ArchRanker: A Ranking Approach to Design Space Exploration

Tianshi Chen[†] Qi Guo[‡] Ke Tang[§] Olivier Temam[♭] Zhiwei Xu[†] Zhi-Hua Zhou[♯] Yunji Chen[†]

[†] State Key Laboratory of Computer Architecture, Institute of Computing Technology (ICT), CAS, China

[‡] Carnegie Mellon University, United States

[§] University of Science and Technology of China, China

[♭] Inria, France

[♯] National Key Laboratory for Novel Software Technology, Nanjing University, China

Abstract

Architectural Design Space Exploration (DSE) is a notoriously difficult problem due to the exponentially large size of the design space and long simulation times. Previously, many studies proposed to formulate DSE as a regression problem which predicts architecture responses (e.g., time, power) of a given architectural configuration. Several of these techniques achieve high accuracy, though often at the cost of significant simulation time for training the regression models.

We argue that the information the architect mostly needs during the DSE process is whether a given configuration will perform better than another one in the presences of design constraints, or better than any other one seen so far, rather than precisely estimating the performance of that configuration.

Based on this observation, we propose a novel ranking-based approach to DSE where we train a model to predict which of two architecture configurations will perform best. We show that, not only this ranking model more accurately predicts the relative merit of two architecture configurations than an ANN-based state-of-the-art regression model, but also that it requires much fewer training simulations to achieve the same accuracy, or that it can be used for and is even better at quantifying the performance gap between two configurations.

We implement the framework for training and using this model, called ArchRanker, and we evaluate it on several DSE scenarios (unicore/multicore design spaces, and both time and power performance metrics). We try to emulate as closely as possible the DSE process by creating constraint-based scenarios, or an iterative DSE process. We find that ArchRanker makes 29.68% to 54.43% fewer incorrect predictions on pairwise relative merit of configurations (tested with 79,800 configuration pairs) than an ANN-based regression model across all DSE scenarios considered (values averaged over all benchmarks for each scenario). We also find that, to achieve the same accuracy as ArchRanker, the ANN often requires three times more training simulations.

1. Introduction

Design Space Exploration (DSE) is a largely iterative trial-and-error process guided by the intuition of the architect. At every step, the architect needs to choose among a vast set of architectural techniques and parameters values; and this is typically done using a set of simulations, which are known to be very slow, even when combined with sampling techniques.

In order to speed up that process, many previous studies have proposed to formulate DSE as a regression problem, and trained regression models (e.g., Artificial Neural Network (ANN), Support Vector Machine (SVM), linear or spline functions, etc) with simulated configurations sampled from the design space [1, 6, 7, 18, 24, 25, 26, 28, 27, 34]. The trained regression model can quickly predict the performance of unseen architecture configurations without additional simulations. By comparing the predicted performance of several architecture configurations, the architect can select the most promising next step of the DSE process.

Interestingly, all regression models have focused on predicting *the performance of any given architecture configuration* rather than predicting *which of any two architecture configurations performs best*, even though it is the latter information that mostly guides the DSE process in the end. However, the former, i.e., accurately predicting the performance of any configuration is a hard problem which requires a large amount of training simulations to achieve decent accuracy, and even that accuracy may not be sufficient to correctly predict the relative merit of two configurations. For instance, consider two configurations $\mathbf{x}_1$ and $\mathbf{x}_2$ which respectively have an IPC of 1.10 and 1.20, and a regression model that predicts $\mathbf{x}_1$ and $\mathbf{x}_2$ to respectively have IPC of 1.15 and 1.14, the model error is only 4.77%, but the model incorrectly predicts $\mathbf{x}_1$ to be a better configuration than $\mathbf{x}_2$.

Based on this observation, we formulate DSE as a *ranking* problem. We introduce a novel DSE technique, called *ArchRanker*, which trains a model to predict the relative ranking of a pair of configurations. We show that not only this

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

Q. Guo is a co-first author, whose work was partially done at ICT. Y. Chen is the corresponding author (cyj@ict.ac.cn). T. Chen, Z. Xu and Y. Chen are partially supported by the NSFC (under Grants 61100163, 61133004, 61222204, 61221062, 61303158), the 863 Program of China (under Grant 2012AA012202), the 973 Program of China (under Grant 2011CB302500), the Strategic Priority Research Program of the CAS (under Grant XDA06010403), and the 10,000 Talent Program of China. Q. Guo is partially supported by DARPA PERFECT program. K. Tang is supported by the NSFC (under Grant 61175065). O. Temam is supported by a Google Faculty Research Award, the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), and the 1000 Talent Program of China. Z.-H. Zhou is supported by the NSFC (under Grants 61333014, 61321491).

approach predicts more accurately the relative merit of two configurations than regression models which predict configurations performance, but it also requires significantly fewer simulations to be trained. Moreover, we show that this method can not only accurately predict which configuration is the best, but also estimate the relative performance between two configurations, another important element of the DSE process.

We compare ArchRanker against a state-of-the-art design space regression technique (ANN [18]) on both unicore and multicore design scenarios. On various constrained scenarios, e.g., with given power or time constraints, we find that the best configuration response predicted by ArchRanker is 1.65% to 31.77% better than the one predicted by ANN. On unconstrained scenarios, we find that ArchRanker makes 29.68% to 54.43% fewer incorrect predictions on the relative merit of two configurations than an ANN-based regression model. To achieve the same accuracy as ArchRanker, the ANN often requires three times more training simulations. Furthermore, we try to emulate an iterative design process (see Section 4.4) common in DSE practices, and we find again that ArchRanker makes 25.91% to 90.82% fewer incorrect decision than the ANN.

In summary, our main contributions are the following. First, we propose to formulate DSE as a ranking problem, and introduce a ranking-based DSE technique called ArchRanker. Second, we show that ArchRanker is generally more accurate than a state-of-the-art regression-based DSE technique at predicting the relative merit of two architecture configurations, and requires fewer simulations to achieve the same level of accuracy as the regression-based technique. Third, we emulate various DSE scenarios and we show that ArchRanker almost systematically outperforms the regression-based technique.

Section 2 describes the ranking formulation of DSE and the ArchRanker framework, Section 3 introduces the methodology, Section 4 presents our experimental results, Section 5 reviews the related work.

2. Ranking Architectural Configurations

Learning-to-rank is a machine learning paradigm which constructs a ranking model from training data, and the goal is to predict the ranking of new items. So far, learning-to-rank (ranking for short) techniques have been widely applied to real-world information retrieval applications, such as web search, recommender system and so on [29]. Ranking techniques are typically effective for problems requiring a partial or total order of items, according to an item quality metric. Architectural design space exploration is typically such a problem. In this study, we formulate DSE as a ranking problem, and design a novel ranking-based DSE framework called ArchRanker.

Figure 1 presents the ArchRanker framework. ArchRanker is composed of a training phase and a ranking phase. In the training phase, ArchRanker simulates n architecture configurations sampled from the design space. It employs a learning-to-rank technique called RankBoost [12], which significantly

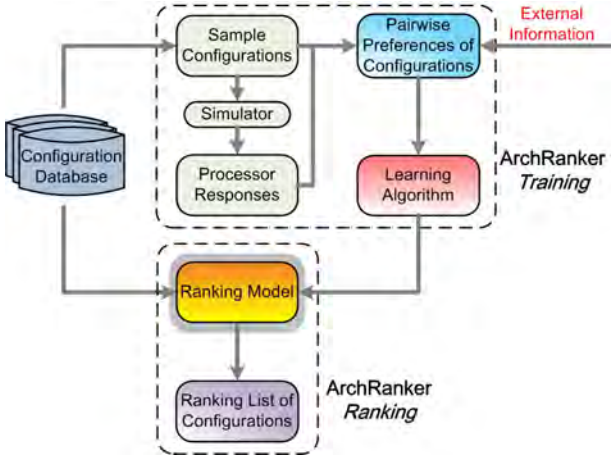

Figure 1: Framework of ArchRanker.

differs from regression techniques in the way it uses simulated configurations. Given n simulated configurations (and their metric value, e.g., execution time or power), ArchRanker converts them into $C_n^2 = n(n-1)/2$ pairs. Based on the metric value of the two configurations in a pair, a pairwise *preference* is created (i.e., which of the two configuration is preferred), and the model is trained using these preferences. Because n configurations become C_n^2 such preferences, the ranking model actually has a significantly larger training set than a traditional regression model ($n(n-1)/2$ vs. n) with the same number of simulations. In the ranking phase, ArchRanker assigns each configuration a ranking score. This score can be obtained for training configurations but more importantly, for any yet unseen configuration.

