Automatic Performance Model Construction for the Fast Software Exploration of New Hardware Designs

John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael F.P. O'Boyle
Member of HiPEAC
School of Informatics
University of Edinburgh, UK

Grigori Fursin and Olivier Temam
Member of HiPEAC
ALCHEMY Group
INRIA Futurs and LRI, Paris-Sud University, France

ABSTRACT

Developing an optimizing compiler for a newly proposed architecture is extremely difficult when there is only a simulator of the machine available. Designing such a compiler requires running many experiments in order to understand how different optimizations interact. Given that simulators are orders of magnitude slower than real processors, such experiments are highly restricted. This paper develops a technique to automatically build a performance model for predicting the impact of program transformations on any architecture, based on a limited number of automatically selected runs. As a result, the time for evaluating the impact of any compiler optimization in early design stages can be drastically reduced such that *all* selected potential compiler optimizations can be evaluated. This is achieved by first evaluating a small set of sample compiler optimizations on a prior set of benchmarks in order to train a model, followed by a very small number of evaluations, or probes, of the target program.

We show that by training on less than 0.7% of all possible transformations (640 samples collected from 10 benchmarks out of 880000 possible samples, 88000 per training benchmark) and probing the new program on only 4 transformations, we can predict the performance of all program transformations with an error of just 7.3% on average. As each prediction takes almost no time to generate, this scheme provides an accurate method of evaluating compiler performance, which is several orders of magnitude faster than current approaches.

Categories and Subject Descriptors

D.3 [**Software**]: Programming languages; D.3.4 [**Programming languages**]: Processors—*Compilers, Optimization*; I.2.6 [**Artificial intelligence**]: Learning—*Induction*

General Terms

Performance, Experimentation, Languages

Keywords

Compiler optimization, Architecture, Performance Modelling, Machine learning, Artificial Neural Networks

CASES'06, October 23–25, 2006, Seoul, Korea.

1. INTRODUCTION

Computer architectures have significantly increased in complexity in the last 20 years. As a result, simulators have also become increasingly complex and, critically, three or more orders of magnitude slower than real processors. However, early on in the design cycle of a new processor, software and compiler engineers need to port system applications, develop compilers, tune libraries and run large applications for prospective customers on the new target architecture before the processor is available. They therefore have to rely on simulators for their task. As the design cycle is long, software and compiler engineers are left with using exceedingly slow simulators for tuning purposes. Several years can elapse between the time a first simulator is available, and the time the actual architecture is available. This problem is further aggravated by stringent time-to-market constraints, and the increasing complexity of compilers [2]. What is needed is a fast *proxy* of the hardware. Given a new program, we would like an accurate prediction of the performance of the program without having to run it, or perhaps only running it a few times. This would dramatically overcome the performance shortcomings of existing simulators and allow many different versions of a program to be evaluated for tuning purposes.

In this article, we propose a method for drastically reducing the overall time required to tune applications for new architectures early on in the design cycle, when only slow simulators are available. We can build a performance model of a new architecture that is accurate enough for program tuning purposes and dramatically faster than simulators. Though model construction requires a number of training runs on the available simulator, it is entirely *automatic*. This model is built by monitoring how programs react to certain program transformations on the target architecture. The model is first trained using a few programs to which program transformations are randomly applied; then it requires a few test runs from a selection of characteristic transformations for the program to be tuned. These latter characteristic transformations are selected automatically based on a technique called *mutual information maximization*, which determines the transformations likely to give the most information about a program's performance on a particular machine. From then on, the model can instantly estimate the speedup of the target program if any known program transformation were applied; in fact, the model enables the evaluation of *all* program transformations in any order and in a very small time. We empirically show that with 640 training runs (corresponding to only 0.7% of the total number of possible transformations sequences), and 4 test runs to characterize a new target program, our model can predict the program speedup after applying a transformation with an error of 7.3%.

