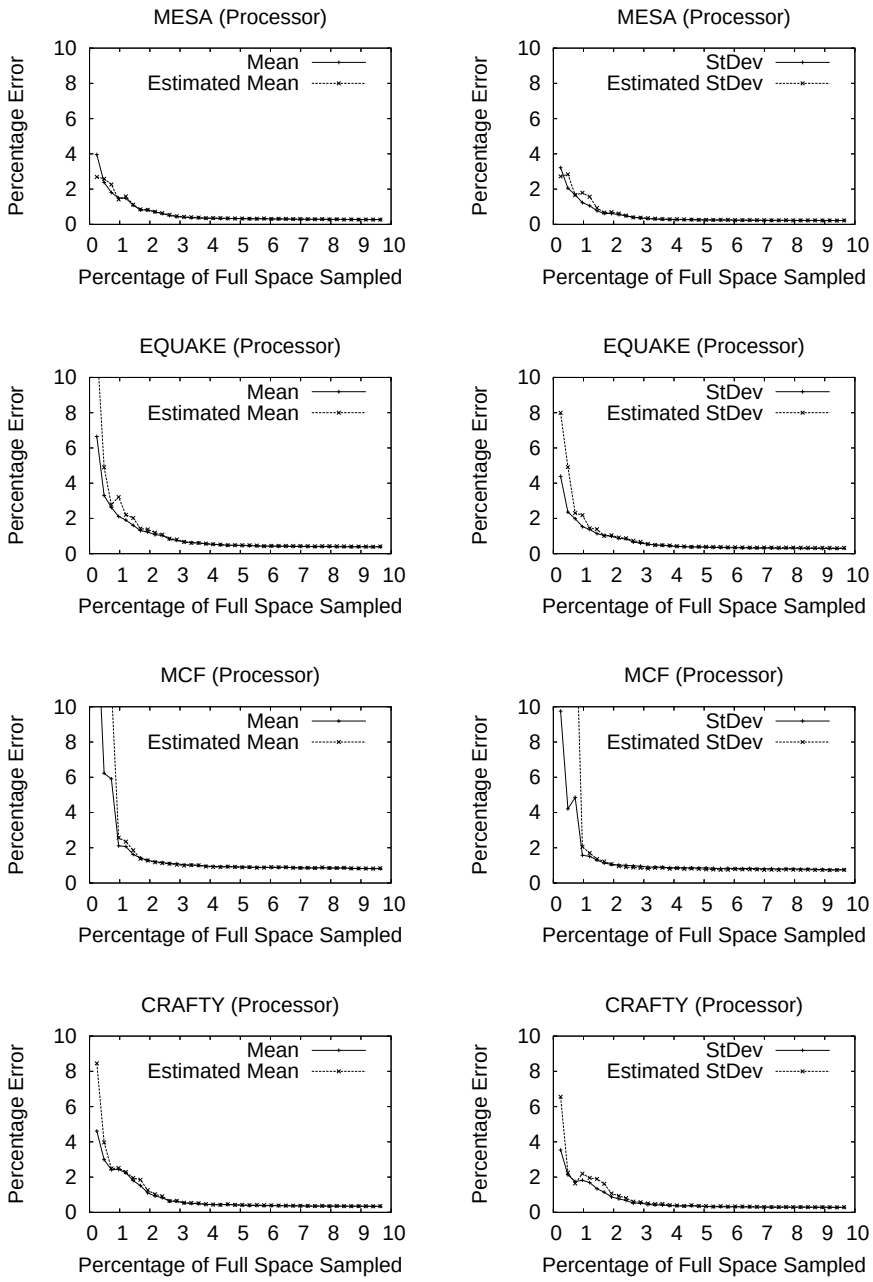

Figure 5.3: Estimated and true means and standard deviations for percentage error on the processor study.

actual values once sample sizes exceed 1%. When the sample size is smaller than 1%, differences between estimated and actual error values are higher and vary between 0.5-4%. Note that in this regime the estimates are conservative. Cross validation estimates

error from the error rates of individual models in the ANN ensemble on the test folds. Final predictions, however, are made by averaging the predictions of all models, which typically yields lower error rates. Because of this, cross validation slightly *overestimates* actual error (especially when sampling is too sparse and error rates are high), providing a conservative estimate of the average prediction accuracy and standard deviation. When the sample size is larger than 1%, the differences between true and estimated error rates are negligible. The accuracy of the estimates allows the architect to stop collecting simulation results as soon as the error rates become acceptable. In our experiments, cross validation almost never underestimates error when trained on actual simulation results.

5.3 Integration with Existing Schemes

Our predictive modeling approach directly targets the problem of large parameter spaces and is orthogonal to techniques that reduce the running times of single simulations. Note, however, that this orthogonality does not necessarily imply that multiple techniques can be combined successfully. For instance, statistical simulation techniques that reduce the runtime of simulations typically do so at the expense of loss in accuracy. Error rates induced by these techniques vary from one point in the parameter space to another. As a result, during training, the ANN ensemble never sees the true outcomes of predictions, but rather sees noisy simulation results where the precise amount of noise (error) depends on the statistical simulation technique, its parameters, and the parameters of the design space. Hence, if the two approaches are to be combined and a predictive model is to be built based on such noisy samples, it is crucial for the predictive model to handle output noise well and not amplify this inherent error. Fortunately, ANNs work well in the presence of noise and can be combined with these techniques successfully.