2.1. Ranking Formulation

We now present a formal description of ranking. Let $\mathscr{A}$ denote an architectural design space, $\mathbf{x} \in \mathscr{A}$ denote an architectural configuration, and $R(\mathbf{x})$ denote the metric value (also called the response) of configuration $\mathbf{x}$. In DSE, architects are given a set of simulated configurations $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ which have been ordered by their responses, i.e., $R(\mathbf{x}_1) < R(\mathbf{x}_2) < \cdots < R(\mathbf{x}_n)$ (note that if two configurations have the exact same metric value, the pair yields no preference). The set of simulated configurations induce up to $n(n-1)/2$ pairwise response preferences of configurations, which are collected in the training set $U = \{(\mathbf{x}, \mathbf{x}'); R(\mathbf{x}) < R(\mathbf{x}')\}$ for the ranking-based DSE.

Definition 1 (Ranking-based DSE) *Taking the set of pairwise preferences of configurations (U defined above) as input, ranking-based DSE constructs a ranking model (function) $H : \mathscr{A} \to \mathbb{R}$, which assigns a ranking score $H(\boldsymbol{x})$ for each configuration $\boldsymbol{x}$ in the design space $\mathscr{A}$. In ranking, configurations are sorted according to the ranking score such that $\boldsymbol{x}' \succ \boldsymbol{x}$ ($\boldsymbol{x}'$*

is ranked higher than $\mathbf{x}$) if $H(\mathbf{x}') < H(\mathbf{x})$.[1] *The architectural configuration ranked at the first place is the winner of that DSE step.*

2.2. Learning Algorithm

ArchRanker employs a representative learning-to-rank algorithm called RankBoost [12] to construct the ranking model. RankBoost is a kind of ensemble method which uses multiple base learners to create a strong learner having excellent generalization ability [45], and particularly belongs to the family of Boosting algorithms [36]. In a nutshell, RankBoost iteratively trains several base rankers $h_1, \ldots, h_T$, and provides a linear combination of these base rankers as the final ranking model H. Algorithm 1 presents the standard flow of RankBoost, whose input is a training set consisting of pairwise preferences of simulated configurations. In order to minimize the ranking error with respect to the training set, the t-th base ranker (h_t in Algorithm 1) trained at the t-th iteration of RankBoost will concentrate more on those pairwise preferences that are not correctly ranked by the former base ranker h_{t-1}. Such online adaptation is achieved by dynamically adjusting a discrete probability distribution called the weight distribution (D_t for the t-th iteration) throughout iterations.

We now elaborate how the weight distribution D_t is initialized and updated. Let S be the set consisting of sample configurations, $R(\mathbf{x})$ be the processor response of a simulated sample configuration $\mathbf{x}$, and $(\mathbf{x}, \mathbf{x}')$ be the pair of simulated configurations $\mathbf{x}$ and $\mathbf{x}'$. The initial distribution D_1 assigns a uniform probability for every configuration pair $(\mathbf{x}, \mathbf{x}')$ having the response preference $R(\mathbf{x}) < R(\mathbf{x}')$:

$$D_1(\mathbf{x}, \mathbf{x}') = \begin{cases} 1/g, & \text{if } R(\mathbf{x}) < R(\mathbf{x}'); \\ 0, & \text{if } R(\mathbf{x}) \geq R(\mathbf{x}'), \end{cases} \tag{1}$$

where $g = |\{(\mathbf{x}, \mathbf{x}') \in S \times S; R(\mathbf{x}) < R(\mathbf{x}')\}|$ is a normalization factor guaranteeing that D_1 is a correct probability distribution (i.e., $\sum_{\mathbf{x} \in S, \mathbf{x}' \in S} D_1(\mathbf{x}, \mathbf{x}') = 1$). With the setting presented in Eq. 1, RankBoost does not need to repeatedly consider the pair $(\mathbf{x}', \mathbf{x})$ if the symmetric pair $(\mathbf{x}, \mathbf{x}')$ has already been taken into account. At the end of the t-th iteration, RankBoost updates the distribution D_{t+1} with respect to each configuration pair $(\mathbf{x}, \mathbf{x}')$:

$$D_{t+1}(\mathbf{x}, \mathbf{x}') = \frac{1}{Z_t} D_t(\mathbf{x}, \mathbf{x}') \exp\Big(\alpha_t \big(h_t(\mathbf{x}) - h_t(\mathbf{x}')\big)\Big) \tag{2}$$

where Z_t is also a normalization factor guaranteeing that D_{t+1} is a correct probability distribution, and the weight α_t, defined in Algorithm 1, is the coefficient of base ranker h_t. The above rule can intuitively be explained as follows. If the t-th base ranker correctly predicts the pairwise order between simulated configurations $\mathbf{x}$ and $\mathbf{x}'$ (i.e., $h_t(\mathbf{x}) < h_t(\mathbf{x})$), then the weight assigned to the pair $(\mathbf{x}, \mathbf{x}')$, represented by $D_{t+1}(\mathbf{x}, \mathbf{x}')$, will decrease. Otherwise, if the t-th base ranker incorrectly predicts the pairwise order between simulated configurations $\mathbf{x}$ and $\mathbf{x}'$ (i.e., $h_t(\mathbf{x}) > h_t(\mathbf{x}')$), then the weight will increase. Owing to this updating rule, the training of the next base ranker h_{t+1} will put less emphasis on correctly ordered pairs, and concentrate more on incorrectly ordered pairs.

At the t-th iteration, RankBoost trains the base ranker h_t based on the weight distribution D_t such that h_t can concentrate more on pairwise preferences that cannot be correctly ranked by the former base ranker h_{t-1}. Following the standard setting of RankBoost, each base ranker is a simple learning model called the *decision stump* [12, 17], which is mathematically defined as the following step function:

$$h(\mathbf{x}) = \begin{cases} 1, & \text{if } f_i(\mathbf{x}) > \theta; \\ 0, & \text{otherwise}, \end{cases} \tag{3}$$

where $f_i(\mathbf{x})$ represents the i-th design parameter of the configuration $\mathbf{x}$, and θ is a constant threshold. To train a base ranker, we iteratively search for the best combination of feature index i and threshold θ following Freund *et al.* [12]. After T iterations of RankBoost which train T base rankers (decision stumps) $h_1, \ldots, h_T$ respectively, the final ranking model H

$$H(\mathbf{x}) = \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}) \tag{4}$$

is ready to assign any input configuration $\mathbf{x}$ a ranking score such that different design configurations can be ranked according to their ranking scores. The combination of multiple base rankers (decision stumps) is capable of representing non-linear mappings [36, 45].

Note that, because ArchRanker uses relative information for training, i.e., preferences between configurations, it is possible to leverage external "hints". Such hints can carry expert knowledge, e.g., "the ratio of cache size to cache block size should not drop below value X", which can increase the training set size at no simulation cost (e.g., the configuration complying the hint may be better than the one violating it), and thus improve the ranking accuracy. We leave the addition of such expert knowledge for future work.

2.3. Constrained DSE

So far, we have described how ArchRanker can predict which of two configurations is the best. This is sufficient to find the best possible configuration within a given design space. However, DSE is often *constrained*. For instance, one sets a given power constraint, and attempts to find the configuration which achieves the best possible execution time (or lowest possible area, or a pareto combination of both) and still satisfies the power constraint.

[1] When we consider the execution time or power of architectural configuration, we would prefer a smaller ranking score, which is the definition presented here. When we consider the IPC of architectural configuration, we would prefer a larger ranking score, thereby we inversely define $\mathbf{x} \succ \mathbf{x}'$ ($\mathbf{x}$ is ranked higher than $\mathbf{x}'$) if $H(\mathbf{x}') < H(\mathbf{x})$.