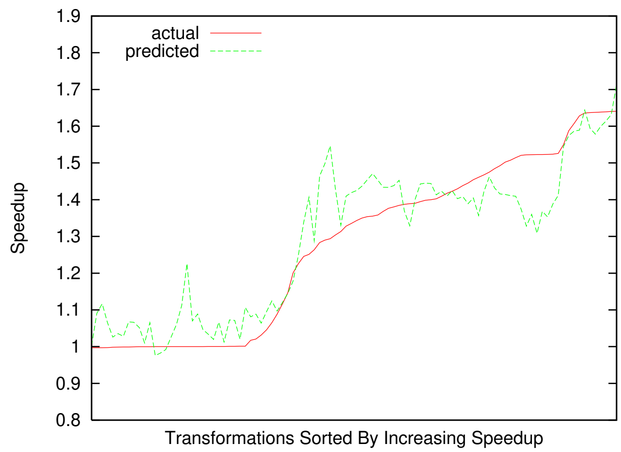

Figure 1: ***Model accuracy for*** **`compress`*. The y-axis corresponds to speedup relative to the original untransformed program. The x-axis corresponds to transformation sequences sorted in increasing performance. The line marked*** `actual` ***corresponds to the real measured performance and*** `predicted` ***corresponds to the model's predictions. The predictions are averaged over 30 trials.***

Figure 2: ***Actual speedups for all transformations applied to 3 benchmarks, sorted according to the speedup achieved for*** **`compress`** ***(one trial). The comparison highlights the differences in both the absolute and relative program behavior to transformations.***

Beyond providing an automatic process for building a reliable and fast performance model, our approach has several assets. The model can not only be used to predict the impact of a program transformation, but can also predict the best possible speedup after applying any known transformation. Beyond the initial training, the model accuracy can be regularly improved through continuous on-line training. Any simulation performed on any program after the initial training, including the test runs to characterize new programs, can augment the total training set, all without any human intervention. Moreover, since the model provides speedup estimates in almost no time, it has applications beyond the design cycle since it is much faster than running applications on the real machine. It can be used for benchmarking purposes, i.e., tuning applications of prospective customers, and can even be provided to end-users. Finally, the approach can also be used by architecture designers to take into account the impact of software tuning when designing architectures. After a number of training runs on a given simulator configuration during design-space exploration, our model would behave as if a compiler had been tuned for this architecture configuration. Design decisions would then not only be based upon estimated architecture performance but combined estimated architecture+compiler performance.

The paper is organized as follows: Section 2 provides an example showing how our automatically generated model can predict the speedups over a large transformation space. Section 3 describes our reaction-based predictive modeling technique and how we automatically find the best transformations to characterize a program. Section 4 describes the experimental methodology, while Section 5 provides empirical evidence of the accuracy of our technique, followed by related work in Section 6 and conclusions in Section 7.

2. EXAMPLE

Let us assume that we are at the beginning of the design cycle of a processor, that only a performance simulator is available, and that software and compiler engineers start tuning applications for this processor. In this section, we provide a simple example showing that it is possible to automatically train a model, using a number of runs on a few benchmarks, in order to predict the performance impact of compiler optimizations on any new program. As a result, we can drastically reduce the overall simulation time necessary to evaluate tentative architectures and tune programs to new architectures; using the model has virtually no cost.

We wish to predict the performance of a new program on a particular platform, in this case the Texas Instrument C6713 clustered VLIW processor. In order to evaluate our predictor, we generate many different versions of the program using 13 different program transformations, listed in Table 1. We wish to predict what effect any combination of these will have on any new program. Since program transformations can be composed into arbitrarily long sequences, the number of possible program versions is large. In practice, combinations of 5 transformations are a reasonable maximum. As a result, the total transformation space size we consider is 88000. Thus, we wish to build a model that can predict the performance of 88000 different versions of a new program.