To explore the efficacy of ANN learning in the presence of noisy — but faster — simulation results, we combine our proposed approach with SimPoint [23]. The sheer number of simulations we run prevents us from using SPEC reference input sets, applying SimPoint to them, and measuring the results of combining our approach with respect to those reference inputs. Instead, we select the four longest-running applications in our study (mesa, mcf, crafty, equake), and use SimPoint to find representative simulation points for these applications, scaling the default interval from 100 million dynamic instructions to 10 million. This adjusts for shorter running times with MinneSPEC inputs and allows SimPoint to reduce simulation times of individual applications significantly. Aside from this change, we run SimPoint out-of-the-box. After finding simulation points and their corresponding weights, we perform the processor study a second time, this time collecting per-interval performance results for each application on every point in the full parameter space, and calculating SimPoint's estimate for the performance of each run. We train our ANN ensembles on these noisy data sets, and measure accuracy with respect to the actual design space in the absence of SimPoint. Figure 5.4 and Figure 5.5 show the results.

Figure 5.4 shows learning curves obtained when SimPoint and ANN modeling are combined. When only 0.24% of the full parameter space is simulated (50 simulations), average error rates and standard deviations vary between 3.7-13% and 2.9-7.7%, respectively. As in the initial processor sensitivity study (without SimPoint), these training sets contain insufficient information to build accurate models. Error rates steadily decrease as more simulation results are added to the training sets. When 0.96% of the full space is simulated using SimPoint and models are trained on this data, average error rates drop to less than 2.5%, while standard deviations are in the 1.2-1.6% range. At this point, the models are both accurate and perform consistently well in all regions of the design space, as indicated by the lower standard deviations. When an additional 1% of the

design space is sampled, training sets contain 1.92% of the full space, and average error falls between 0.6-1.5%. In this regime, standard deviation varies between 0.5-1.5%. Error rates start to decrease asymptotically beyond a 4% sample size. When compared to training on full simulation runs, learning models from SimPoint results give slightly higher error, but in all cases the differences are negligible.

Figure 5.5 plots the estimated and average error and its standard deviation as a function of the training set size when ANN modeling is combined with SimPoint. As in the original processor study, the estimates are accurate, and are conservative when the sampling of the design space is too sparse. One difference between these results and the original ones is that outside of the conservative error estimation regime described earlier, the estimates provided by cross validation are slightly lower than actual (differences are small in all cases). When cross validation calculates error estimates, it performs its calculations with respect to the SimPoint results, unaware of the noise in those results. Note, however, that the 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 less than 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 (50-100$\times$ fewer simulations).

Figure 5.6 shows factors of reduction in number of simulated instructions at three different values of average error between 1% and 4% when ANN modeling and SimPoint are combined. The ANN+SimPoint approach yields orders of magnitude reductions in the number of simulated instructions. Even when error rates as low as 1% are required, the ANN+SimPoint approach reduces the number of simulated instructions by 172-906$\times$. For error rates of roughly 2%, reductions reach 671-8681$\times$. If 3.5% error can be tolerated, reductions reach 1129-13018$\times$. Of these gains, 41-208$\times$ come from ANN modeling, while SimPoint contributes an additional 8-63$\times$ (Figure 5.7).

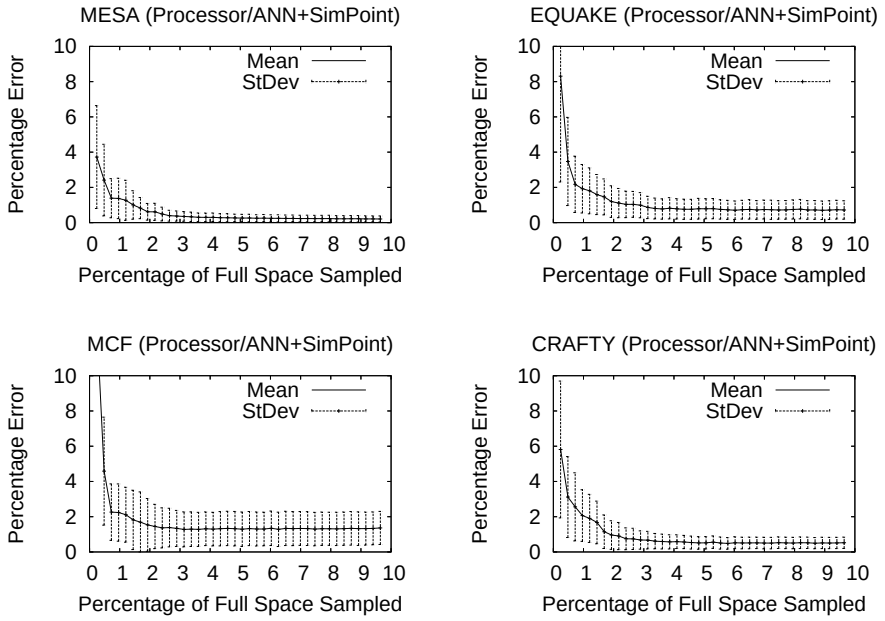

Figure 5.4: Error rates when ANN modeling and SimPoint are combined.