Algorithm 1: Training a Ranking Model with RankBoost

Input S: set of sample configurations;
Input T: number of training iterations;
D_t ($t = 1, 2, \ldots$): distribution (over $S \times S$) for t-th iteration (Initial distribution D_1 is specified by Eq. 1);
α_t ($t = 1, 2, \ldots$): control parameter for the t-th iteration;
begin
  for $t = 1, \ldots, T$ **do**
    Train a base ranker $h_t(\mathbf{x})$ with distribution D_t;
    $\alpha_t = \frac{1}{2} \ln\left(\frac{1+r_t}{1-r_t}\right)$;
      // r_t is an auxiliary parameter having value $\sum_{\mathbf{x},\mathbf{x}'} D_t(\mathbf{x},\mathbf{x}') \exp\Big(\big(h_t(\mathbf{x}') - h_t(\mathbf{x})\big)\Big)$.
    $\forall (\mathbf{x},\mathbf{x}') \in S \times S$:
    $D_{t+1}(\mathbf{x},\mathbf{x}') = D_t(\mathbf{x},\mathbf{x}') \exp\Big(\alpha_t\big(h_t(\mathbf{x}) - h_t(\mathbf{x}')\big)\Big)/Z_t$;
      // Z_t refers to the normalization factor $\sum_{\mathbf{x},\mathbf{x}'} D_t(\mathbf{x},\mathbf{x}') \exp\Big(\alpha_t\big(h_t(\mathbf{x}) - h_t(\mathbf{x}')\big)\Big)$.
  end
  Output the final ranking model: $H(\mathbf{x}) = \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})$;
end

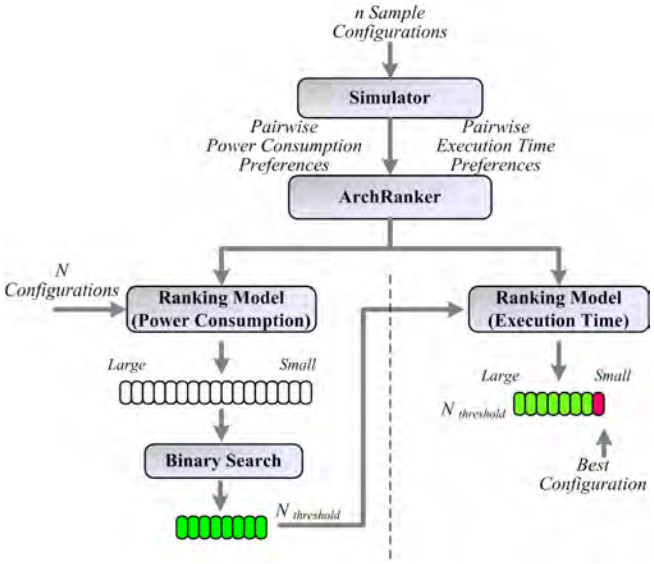

Figure 2: Constrained DSE.

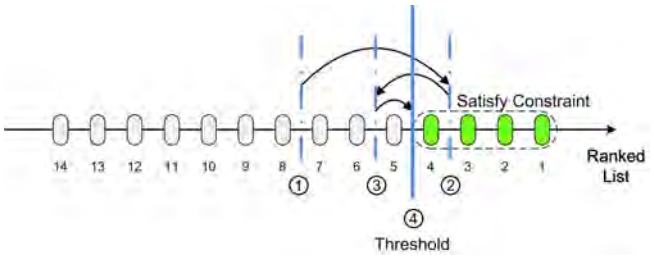

Figure 3: Binary search.

For that purpose, ArchRanker uses a two-step process, see Figure 2. Let us consider again the above example of minimizing the execution time within a given power budget, e.g., 10W. We assume that ArchRanker has n (e.g., $n = 100$) training configurations, and we wish to explore about N (e.g., $N = 100,000$) configurations. We first build a power model (response = power) using ArchRanker, and use it to rank the N configurations. We now need to find which among these N configurations satisfy the 10W constraint. For that purpose, we use a simple binary search coupled with a few additional simulations (we call $n_{constraint}$ the number of additional simulations): we pick the median point of the ranked list (the one which divides the list into two parts of the same size), simulate it to get its power, and thus decide which part of the list we keep, i.e., which part contains the configurations which will satisfy the constraint. And we recursively iterate the process until the list contains only two configurations, one with a power slightly less than or equal to 10W, and one with a power slightly greater than 10W (or an empty set if no configuration satisfies the power constraint), see Figure 3; the former is the threshold configuration. At most, ArchRanker needs $n_{constraint} = \lfloor log_2(N) \rfloor + 1$ configurations, e.g., 17 for $N = 100,000$ configurations, or 20 for $N = 1,000,000$ configurations.

Then, ArchRanker discards all configurations (among the N-configuration design space) with a worse ranking than the threshold configuration, and keeps the remaining $N_{threshold}$ configurations. Using the n training configurations, it builds a second ranking model, but for the execution time response now. It then feeds all $N_{threshold}$ configurations to this new model, and selects the top configuration as the best one (best execution time within the power constraint).

2.4. Response Gap

While it is essential to know which among two configurations is the better one for the DSE process, it is not always a sufficient information. The architect may also want to know how much the better configuration improves performance over the configuration previously selected in the DSE process. A rough estimate is sufficient to decide whether the increment is significant enough to shift to the new configuration.

In this section, we show how ArchRanker can be used for that purpose as well. The principle is to use the simulated points in the training set as references to estimate the response gap between two configurations. ArchRanker also uses this approach to decide whether the training set is sufficient to estimate the response gap, or if it is necessary to trigger new simulations.

Consider two yet unseen configurations $\mathbf{x}$ and $\mathbf{x}'$ between which we want to estimate the response gap $R(\mathbf{x}) - R(\mathbf{x}')$. Without loss of generality, we assume that $H(\mathbf{x}') < H(\mathbf{x})$ (recall $H(\cdot)$ is the ranking function given by ArchRanker). Note that when ArchRanker correctly predicts the pairwise preferences between $\mathbf{x}'$ and $\mathbf{x}$, $R(\mathbf{x}') < R(\mathbf{x})$ also holds (recall $R(\cdot)$ is the response).

We first rank all *simulated* configurations $\{y_i, 1 \leq i \leq n\}$. Then, we select configuration $\mathbf{y}_k$ such that $H(\mathbf{y}_k) < H(\mathbf{x})$ and it minimizes the difference $H(\mathbf{x}) - H(\mathbf{y}_k)$. Similarly, we select

Table 1: Unicore Design Space.

Parameters	Values	Number
Fetch/Commit Width	2,4,8	3
FP Unit	2,4,6,8	4
ALU	2,4,6,8	4
L1 ICache	1,2,4,8,16,32KB	6
L1 DCache	1,2,4,8,16,32KB	6
L2 UCache	256-4096KB: 2*	5
ROB Size	16-256: 16+	16
LSQ Size	8-128: 8+	16
GShare Size	1,2,4,8,16,32K	6
BTB Size	512-4096: 2*	4
Total Number		53,084,160

$\mathbf{y}_{k'}$ such that $H(\mathbf{x}') < H(\mathbf{y}_{k'})$ and it minimizes the difference $H(\mathbf{y}_{k'}) - H(\mathbf{x}')$.

As a result, $H(\mathbf{y}_k) - H(\mathbf{y}_{k'}) < H(\mathbf{x}) - H(\mathbf{x}')$, which yields $H(\mathbf{y}_k) + H(\mathbf{x}') < H(\mathbf{y}_{k'}) + H(\mathbf{x})$. When ArchRanker correctly ranks pairs $(\mathbf{y}_k, \mathbf{y}_{k'})$ and $(\mathbf{x}, \mathbf{x}')$, the above inequality implies that $R(\mathbf{y}_k) + R(\mathbf{x}') < R(\mathbf{y}_{k'}) + R(\mathbf{x})$. In other words, $R(\mathbf{y}_k) - R(\mathbf{y}_{k'})$ is a lower bound of $R(\mathbf{x}) - R(\mathbf{x}')$, i.e., the response gap between $\mathbf{x}$ and $\mathbf{x}'$.

3. Methodology

In this section, we introduce the platform and experimental methodology of our study.

3.1. Simulator

Simulations are conducted on a cycle-accurate in-house simulator of a MIPS-compatible commercial processor which has been validated against both RTL design and manufactured chips at 65nm; the baseline characteristics of the architecture are shown in bold in Table 1.

The unicore simulator supports out-of-order execution, register renaming, dynamic scheduling, and branch prediction. The load/store queue of the core supports dynamic memory disambiguation, out-of-order memory accesses, non-blocking cache, and load speculation. The multicore simulator is built upon the aforementioned processor simulator. It adopts a static Non-Uniform Cache Architecture, and its cache coherence is maintained with a directory-based MSI protocol. The memory consistency model is a variation of the weak consistency model. The cores are connected using a 2D mesh Network-on-Chip (NoC), where each hop takes 2 router pipeline stages and 1 link traversal stage.