In order to collect the data for our model, we performed 640 runs of randomly chosen transformations on training benchmarks (10 training benchmarks, 64 samples per benchmark). Then 4 carefully selected runs are performed on a *new* program. These runs are used to "probe" the new program, and characterize its reactions to program transformations. We will show that only a few such probes are necessary to characterize the new program behavior on a large range of program transformations. We call such an approach *reaction*-based modeling. Given this total of 644 evaluations, we are then able to build a model that accurately predicts the entire 88000 different versions of the program.

In order to validate the accuracy of our model, we have *exhaustively* searched the transformation space for each of the benchmarks on the TI C6713 processor, with a total running time of 33 days. Figure 1 shows how the model accuracy is exercised on program `compress`. The y-axis is the speedup obtained after applying a transformation, and the x-axis is simply the transformations sequences sorted by increasing (`actual`) speedup. The dotted line shows the speedup estimated by the model, and the solid line is the actual speedup. The average error is 10.3% for this benchmark.

At first, it may be surprising that such a small training set size is sufficient to capture such a huge space. However, Figure 2 ex-

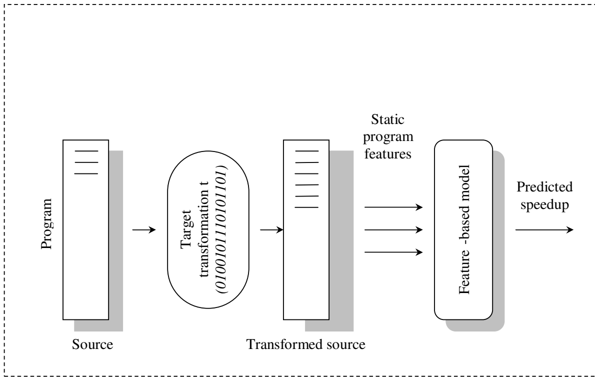

Figure 3: ***Feature-based model.*** **Input:** ***static features extracted from the transformed program at the source level;*** **Output:** ***program speedup.***

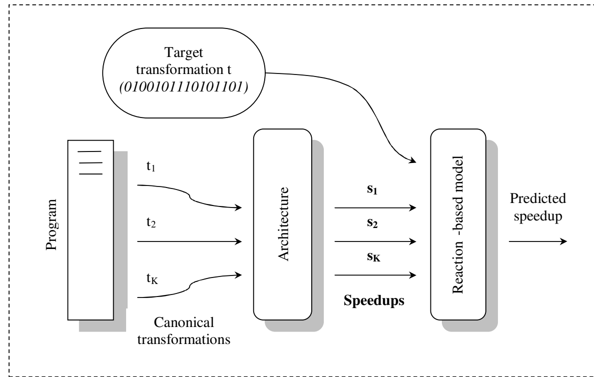

Figure 4: ***Reaction-based model.*** **Input:** ***target transformation and speedups on canonical transformation sequences;*** **Output:** ***transformation sequence speedup.***

plains why this is possible: programs exhibit a "plateau"-like profile, which means there are few different performance levels for each benchmark, or, in other words, many transformation sequences have similar impact. Almagor *et al.* [2] have built similar speedup graphs which again show few plateaus, but for different transformation spaces. As a result of this small number of plateaus, if (1) we can automatically identify a few characteristic transformations which capture these plateaus, and (2) we can cluster transformations according to the performance plateau to which they belong, we can build a performance model with a small number of training runs. This is what we achieve with reaction-based modeling.

Still, it does not mean the behavior of all benchmarks to all transformations is the same, and that the transformation space is easily predicted. Figure 2 shows the same speedup graph as in Figure 1, using the transformation order given by benchmark `compress` to plot benchmarks `adpcm` and `fir`. If all benchmarks would react similarly to most transformations, all the benchmark curves would be monotonic and increasing. Obviously it is not the case, hinting at the complexity of the transformation space.