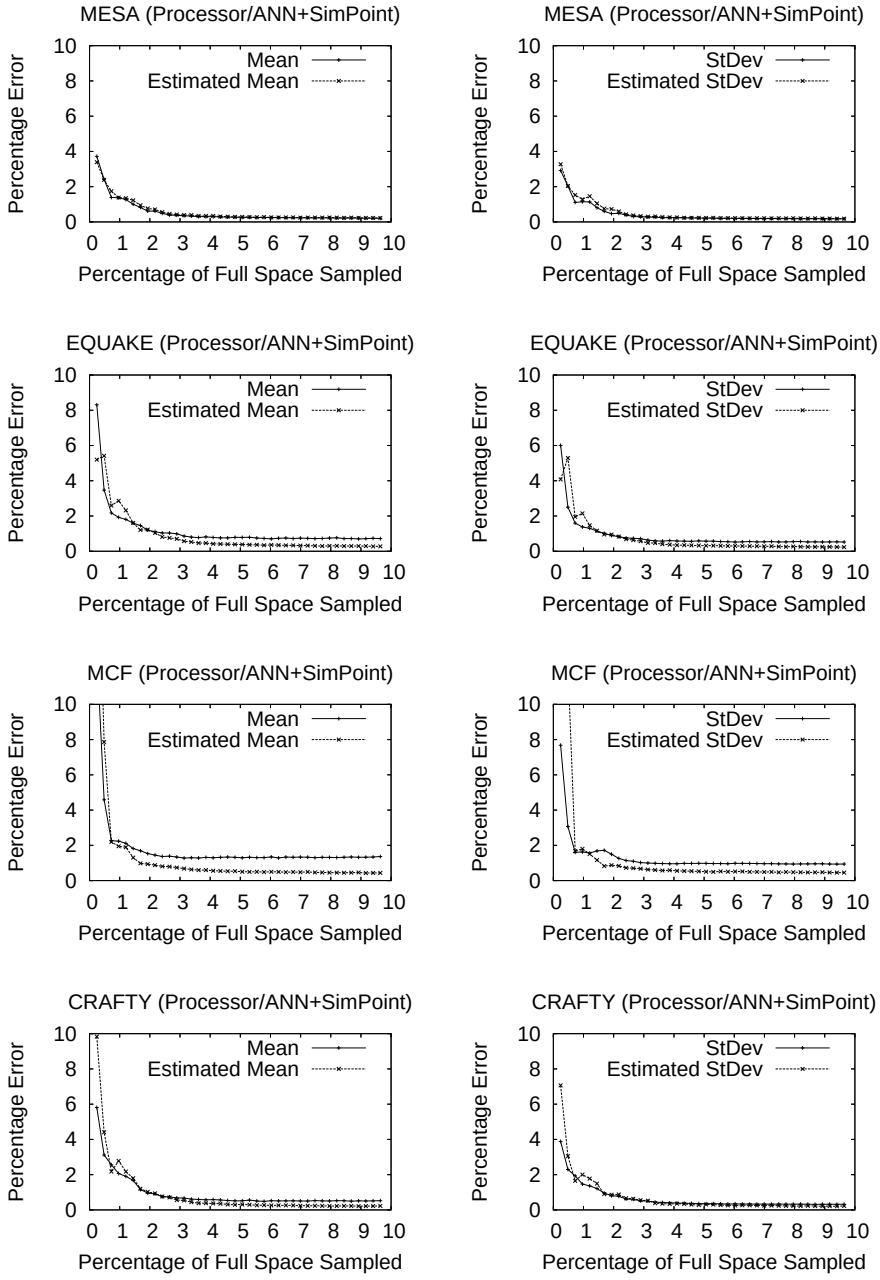

Figure 5.5: Estimated and true means and standard deviations for percentage error when ANN modeling is combined with SimPoint.

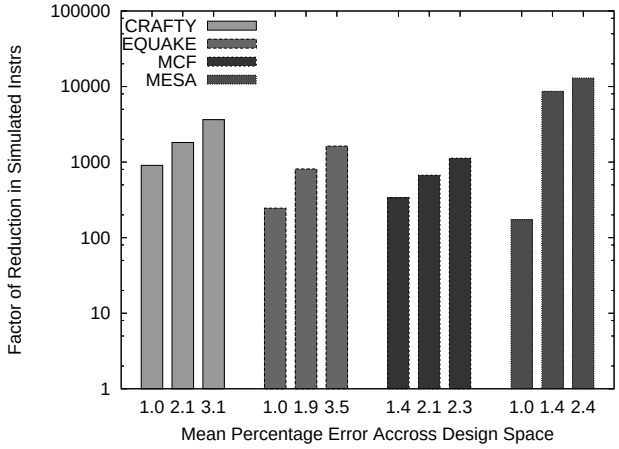

Figure 5.6: Gains from combining ANN+SimPoint.