We measure both unicore and multicore performance using execution time (in milliseconds). We also measure power using Wattch [2] for processor components, and CACTI [39] for caches.

3.2. Design Space

We evaluate ArchRanker over both unicore and multicore design spaces. The unicore design space contains 10 superscalar processor parameters, see Table 1, for a total of 53,084,160 configurations. The multicore design space contains 9 parameters, see Table 2, for a total of 604,800 configurations. We uniformly sampled 500 configurations out of each design space, for a total of 1000 architecture configurations.

Table 2: Multicore Design Space.

Parameters	Values	Number
Issue Width	1-4: 2*	3
Core No.	1-8: 2*	4
SMT Context	1-4: 2*	3
Memory Bandwidth	8-64GB/s: 8+	8
Frequency	1-4GHz: 0.5+	7
L2 Size	1-16MB: 2*	5
L2 Block Size	32-128B: 2*	3
L2 Ways	1-16Way: 2*	5
L2 MSHRs	32-256: 2*	4
Total Number		604,800

Table 3: Single-threaded programs from SPEC CPU2006.

Program	*Type*	*Remarks*
bwaves	CFP2006	Fluid Dynamics
bzip2	CINT2006	Compression
gcc	CINT2006	C Compiler
gobmk	CINT2006	Artificial Intelligence: Go
gromacs	CFP2006	Biochemistry / Molecular Dynamics
hmmer	CINT2006	Search Gene Sequence
lbm	CFP2006	Fluid Dynamics
leslie3d	CFP2006	Fluid Dynamics
libquantum	CINT2006	Physics / Quantum Computing
milc	CFP2006	Physics / Quantum Chromodynamics
sjeng	CINT2006	Artificial Intelligence: chess
zeusmp	CFP2006	Physics / CFD

3.3. Benchmarks

For the unicore design space, we use 12 benchmarks from the SPEC CPU2006 suite, see Table 3. The benchmarks are compiled using GCC 4.3.0 and the -O3 optimization level, and they are executed using *reference* input sets. We used SimPoint [35, 31] to reduce the simulation time of these benchmarks. For the multicore design space, we use 6 multi-threaded benchmarks from the SPLASH-2 suite [42]. They are explicitly parallel, shared-memory programs written using the Pthread library; their inputs are described in Table 4. No particular criterion led to the selection of the above benchmarks, except for the time required to perform all training simulations, and simulation issues for some of the benchmarks.

We simulate each benchmark on the 500 architecture configurations of the corresponding design space, for a total of 9,000 simulations ($500 \times 12 + 500 \times 6$).

Table 4: Multi-threaded programs from SPLASH-2.

Program	*Input Size*
barnes	16384 particles
fft	65536 data points
lu	512×512 matrix, 16×16 blocks
ocean	256×256 grid
radix	1048576 integers
water-nsquared	512 molecules

3.4. Experiments

We compare ArchRanker against an Artificial Neural Network (ANN), a state-of-the-art regression-based DSE technique [19]. Unless otherwise specified, the ANN model uses the same numbers of simulated configurations for training and testing as ArchRanker does. We have used the ANN parameters of Ïpek *et al.* [19] (16-unit hidden layer). For a thorough comparison, we have also retrained two other ANN configurations: we have increased the number of neurons in the hidden layer until it provides no additional benefit and obtained a 20-neuron hidden layer, and we did the same for a 2-layer ANN and obtained 16-neuron + 4-neuron hidden layers. In each graph, we only present one bar for the ANN results, but we are careful to select the configuration which performs best among the three aforementioned ones.

For each experiment, unless otherwise specified, we randomly split the 500 simulated configurations available for each benchmark into two parts, 100 configurations for *training* the ranking model (which will be converted into $C^2_{100} = 4,950$ pairwise preferences), and the remaining 400 configurations for *testing*. For each experiment, we repeat the training 10 times to reduce the statistical bias, changing the training/testing groups every time. The ArchRanker training process is iterative, and we use $T = 200$ iterations, see Algorithm 1. At most, the trained ranking model is evaluated over $C^2_{400} = 79,800$ testing pairs corresponding to the 400 simulated configurations.

4. Experimental Evaluation

We evaluate ArchRanker under different DSE scenarios, and according to different metrics. Whenever possible, we compare ArchRanker against an ANN-based regression model.

4.1. Constrained DSE

In this section, we emulate the typical constrained DSE case presented in Section 2.3. We consider three scenarios for both unicore (U) and multicore (M): optimizing execution time under a power constraint (Time/Power), optimizing execution time under both execution time and power constraints (Time/Time+Power), and optimizing EDDP under both execution time and power constraints (EDDP/Time+Power), i.e., six scenarios in total; for instance, the (Time/Time+Power) scenario for multi-cores is denoted M-Time/Time+Power. Note that EDDP corresponds to $energy \times time^2$; this metric has been used in several recent studies on regression techniques [8, 26, 27].

For each benchmark in each scenario, we train ArchRanker using $n = 100$ simulated configurations, and due to the $n_{constraint}$ additional simulations required by the binary search, see Section 2.3, we train the ANN using $n + n_{constraint}$ configurations for the sake of fairness.

In order to set challenging constraints for each scenario, we use the training set to decide the constraint metric values as follows: for each metric m (e.g., power, execution time), we find m_{min} and m_{max} within the training set, and we set the constraint at the lowest k%, i.e., $\frac{k}{100} \times (m_{max} - m_{min}) + m_{min}$ (i.e., $k = 100$ is m_{max}, the easiest constraint, and $k = 0$ is m_{min}, the hardest constraint), with the implicit assumption that the randomly generated training configurations are somewhat representative of the global design space. For scenarios having a single constraint (e.g., U-Time/Power and M-Time/Power), we use $k = 20$. For scenarios with two constraints (e.g., M-Time/Time+Power), we use $k = 40$ for both constraints because setting a too stringent constraint would too severely reduce the number of eligible configurations.

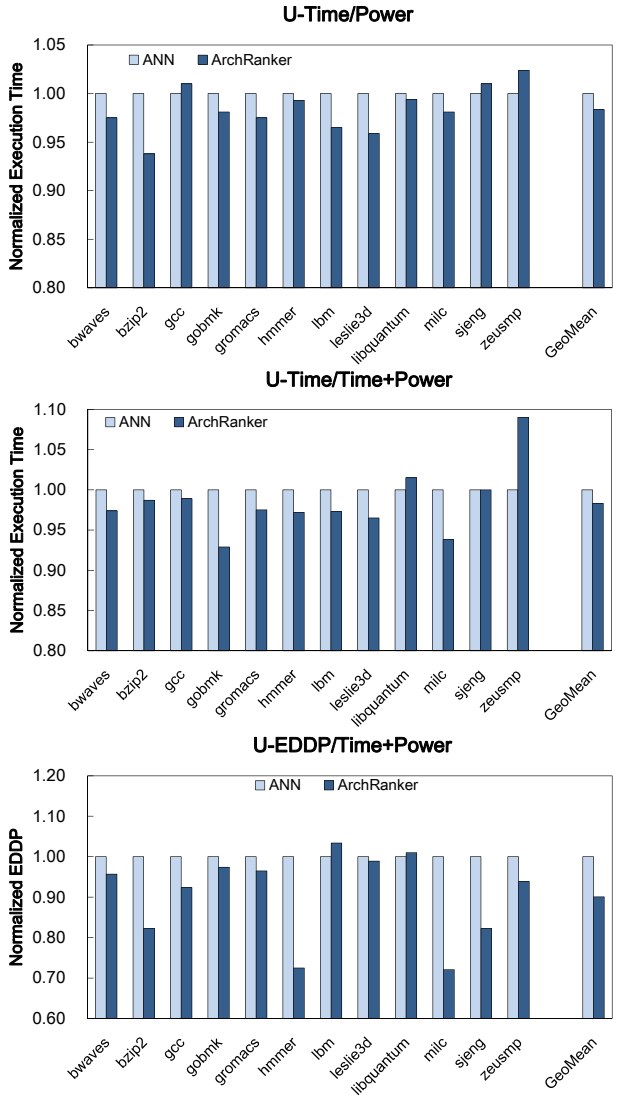

Figure 4: Performance comparison of ArchRanker and ANN; for both time and EDDP, the values are normalized to the ANN results, used as the baseline; recall that the ANN results are the best obtained across the three ANN configurations.

For any scenario $U - m_1/m_2$ or $M - m_1/m_2$, we compare ArchRanker against ANN using the primary metric, i.e., m_1,

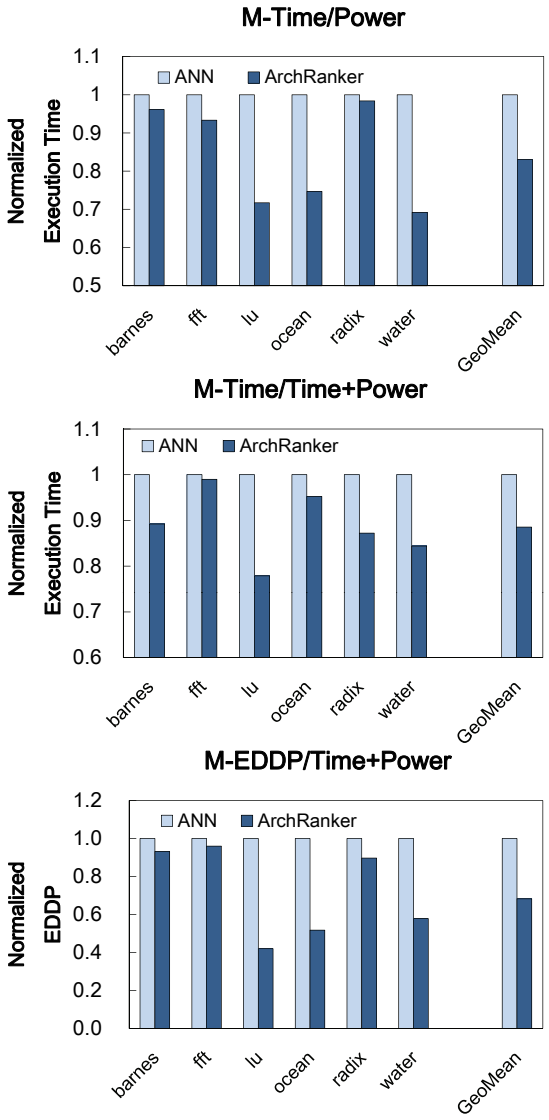

Figure 5: Performance comparisons of configurations found by ArchRanker and ANN under constrained multicore scenarios. On each benchmark, the result of ArchRanker is normalized to ANN.

not the constraint metric m_2. For instance, for a scenario where we seek the best possible time under power constraints on unicores, i.e., U-Time/Power, we compare the *Time* of the best solutions selected by ArchRanker and ANN. The results are shown in Figure 4 for unicore scenarios, and Figure 5 for multicore scenarios.

ArchRanker outperforms ANN (recall we use the best of the three ANNs, see Section 3) on most benchmarks for unicore time scenarios, i.e., U-Time/Power and U-Time/Time+Power. However, the best solution found by ArchRanker is only 1.65% and 1.68% better than the one found by ANN on average. We observed that, on unicore experiments, execution time values aggregate into a few clusters only within the design space, though not the power values. For instance, increasing

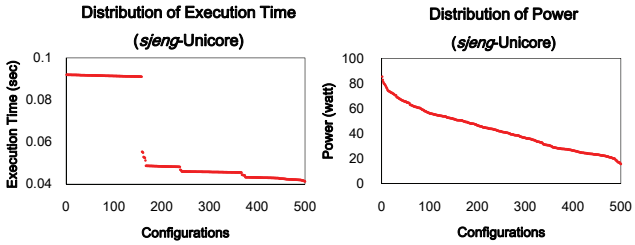

Figure 6: Execution time and power of 500 simulated unicore configurations on *sjeng*, ranked in descending order.

cache size may bring no benefit until a threshold corresponding to the reuse distance of an array is reached; on the other hand, increasing cache size always results in a proportional increase of power. In order to illustrate this clustering phenomenon, we plot the 500 configurations for benchmarks *sjeng* (unicore) for both time and power, in Figure 6. Due to this clustering property, as long as ArchRanker or ANN locates a configuration which resides in the cluster corresponding to the best execution time, its result will not be too far away from the optimum, and only small variations can be expected within the cluster. On the other hand, the configurations are significantly more spread out for unicore EDDP scenarios, and ArchRanker significantly outperforms ANN on many benchmarks of the U-EDDP/Time+Power scenario, by 9.92% on average.

ArchRanker outperforms ANN on all benchmarks of all M-Time/Power, M-Time/Time+Power and M-EDDP/Time+Power scenarios by respectively 16.98%, 11.44%, and 31.77%. For *water*-M-Time/Power, the execution time of the best configuration found by ArchRanker is 30.79% lower than the one found by ANN. On *lu*-M-EDDP/Time+Power, the EDDP of the best configuration found by ArchRanker is about 58.02% lower than the one found by ANN.

4.2. Unconstrained DSE

In this section, we consider unconstrained DSE: we evaluate how ArchRanker performs on any possible pair of the 400 test configurations, i.e., 79,800 pairs in total for both unicores/multicores and time/power scenarios. We compare the performance of ArchRanker against ANN in terms of the fraction of incorrect pairwise predictions, i.e., which configuration of a pair is the best, see Section 2.1. As explained in Section 3, for each architecture (unicore or multicore), we split the 500 simulated configurations into 100 training configurations, and 400 test configurations. We repeat the process 10 times, and the results presented in Figure 7 are averaged over these 10 experiments.

ArchRanker outperforms ANN for all benchmarks of the unicore time scenario, and has respectively 38.13% fewer incorrect predictions (fraction of incorrect predictions: 0.0931 vs. 0.1505) on average. The same goes for the multicore time and power scenarios where ArchRanker has respectively 54.43% (fraction: 0.0362 vs. 0.0796) and 46.22% (fraction: 0.0411 vs. 0.0764) fewer incorrect predictions on av-

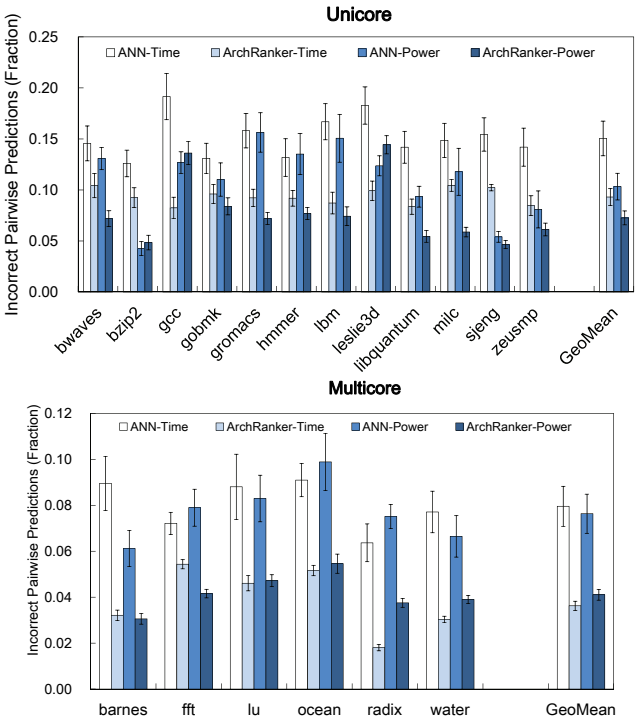

Figure 7: Incorrect pairwise predictions made by ArchRanker and ANN.

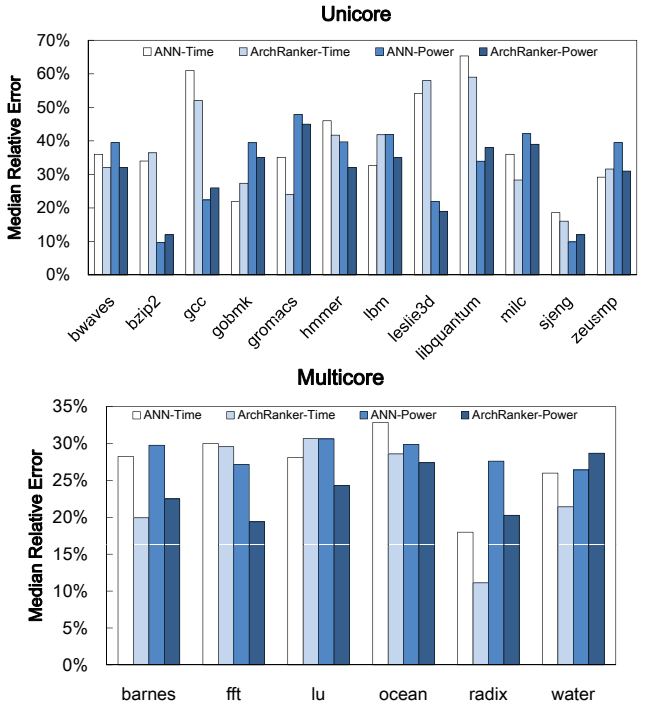

Figure 8: Median relative error of the predicted response gap by ArchRanker and ANN.

erage. Only for 3 benchmarks of the unicore power scenario, ANN provides more correct predictions than ArchRanker, though ArchRanker still outperforms ANN by 29.68% (fraction: 0.0726 vs. 0.1033) on average. For instance, for benchmark *barnes* of the multicore time scenario, ArchRanker has 64.13% fewer incorrect predictions than ANN.