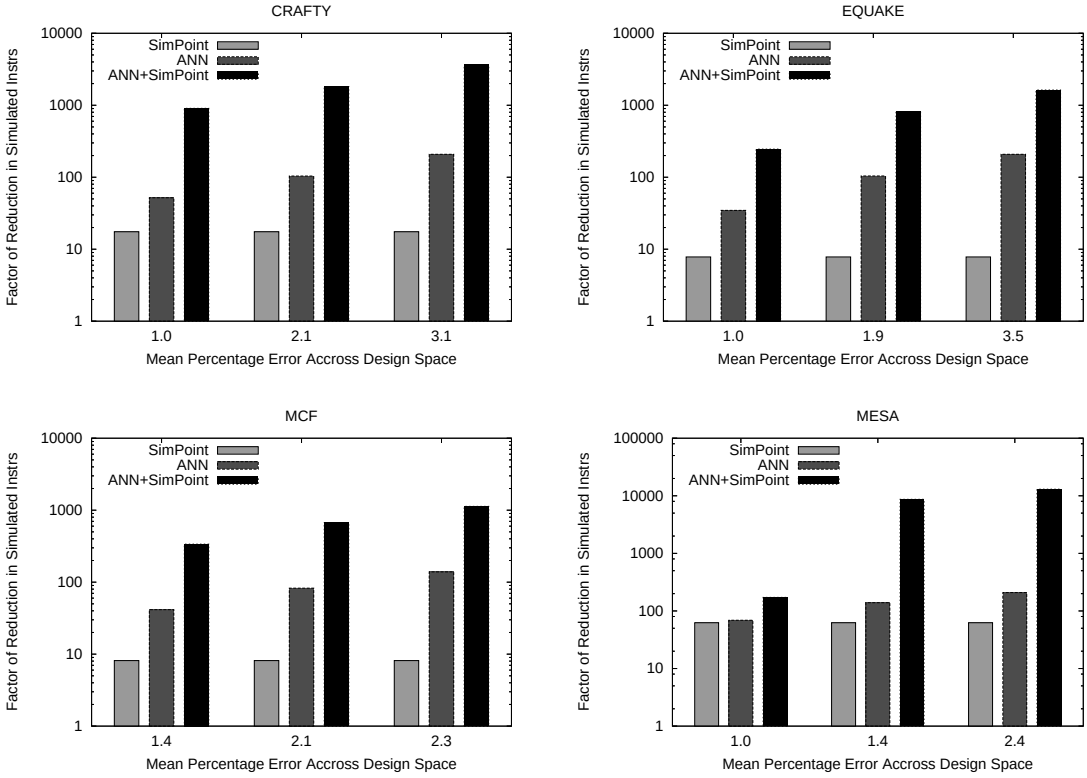

Figure 5.7: Contributions of SimPoint and ANN to total gains.

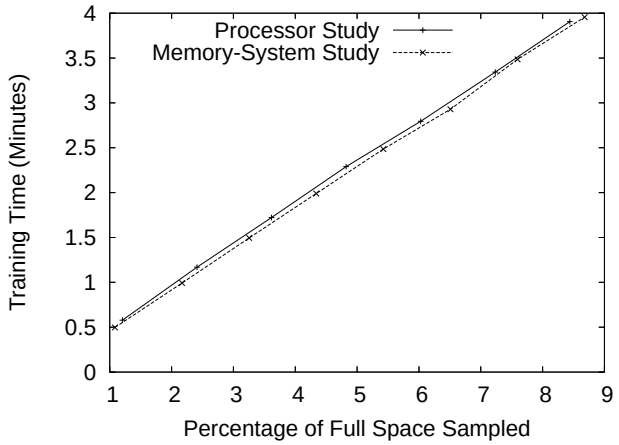

Figure 5.8: Training times.

5.4 Training Times

Our results show that the amount of simulation required to build accurate ANN models of a design space is orders of magnitude smaller than what would be required for performing a full simulation-based sensitivity study. If ANN models of large design spaces are to enable exploration of such spaces in reasonable time with reasonable computational resources, it is also critical that the time required to *train* the ANN models be *much* smaller than architectural simulation time.

Figure 5.8 shows the amount of time required to train the models for both the memory system and processor studies as a function of training set size. The networks in the 10-fold cross validation ensemble are trained in parallel on a standard cluster with 10 nodes of 3GHz Intel Pentium $4^{\textcircled{R}}$ CPUs with 1GB of DRAM. Every point in the plot represents the average of three measurements. As training set size increases from 1% to 9% of the full parameter space, training times scale linearly from 30 seconds to roughly four minutes. [1] Time required to train the models is negligible compared to architectural simulation time. Furthermore, as described in Section 3.3, simulation

[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 $O(H(I+O)PD)$.

results are collected in batches, and each round of training is amortized over multiple rounds of simulation (50 in these experiments). Learning curves presented in these studies typically level off between training set sizes corresponding to 2-4% of the full space, requiring less than two minutes' training time per every 50 simulations.

Chapter 6
Related Work

Prior work most relevant to ours includes methods to reduce input size when simulating applications; partial simulation techniques that model only a portion of an application in detail; and analytic and statistical approaches to model application behavior. These are not mutually exclusive: many techniques fall into more than one category, and techniques may be combined to reduce time spent per simulation experiment, to explore a large design space quickly, to choose design parameters of greatest importance, and to identify design bottlenecks. The ideal approach will provide performance projections for a given architecture or application and insight into the relationships among design parameters or inputs, and will be efficient to use. Unfortunately, no combination of approaches to date delivers this "holy grail."

Karkhanis and Smith [15] review work in analytical models of microprocessors, including methods for analyzing in-order pipelines [9], analytical models for determining the optimal front-end pipeline depth [11] and analytical models for expressing ILP as a function of the window size [17]. Yi et al. [28] give a thorough treatment of common approaches to architectural simulation issues, including workload design [8] and design parameter prioritization [29]. We briefly discuss approaches most relevant to ours.

Noonberg and Shen [20] take a statistical approach, using probability vectors to compose a set of components as linked Markov chain models solved using an iterative technique. Their approach yields accuracies between 2-10%, but is sufficiently complicated that modeling complex machines and large applications has not been thoroughly studied. Karkhanis and Smith [15] construct a first-order analytic model of superscalar microprocessors. Their approach is intuitive, affords insight, and delivers performance estimates with error between 5-13% with respect to detailed simulation. While intuitive, the approach is somewhat ad hoc and currently limited in the features it models. It

nonetheless provides valuable insights into both behavior of current superscalar processors and effects of long-term microarchitecture design trends.

The observation that machine structures or events may interact in at most one of two ways—serially or in parallel—leads Fields et al. [10] to define *interaction costs* (icosts) to capture interactions quantitatively. Their model provides insight into the set of events that affect an event of interest (and thus may have contributed to the triggering of that event). To measure icosts efficiently, they propose a hardware *shotgun profiler*, an augmentation to event counting registers that enables sampling execution in sufficient detail to construct a statistically representative microarchitecture graph. In the absence of appropriate hardware sampling infrastructure, computing icosts for N different sets of events requires 2^N simulations. Only computing pairwise icosts still requires a quadratic number of simulations. Nonetheless, given efficient means for gathering the required information, the technique gives new insight into (perhaps obscure) bottlenecks.

The statistical simulation approach developed by Eeckhout, et al [7] represents an attractive alternative to full simulation for many purposes. The technique first derives application characteristics (e.g., from program traces), generates a synthetic trace exhibiting those characteristics, and then simulates that trace. Statistically generated synthetic traces are orders of magnitude smaller than whole-program traces, speeding simulation time significantly. Oskin et al. [21] develop a hybrid simulator (HLS) that uses an application's statistical profiles to model instruction and data streams. HLS dynamically generates a code base and symbolically executes it on a superscalar microprocessor core, resulting in much faster experiments than possible with detailed simulation. Its average error falls within 5-7% of cycle-by-cycle simulation for a MIPS R10000 [18] processor model. Iyengar et al. [13] introduce the *R-metric* to evaluate *representativeness* of sampled, reduced traces (with respect to actual application workloads) applied to a wide class of processors. They develop a novel, graph-based heuristic to generate better syn-

thetic traces. Eeckhout et al. [6] build on this to generate statistical control flow graphs characterizing program execution, attaining better accuracy (1.8% average error on 10 SPEC CINT 2000 benchmarks) than HLS. In the SMARTS framework, Wunderlich et al. [27] select minimal subsets from instruction execution streams such that modeling those subsets yields results within desired confidence intervals. The approach can deliver high accuracies, even with small sampling intervals.

Conte et al. [4] and Haskins and Skadron [12] sample portions of application execution, performing *warmup* functional simulation before beginning detailed simulation. This attempts to create correct cache and branch predictor states for portions of the application being simulated in detail. For the large simulation intervals used in Sherwood et al.'s SimPoint [23], state warmup becomes insignificant, but is significant for other statistical techniques (e.g, SMARTS [27]) that sample detailed simulation at finer granularities.

Yi et al. demonstrate Plackett and Burman fractional factorial design [29] in prioritizing parameters for sensitivity studies. This requires $2N$ simulations to rank N parameters (they model a high and low value for each, varying parameters independently). Once the ranking between the parameters is found, a sensitivity study can be performed on the most important parameters that can be afforded with available computational resource. The approach cannot provide information on absolute parameter importance, and cannot account for many potential interactions between parameters. Instead, it provides a relative ranking between the parameters without clearly indicating the importance of less significant parameters. Nonetheless, it may profitably be used to expend computational resources on design spaces commensurately with the significance of the parameters under consideration. This approach is orthogonal to our work since a sensitivity study on the selected design parameters is still required.

Chow and Ding [3] and Cai et al. [1] apply principal component analysis and mul-

tivariate analysis to identify the most important parameters and their correlations for processor design. The energy/performance correlation analysis of Cai et al. focuses on relationships among variations of performance and energy consumption. Such analyses complement our approach and Yi et al.'s, helping choose which parameters to vary over what ranges of interest.

Chapter 7
Conclusions and Future Work

Computer architects rely on design space exploration to evaluate the impact of varying architectural parameters. Many factors have increased the time required to complete thorough design space studies, placing many such studies beyond our computational abilities. Techniques that reduce time required for individual simulations do not target the exponential number of simulations required to explore a complete design space. To attack this problem, we developed a predictive modeling approach based on artificial neural networks.

We have presented a fully automated and general mechanism to build accurate models of architectural design spaces from limited simulation results, finding that our models can predict IPC with 1-2% error, even when trained on as little as 2% of entire design spaces. Our framework allows simulation data to be collected incrementally, and estimates model accuracy reliably. Our approach is orthogonal to statistical techniques that reduce single simulation times, and we have shown that combining our approach with one particular such technique leads to 1000-13,000$\times$ reductions in the total number of simulated instructions to explore two example architectural design spaces. Furthermore, overhead for building these models is negligible compared to architectural simulation time.

We predict IPC in the studies presented here, but our approach is sufficiently general to predict other architectural statistics of interest. Our mechanism enables much faster exploration of design spaces of currently feasible sizes, and makes possible the exploration of massive design spaces outside the reach of current simulation infrastructures. Ultimately, we provide the computer architect with another tool to assist in the design of new systems and the evaluation of existing ones. In so doing, we hope to increase understanding of design choices and tradeoffs in a world of ever increasing system com-

plexity.

Several future research directions related to predictive modeling of architectural design spaces remain to be explored. One potentially promising area of research is cross-application predictive modeling. The work presented in this thesis treats the design space exploration process on distinct benchmarks as independent problems, where separate models are trained on simulation results collected on different applications. In cases where the are similarities between the benchmarks such that the same functional relationship between design parameters and metrics are observed across several applications, it could be possible to decrease sampling requirements by making the application name an input into the models and training one large model for all of the benchmarks.