Response Gap. During the aforementioned experiments, we also record how well ArchRanker predicts the response gap between two configurations, i.e., not only whether a configuration is better than another, but also by how much, see Section 2.4. We also compare against the response gap predicted by ANN. For each pair of testing configurations, we measure the relative error made by ArchRanker or ANN, i.e., $\frac{|Predicted\_Gap - Real\_Gap|}{Real\_Gap}$, and we report the median relative error in Figure 8. Note that we use *median* instead of *average* relative error because *Real_Gap* can be close to 0 (two configurations with same response), and thus the average relative error would become artificially big; the median error is the value separating the lower from the higher half of the set of all values. We observe in Figure 8 that both techniques can only provide a rough estimate of the response gap, though it is often sufficient to decide whether a configuration is potentially worth simulating or not. Even though ANN models are designed to estimate the time/power responses of a given configuration, we can observe that the median relative error of ANN is worse than ArchRanker for most benchmarks (we do not report mean values as a "mean of median errors" is an awkward statistical metric). ArchRanker outperforms ANN on 7 out 12 benchmarks for the unicore time scenario, on 8 out 12 benchmarks for the unicore power scenario, on 5 out of 6 benchmarks for the multicore time scenario, and on 5 out of 6 benchmarks for the multicore power scenario.

4.3. Number of Training Simulations

We now want to evaluate how many training simulations ArchRanker and ANN respectively need to achieve the same level of ranking accuracy (pairwise prediction accuracy). In order to maximize the number of training simulations we can consider, we use the unconstrained DSE scenario of Section 4.2. We proceed as follows. For each benchmark, we train ArchRanker with 100 randomly selected simulated configurations. Then we train ANN with the same 100 configurations on the same benchmark. If the pairwise prediction accuracy of ANN is lower than that of ArchRanker, we randomly select 10 more configurations among of the 500 simulated configurations available, see Section 3, and we train again the ANN model (recall that we actually train three ANN models and use the best one, see Section 3). We compare the new ANN model against ArchRanker (still trained on 100 configurations only) on the remaining $500-(100+10)=390$ configurations, and we repeat the process (using increments of 10 configurations) until the ANN model achieves the same accuracy as ArchRanker, or until there are fewer than 100 configurations left for testing. We call N_{train} the total number of training configurations used at any step, so $500-N_{train}$ configurations are used for testing (naturally, both ArchRanker and ANN are tested using the same configurations for the sake of fair-

Table 5: Numbers of simulated configurations (training examples) required by ANN to achieve the same level of prediction accuracy as ArchRanker. ArchRanker uses 100 simulated configurations for training.

Unicore Design Scenarios		
Program	**Unicore-Time**	**Unicore-Power**
bwaves	230	250
bzip2	260	110
gcc	400+	110
gobmk	230	400+
gromacs	390	400+
hmmer	290	390
lbm	400+	270
leslie3d	400+	100
libquantum	400+	370
milc	400+	290
sjeng	400+	100
zeusmp	380	260
AVG	*348+*	*254+*

Multicore Design Scenarios		
Program	**Multicore-Time**	**Multicore-Power**
barnes	400+	400+
fft	250	400+
lu	300	400+
ocean	400+	400+
radix	400+	400+
water-nsquared	400+	400+
AVG	*358+*	*400+*

ness). Using 10 repeated trials, we statistically compare ANN against ArchRanker; both ArchRanker and ANN are retrained on every trial, with a new set of 100 original configurations (but N_{train} in total for ANN). We use the Wilcoxon rank sum test [41] to check whether the ANN model has a comparable ranking accuracy with ArchRanker; the test assigns a *significance* to the comparison. If ArchRanker still significantly outperforms ANN at the significance level of 0.05, we keep increasing N_{train} by 10. Otherwise, we stop the process and record N_{train}. Note that this process is rather optimistic for ANN because the process is stopped as soon as the advantage of ArchRanker over ANN is no longer overwhelming, i.e., significance less than 0.05.

We report the results in Table 5. On average, the ANN regression model requires $248_{+}\%$, $154_{+}\%$, $258_{+}\%$, and $300_{+}\%$ *more* simulated configurations than ArchRanker on respectively unicore time, unicore power, multicore time, and multicore power scenarios, in order to achieve a comparable level of prediction accuracy; note that the "+" means that for some benchmarks, ANN could not achieve the level of accuracy of ArchRanker after exhausting all available training configurations. Overall, ArchRanker can save a considerable amount of training simulations. Beyond the training time benefit, this result corroborates the intuition at the root of ArchRanker that the ranking problem is significantly less difficult than the performance value prediction problem, while still being more aligned with the preoccupation of the architect during the DSE process.

4.4. Iterative DSE Process

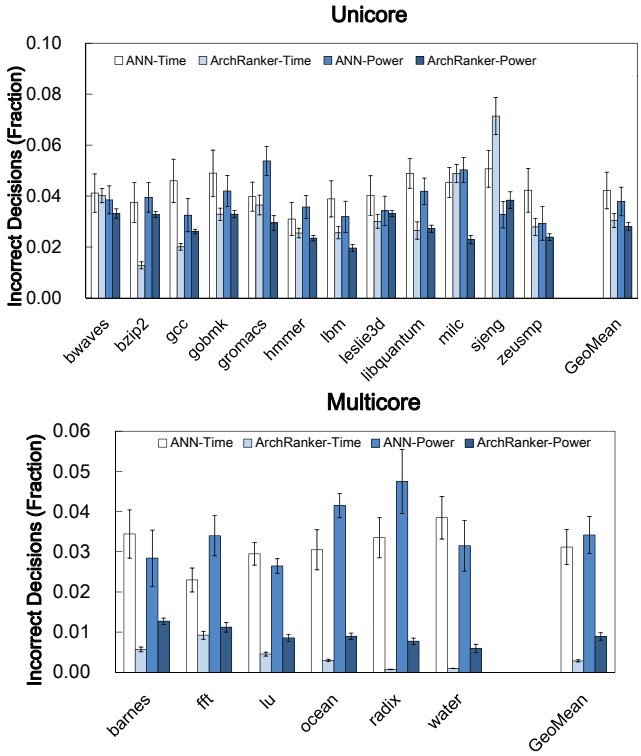

Figure 9: Fraction of incorrect DSE decisions made by ArchRanker and ANN.

In this section, we try to emulate even more closely the DSE process, which is inherently iterative, and broken down into a long sequence of trial and error iterations. After N such iterations, the architect will typically seek an $(N+1)$-th configuration which is hopefully better than the best obtained after the N previous iterations.

We emulate this scenario using $N+1$ configurations as follows: we pick a subset of N configurations (as if they were the N previous steps of the DSE process), and we try to predict whether the $(N+1)$-th configuration (the new candidate step) is better or not than the best configuration among the N other configurations. We emulate the scenario for $N = 100$; we randomly select the 100 configurations among our pool of 500 configurations, and we train ArchRanker using these 100 configurations. We consider in turn each of the remaining 400 configurations as the 101-th configuration, and ArchRanker tries to predict whether it is better or not than the best configuration among the 100 configurations simulated so far. We repeat this process 10 times, by randomly changing the set of 100 configurations and retraining every time (in total, $10 \times 400 = 4000$ predictions per benchmark).