Another future direction for research is the use of active learning for further reducing sampling requirements. In active learning, rather than sampling the design space randomly and training the model on these samples, one allows the model to identify which data points it would benefit most from if those data were given to it. This often increases the quality of the sampling and cuts down on the required training set sizes.

Other sophisticated methods of reducing the training set size requirements exist. Multi-task learning is one such technique that is especially well suited to architectural design spaces. At the end of a cycle-by-cycle simulation, simulators typically output several statistics in addition to the main metric of interest. For instance, an architect could be interested in studying the effect of processor parameters on IPC, and the simulator could report cache miss rates, front-side bus occupancy and branch misprediction rates in addition IPC. Although strong correlations exist between IPC and these other metrics, they cannot be used as inputs for the model because they are are unavailable prior to simulation and cannot be presented to the ANNs when a prediction needs to be obtained on a design point that is not simulated. Multi-task learning allows these correlations to be exploited without requiring the additional metrics to be available at

the time predictions are made. To do this, a large ANN with several output units is used, where the additional outputs are allocated to these correlated metrics. This ANN is then trained on sample simulation results, and correlations between the additional metrics and the main metric of interest are exploited through the sharing of the weights in the hidden layers.

Finally, a promising direction for future research involves optimizing statistical simulation techniques and ANN modeling simultaneously. The combined ANN+SimPoint results presented in this thesis are based on collecting training sets by SimPoint, where SimPoint is run out-of-the-box. It is possible to go beyond this framework by optimizing the accuracy vs. simulation-time tradeoff presented by SimPoint and the accuracy vs. number of simulations tradeoff offered by ANN modeling simultaneously.

Appendix A

The learning curves and error estimates for the remaining applications (Applu, Mgrid, Gzip, Twolf) are shown below. The results are qualitatively similar to the other applications that are discussed in Section 5.

When the training sets contain less than 1% of the full design space, the sampling is too sparse and both error rates and standard deviations are high. In this regime, the training sets do not contain enough information to capture the functional relationships between the design parameters and performance. As more data are added to the training sets through simulation, error rates and standard deviations decrease dramatically and reach asymptotes when roughly 4% of the design space is sampled. Error estimates are accurate in all cases, and are often conservative when less than 1% of the full design space is sampled. As explained in Section 5, this is due to the fact that cross-validation estimates error from the error rates of individual models in the ensemble, whereas final predictions are made by averaging the predictions of all models. Averaging typically reduces the variance in the predictions and leads to lowers the error rates, especially in cases where the sampling is sparse and variance in the predictions is significant.

As discussed in Section 5, Twolf's error rates and standard deviation fall more slowly than the other applications. Twolf also has higher variance as indicated by the slight perturbations on the learning curves. To verify that this was not an anomaly due to the specific training sets chosen through random sampling, we repeated the Twolf experiments a second time. In both cases, we got qualitatively similar results, corroborating that the observed behavior is not a consequence of the randomly sampled training points. This different in behavior is not problematic since cross-validation yields highly reliable error estimates, which allows the architect to continue simulations until acceptable error rates are obtained.

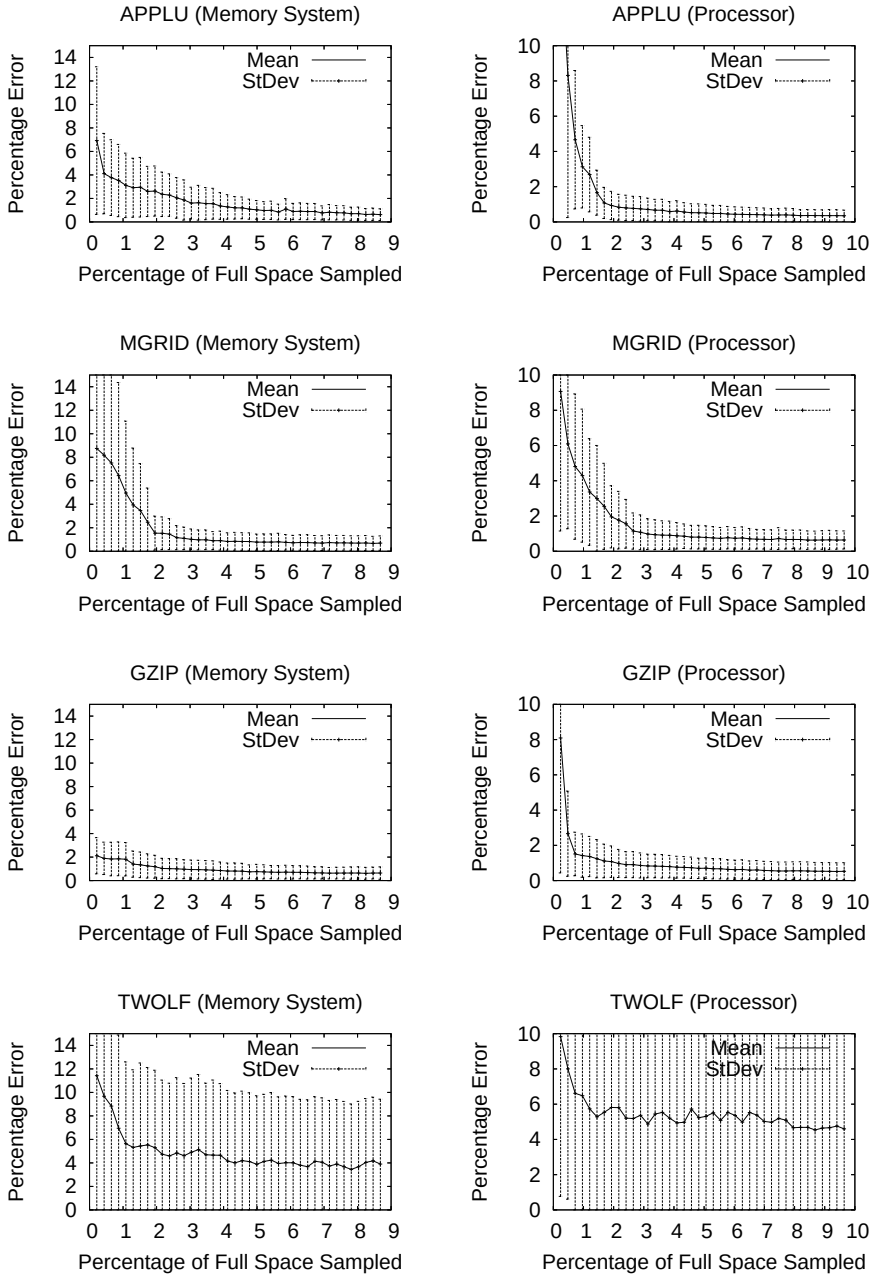

Figure A.1: Error rates of the models on the design space. The columns on the left and right show results for the memory system and processor studies, respectively.

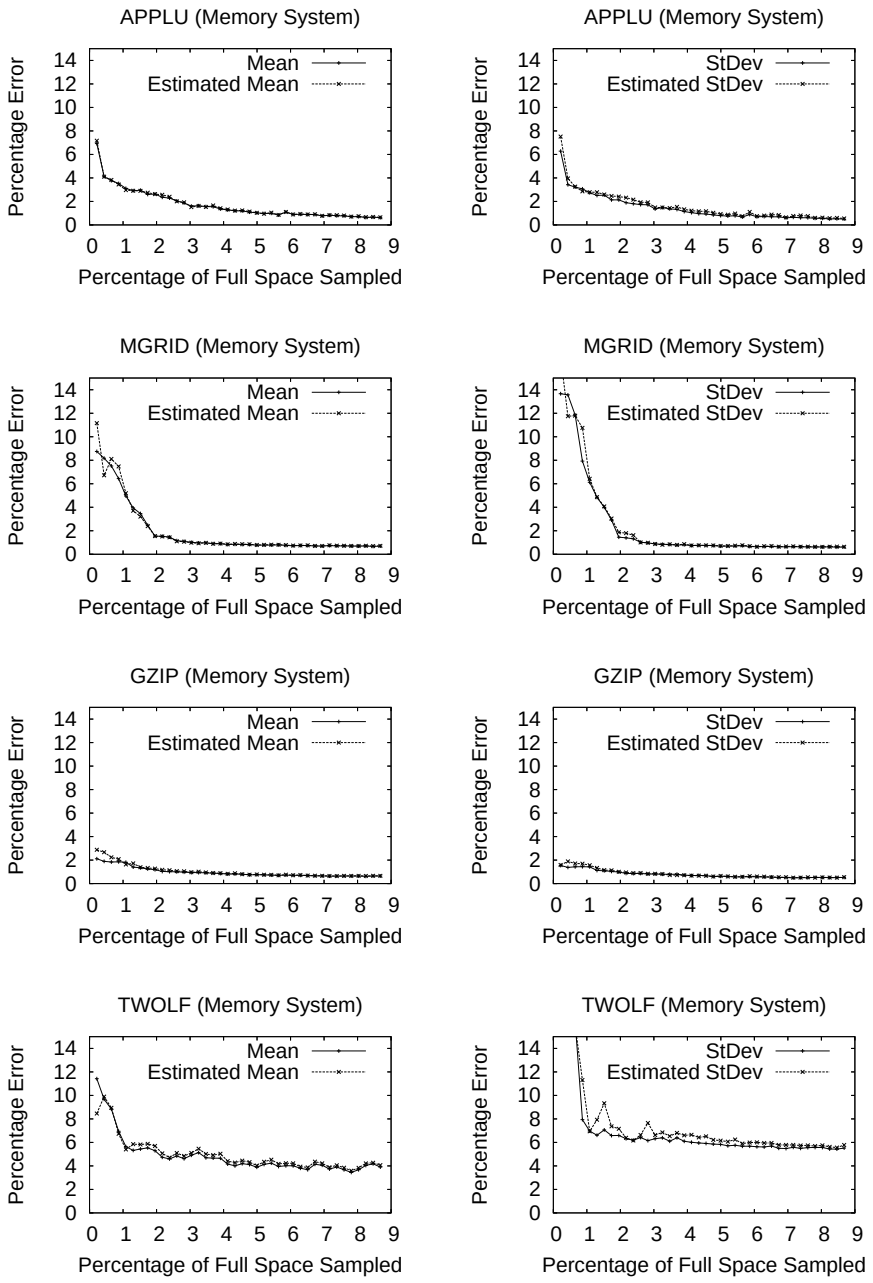

Figure A.2: Estimated and true means and standard deviations for percentage error on the memory system study.