If ArchRanker predicts that this 101-th configuration is better, the architect will simulate it, i.e., the prediction will be used to drive the DSE decision. We measure how many times this decision is correct, i.e., how many times ArchRanker cor-

rectly predicts the relative merit of this 101-th configuration with respect to the best seen so far among the 100 previous configurations. We do the same with the ANN, and we report the results in Figure 9 as the fraction of correct decisions. For 10 out of 12 benchmarks of the unicore time scenario, 11 out of 12 benchmarks of the unicore power scenario, and all the benchmarks of the multicore time and power scenarios, ArchRanker takes better decisions than ANN. On average, ArchRanker makes 27.90%, 25.91%, 90.82%, and 73.80% fewer incorrect decisions than ANN on benchmarks of the unicore time, unicore power, multicore time, multicore power scenarios, respectively. For instance, on *ocean*-M-Time, ArchRanker makes 90.13% fewer incorrect decisions than ANN.

4.5. Reliability Estimate of Pairwise Prediction

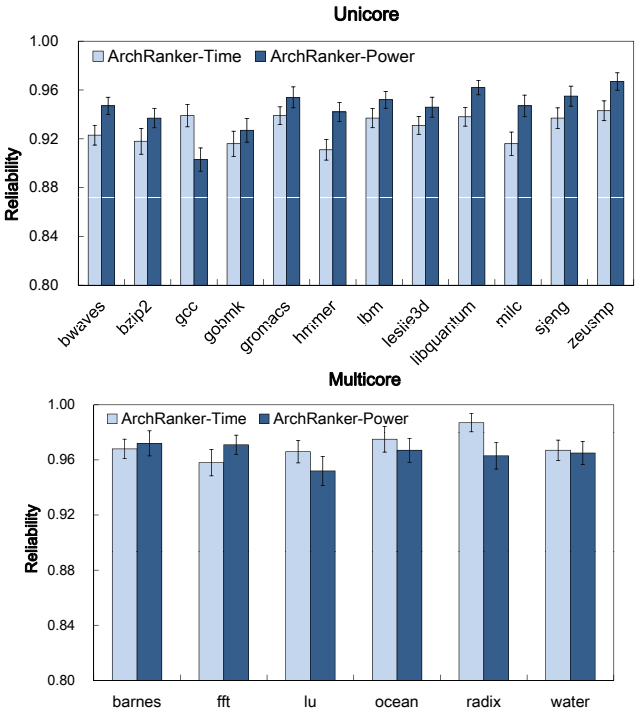

Figure 10: Reliability of pairwise prediction made by ArchRanker.

A useful complementary information of a pairwise prediction is the *reliability* of that prediction. Such a reliability can be evaluated with a sufficiently large number of samples. By *repeatedly* randomly selecting the 100 configurations among 500 used for training, we have a large sample population available to compute such estimates. So we create k ranking models $H_1, \ldots, H_k$ with different training sets (which is a common practice in machine learning), and, assuming $\mathbf{x}$ has a better response than $\mathbf{x}'$ without loss of generality, i.e., $R(\mathbf{x}) > R(\mathbf{x}')$, we estimate the reliability as $\frac{1}{k}\sum_{i=1}^{k}\chi\left(H_i(\mathbf{x}) - H_i(\mathbf{x}')\right)$, where $\chi(a)$ equals 1 for any $a > 0$, and 0 otherwise.

We apply this approach to the unconstrained unicore/multicore time/power scenarios for $k = 20$, and we show the results in Figure 10 in the form of an error bar. We find that the reliability is 92.87%, 94.48%, 97.01% and 96.49% on average for unicore time, unicore power, multicore time and multicore power scenarios, respectively, and varies from 90.36% for *gcc*-U-Power to 98.71% for *radix*-M-Time.

5. Related Work

5.1. Regression-based Modeling of Design Space

Over the past few years, a number of excellent regression-based techniques have been applied to DSE and performance analysis. Joseph *et al.* employed linear regression to construct linear predictive models of the performance of architecture configurations [21]. These linear models, however, are not capable of characterizing non-linear response behavior of sophisticated design spaces. As a result, Joseph *et al.* later employed Radial Basis Function (RBF) networks (a type of neural network) to construct non-linear regression models of processor performance [22]. In parallel with the above work, İpek *et al.* proposed to use ANNs [18, 19], and Lee and Brooks proposed to use spline functions [25], to model processor responses under superscalar or CMP design scenarios. Compared with ANN, the results of spline function are easier to interpret but requires human interventions; both methods were found to have similar accuracy [27]. In another piece of work, Lee and Brooks further demonstrated the effectiveness of spline regression models through Pareto front analysis, pipeline depth analysis, as well as multiprocessor heterogeneity analysis [26]. Guo *et al.* designed a co-training-like algorithm to further enhance the accuracy of regression-based design space modeling [15], which was inspired by the paradigm of disagreement-based semi-supervised learning [44].

Lee and Brooks proposed the Composable Performance Regression (CPR) technique to predict the performance of multiple workloads executed on multiprocessor systems [28], where the CPR technique is based on the combination of spline regression models characterizing uniprocessor performance and contention, respectively. Dubach *et al.* [6] proposed to construct a cross-application regression model for superscalar architectures executing different applications. Their cross-application model linearly combines a number of application-specific regression models (each of them is an ANN), which predicts processor responses of configurations with respect to an unseen application at moderate simulation cost for extracting the signature of that application. Khan *et al.* [24] independently proposed a similar approach to construct a cross-program regression model for a multicore design space. Dubach *et al.* employed Support Vector Machine (SVM) to model a joint architecture-compiler design space, and the trained SVM can predict the compiler performance across different architecture configurations [7]. Azizi *et al.* proposed to characterize and predict the energy-performance trade-off

of a joint circuit-architecture design space with posynomial functions [1]. The above regression techniques for DSE had been shown to be accurate on predicting processor responses, yet it is still not clear how they predict relative rankings of configurations given a limited amount of architectural simulations.

Our work is substantially different from the above investigations. We no longer stick to the hard regression formulation of DSE, but, to the best of our knowledge, we propose for the first time to formulate DSE as a ranking problem. This new formulation can help drastically reduce the number of required simulations.

5.2. Analytical Modeling

Analytical modeling captures architect knowledge and allows to estimate architecture behavior using few or no simulation. Noonburg and Shen estimate the performance of superscalar processors by probabilistically characterizing program parallelism and machine parallelism [32]. Karkhanis and Smith proposed a first-order performance model for superscalar processors, which penalizes the ideal performance with miss events (e.g., branch mispredictions, instruction cache misses, and data cache misses) [23]. Eyerman *et al.* further extended the above model by dividing the instruction execution flow into intervals separated by different miss events [11]. Chandra *et al.* proposed a probabilistic model to predict the extra cache misses caused by cache contention between two different threads on a CMP [3]. Eklov *et al.* proposed a statistical cache contention model called StatCC to predict the performance of a set of co-executed threads [10]. Chen and Aamodt utilized Markov chain to accurately model throughput of multicore architectures running multi-threaded programs [4], and their model was later extended to estimate the throughput of multi-programmed many-core processors [5]. Sun *et al.* proposed an analytical performance model called Moguls to help architects quickly explore the design space of memory hierarchies [40]. Nair *et al.* proposed a first-order mechanistic analytical model for computing the architectural vulnerability factor by estimating the occupancy of correct-path state via inexpensive profiling [30].

Analytical modeling is not simulation-intensive, but it can be difficult to grasp and integrate a large number of interacting parameters within such models. In contrast, our technique is applicable to large and complex design spaces. However, we view analytical models as potentially complementary to ranking models, due to the ability to integrate expert knowledge in ranking models.

5.3. Fast Simulation Techniques

In addition to regression-based/ranking-based techniques which reduce the total number of architectural simulations for DSE, there are many fast simulation techniques which can cut down the cost of each simulation.

Iyengar *et al.* proposed a metric of representativeness for refined traces and developed a novel graph-based heuristic to generate better refined traces [20]. Nussbaum and Smith proposed to extract intrinsic program characteristics from a program instruction trace, with which a compact synthetic instruction trace can be generated for simulation [33]. Eeckhout *et al.* proposed an improved statistical simulation framework employing a statistical flow graph to accurately characterize the control flow behavior of a program [9]. To reduce simulation overhead for CMP design, Genbrugge and Eeckhout proposed several statistical quantities to capture the behavior of cache access and shared resource contentions that are critical to the performance of multi-programmed programs running on CMPs [13, 14]. Hughes and Li proposed to leverage statistical characteristics that capture the behaviors of memory sharing and inter-thread synchronization when constructing synthetic multi-threaded programs [16].