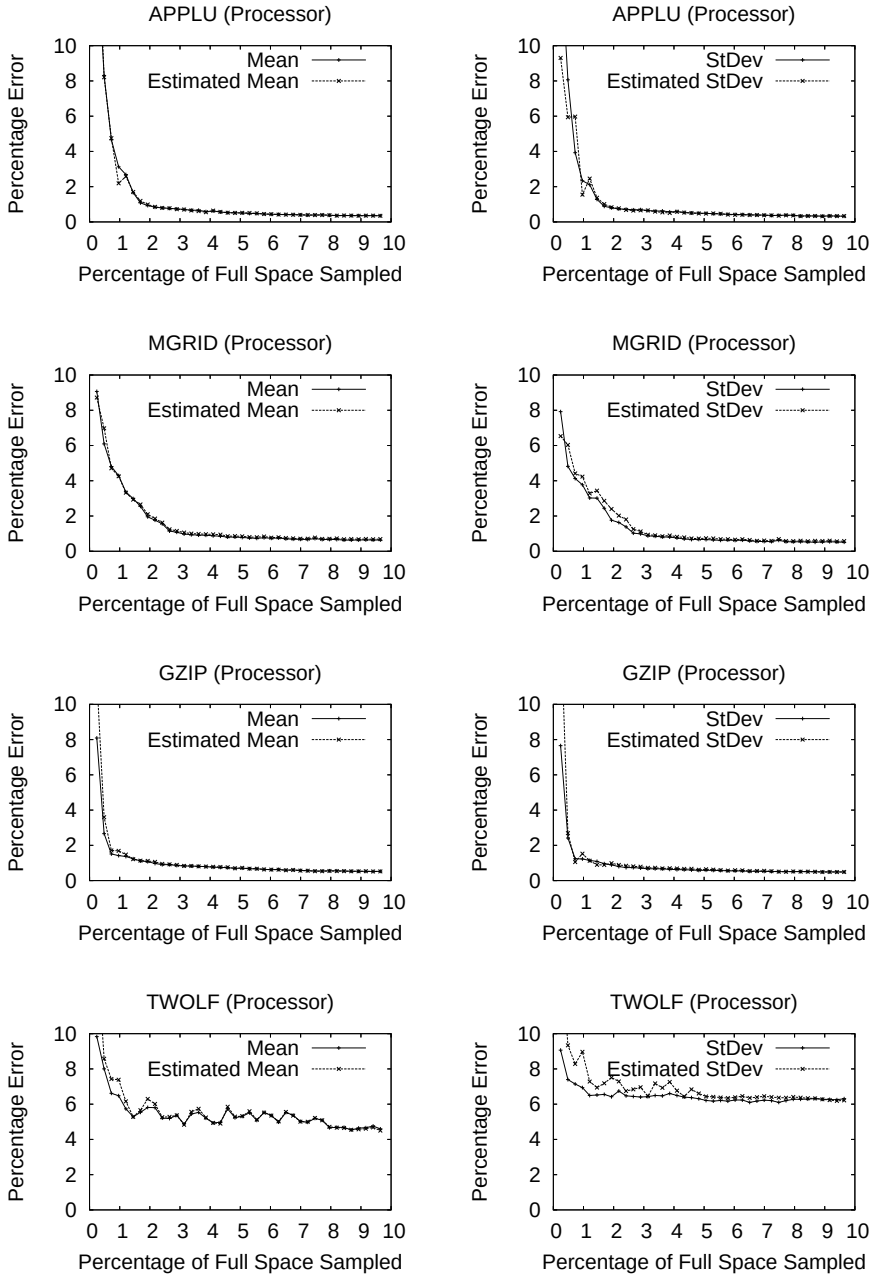

Figure A.3: Estimated and true means and standard deviations for percentage error on the processor study.

BIBLIOGRAPHY

[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 *Neural Information Processing Systems*, 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 *IEEE International Conference on Computer Design*, pages 468–477, Oct. 1996.

[5] G. Cybenko. Continuous valued neural networks with two hidden layers are sufficient. Technical Report 935, University of Illinois Urbana-Champaign Department of Electrical and Computer Engineering, Mar. 1988.

[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 *31st Annual International Symposium on Computer Architecture*, pages 350–361, June 2004.

[7] 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.

[8] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Workload design: Selecting representative program-input pairs. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 83–94, Sept. 2002.

[9] P. Emma and E. Davidson. Characterization of branch and data dependencies on programs for evaluating pipeline performance. *IEEE Transactions on Computers*, 36(1):859–875, Mar. 1987.

[10] 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.

[11] A. Hartstein and T. Puzak. Optimum pipeline depth for a microprocessor. In *29th Annual International Symposium on Computer Architecture*, pages 7–13, May 2002.

[12] J. Haskins, Jr. and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *IEEE International Conference on Computer Design*, pages 195–203, Sept. 2001.

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

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

[15] T. Karkhanis and J. Smith. A first-order superscalar processor model. In *31st Annual International Symposium on Computer Architecture*, pages 338–349, June 2004.

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

[17] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide issue superscalar processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 2–10, 1999.

[18] MIPS Technologies, Inc. *MIPS R10000 Microprocessor User's Manual, Version 2.0*, Dec. 1996.

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

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

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

[22] J. Renau. SESC. `http://sesc.sourceforge.net/index.html`, 2002.

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

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

[25] P. Szwed, D. Marques, R. Buels, S. McKee, and M. Schulz. SimSnap: Fast-forwarding via native execution and application-level checkpointing. In *the 8th IEEE Workshop on Interaction between Compilers and Computer Architectures*, Feb. 2004.

[26] 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.

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

[28] J. Yi, S. Kodakara, R. Sendag, D. Lilja, and D. Hawkins. Characterizing and comparing prevailing simulation techniques. In *11th Annual Symposium on High Performance Computer Architecture*, pages 266–277, Feb. 2005.

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