There are also techniques which directly extract several short but representative instruction phases from the original program. SimPoint [38] clusters execution phases of programs that are characterized by basic block vectors, and then takes the cluster centroids as the representative of simulation phases. SMARTS [43] selects representative instruction segments from the original program based on statistical sampling theory, which can identify the number of samples (i.e., instruction segments) sufficient to achieve a user-specified confidence interval of the performance.

Fast simulation techniques, which reduce the cost-per-simulation, are orthogonal and again complementary to our ranking-based DSE technique, by reducing the time required to build the training set.

6. Conclusions and Future Work

In this paper, we argue that the information the architect mostly needs during the DSE process is whether a given configuration will perform better than another one, or better than any other one seen so far, rather than precisely estimating the performance of that configuration. Therefore, we propose to formulate the DSE as a ranking problem where we train a model to predict which of two architecture configurations will perform best. We also present the ArchRanker framework for ranking-based DSE, and compare it against ANN-based regression.

We create three main types of DSE scenarios: constraint-based DSE emulating the frequent case where the architect has a given performance target (given speedup or power), unconstrained DSE where we consider all possible pairwise comparisons among the simulated configurations, and an iterative decision process which attempts to mimic even more closely the DSE process. For each scenario (unicore/multicore, time/power), we find that ArchRanker outperforms ANN on average (over all scenario benchmarks). For instance, for constraint-based DSE, the best configuration response predicted by ArchRanker is 31.77% better than the one predicted

by ANN in the M-EDDP/ Time+Power scenario; for unconstrained DSE, ArchRanker makes 38.13% fewer incorrect pairwise predictions than ANN on average, in the unicore time scenario; and for the iterative DSE process, 90.82% fewer incorrect decisions than ANN in the multicore time scenario. Moreover, we show that an ANN regression model requires from 154_{+}% to 300_{+}% more training simulations than ArchRanker to achieve the same pairwise prediction accuracy. Finally, we show that ArchRanker can be used to evaluate the response gap between two configurations, and still outperforms ANN on a task which is normally the purview of value prediction models. In summary, ArchRanker realizes the best of both worlds: better prediction accuracy on the key information expected by the architect during the DSE process, and smaller training time.

Future work will focus on incorporating expert knowledge in ArchRanker for further improving prediction accuracy.

References

[1] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis", in *ISCA-37*, 2010.

[2] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations", in *ISCA-27*, 2000.

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture", in *HPCA-11*, 2005.

[4] X. E. Chen and T. M. Aamodt, "A first-order fine-grained multithreaded throughput model", in *HPCA-15*, 2009.

[5] X. E. Chen and T. M. Aamodt, "Modeling Cache Contention and Throughput of Multiprogrammed Manycore Processors", *IEEE Transactions on Computers* 61(7), 2012.

[6] C. Dubach, T. M. Jones, and M. F. P. O'Boyle, "Microarchitectural Design Space Exploration Using an Architecture-Centric Approach", in *MICRO-40*, 2007.

[7] C. Dubach, T. M. Jones, and M. F. P. O'Boyle, "Exploring and predicting the architecture/optimising compiler co-design space", in *CASES'08*, 2008.

[8] C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. P. O'Boyle, "A predictive model for dynamic microarchitectural adaptivity control", in *MICRO-43*, 2010.

[9] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. D. Bosschere, and L. K. John, "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies", in *ISCA-31*, 2004.

[10] D. Eklov, D. Black-Schaffer, and E. Hagersten, "StatCC: a statistical cache contention model", in *PACT-19*, 2010.

[11] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors", *ACM Transactions on Computer Systems* 27 (2), 2009.

[12] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An Efficient Boosting Algorithm for Combining Preferences", *Journal of Machine Learning Research* 4, 2003.

[13] D. Genbrugge and L. Eeckhout, "Statistical simulation of chip multiprocessors running multi-program workloads", in *ICCD-25*, 2007.

[14] D. Genbrugge and L. Eeckhout, "Chip Multiprocessor Design Space Exploration through Statistical Simulation", *IEEE Transactions on Computers* 58(12), 2009.

[15] Q. Guo, T. Chen, Y. Chen, Z.-H. Zhou, W. Hu, and Z. Xu, "Effective and Efficient Microprocessor Design Space Exploration Using Unlabeled Design Configurations," in *IJCAI'11*, 2011.

[16] C. Hughes and T. Li, "Accelerating Multi-core Processor Design Space Evaluation Using Automatic Multi-threaded Workload Synthesis", in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[17] W. Iba and P. Langley, "Induction of One-Level Decision Trees", in *Proceedings of the Ninth International Workshop on Machine Learning*, 1992.

[18] E. Ïpek, S. A. McKee, B. de Supinski, M. Schulz, and R. Caruana, "Efficiently exploring architectural design spaces via predictive modeling", in *ASPLOS-12*, 2006.

[19] E. Ïpek, S. A. McKee, K. Singh, R. Caruana, B. de Supinski, and M. Schulz, "Efficient architectural design space exploration via predictive modeling", *ACM Transactions on Architecture and Code Optimization* 4(4), 2008.

[20] V. S. Iyengar, L. H. Trevillyan, and P. Bose, "Representative Traces for Processor Models with Infinite Cache", in *HPCA-2*, 1996.

[21] P. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis", in *HPCA-12*, 2006.

[22] P. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "A Predictive Performance Model for Superscalar Processors", in *MICRO-39*, 2006

[23] T. S. Karkhanis and J. E. Smith, "A First-Order Superscalar Processor Model", in *ISCA-31*, 2004.

[24] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra, "Using Predictive Modeling for Cross-Program Design Space Exploration in Multicore Systems", in *PACT-16*, 2007.

[25] B. C. Lee and D. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction", in *ASPLOS-12*, 2006.

[26] B. C. Lee and D. Brooks, "Illustrative design space studies with microarchitectural regression models", in *HPCA-13*, 2007.

[27] B. C. Lee and D. Brooks, "Applied inference: Case studies in microarchitectural design", *ACM Transactions on Architectecture and Code Optimization* 7(2), 2010.

[28] B. C. Lee, J. Collins, H. Wang, and D. Brooks, "CPR: Composable performance regression for scalable multiprocessor models", in *MICRO-41*, 2008.

[29] T. Y. Liu, "Learning to rank for information retrieval", *Foundations and Trends in Information Retrieval* 3(3), 2009.

[30] A. A. Nair, S. Eyerman, L. Eeckhout, and L. K. John, "A first-order mechanistic model for architectural vulnerability factor", in *ISCA-39*, 2012.

[31] A. A. Nair and L. K. John, "Simulation points for SPEC CPU 2006", in *ICCD-26*, 2008.

[32] D. B. Noonburg and J. P. Shen, "Theoretical modeling of superscalar processor performance", in *MICRO-27*, 1994.

[33] S. Nussbaum and J. E. Smith, "Modeling superscalar processors via statistical simulation", in *PACT-10*, 2001.

[34] G. Palermo, C. Silvano, and V. Zaccaria, "ReSPIR: A Response Surface-Based Pareto Iterative Refinement for Application-Specific Design Space Exploration", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28(12), 2009.

[35] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for Accurate and Efficient Simulation", in *SIGMETRICS'03*, 2003.

[36] R. E. Schapire and Y. Freund, *Boosting: Foundations and Algorithms*, MIT Press, 2012.

[37] D. Sculley, "Combined regression and ranking", in *KDD-16*, 2010.

[38] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, "Automatically Characterizing Large Scale Program Behavior", in *ASPLOS-10*, 2002.

[39] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An Integrated Cache Timing, Power and Area Model", Technical Report, Compaq Computer Corporation, 2001.

[40] G. Sun, C. J. Hughes, C. Kim, J. Zhao, C. Xu, Y. Xie, and Y.-K. Chen, "Moguls: a model to explore the memory hierarchy for bandwidth improvements", in *ISCA-38*, 2011.

[41] F. Wilcoxon, "Individual comparisons by ranking methods", *Biometrics* 1(6), 1945.

[42] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations", in *ISCA-22*, 1995.

[43] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling", in *ISCA-30*, 2003.

[44] Z.-H. Zhou and M. Li, "Semi-supervised learning by disagreement", *Knowledge and Information Systems* 24(3), 2010.

[45] Z.-H. Zhou, *Ensemble Methods: Foundations and Algorithms*, Boca Raton, FL: CRC Press, 2012.