Figure 1 shows that we can fairly accurately predict the performance of different versions of an unseen program. Still, it may be argued that, in practice, a tuning process may involve manual transformations not considered here. However, manual transformations can usually be decomposed into sequences of simple systematic compiler transformations, as recently illustrated by Parello *et al.* [22]. Furthermore, our code characterization process based on reactions can indifferently target whole programs or specific code sections, because it solely relies on performance measurements/reactions and is not dependent on code structure.

This section has provided a motivating example, illustrating that it is possible to build a model that predicts the performance of different versions of a program without having to execute the program. The next section describes how we can automatically build such a model using machine learning.

3. PREDICTIVE PERFORMANCE MODELING

3.1 Characterization

At the heart of our approach is building a performance prediction model. More precisely, the model must accurately and rapidly predict the performance impact (speedup) of a large range of program transformations on a given program, so that a large software design-space can be explored without having to simulate or run the multiple transformed versions of the program.

More formally, let P be a program, t a program transformation sequence, and s the speedup of P after applying t, we wish to build a predictive performance model f, that predicts a speedup $\hat{s}$ close to the real speedup s, i.e., $f(P,t) = \hat{s}$. In standard predictive modeling techniques, such a model is built using a training set, here, a set of tuples (P_i, t_i, s_i).

3.1.1 Static program feature-based modeling

Most machine-learning approaches applied to compiler optimization have used *program feature*-based characterization to automatically build such a model (see Section 6). This approach implicitly assumes one can identify a set of static program features which characterize program behavior. Feature-based modeling takes a summary P^* of the input static program P. In order to capture the effect of program transformations, we collect the static features at the source code level, after applying program transformations using SUIF, see Figure 3.

After training, the model function f takes as inputs the code features of the transformed program P^* , and outputs the predicted speedup, e.g.: $f(P^*) = \hat{s}$.

Such an approach is attractive because the transformed program needs not be executed in order to predict its performance, and it has been successfully used in the past for specific optimizations, e.g., targeting loop nests. Moreover, this approach can be used on any new transformed program since the model just uses code features as input. In particular, the model is not restricted to the set of transformations used during training; a new transformation can be used to generate the transformed program. However, it is harder to extract appropriate features to characterize a whole program, and we will also empirically show in Section 5 that feature-based characterization does not perform as well when a large range of program transformations is considered.

As a result, we need to come up with a different characterization method, compatible with a large range of program transformations and whole-code characterization.

3.1.2 Reaction-based modeling

Building an analytical model of the performance behavior of a complex program on a modern processor architecture is known to be a difficult task [4]. Rather than building a processor performance model, we have developed a modeling approach capable of capturing the performance effect of program transformations which we call *characterization by reactions*. Characterization by reactions is an empirical analysis method used in many scientific domains. A new target may be "probed" in various ways, and its reactions are observed. If previously studied targets had similar reactions, the new target may be classified to be similar to one of the previous targets or a combination of some of these targets.

The principles of characterizing programs by reactions are exactly the same. In our case, the "probes" are automatically selected program transformations which are applied to the program, and the behavior observed is the resulting speedup. Using this approach, we build a model f which takes as inputs the reactions $s_i(t_i; P)$ (i.e., the speedups after applying transformation t_i) of the program P to carefully chosen transformation sequences $t_1, \ldots, t_K$, and a new transformation sequence t. We use the trained model in order to predict speedups on new programs and transformation sequences: $f(s_1(t_1; P), \ldots, s_K(t_K; P), t) = \hat{s}$ – see Figure 4.

Because this characterization method is based on program runtime behavior rather than a static program characterization (as for code features), it is well suited to whole-code characterization, as opposed to small code constructs, e.g., loop nests. On the other hand, it has the disadvantage of requiring K program runs for building a predictor. However, we empirically show in Section 5 that we can keep K very small (less than or equal to 4), and still get accurate predictions.

3.2 Building the model

There exist many modeling techniques that can be used to automatically produce a predictive model. In this paper we use a feed-forward ANN (Artificial Neural Network), with one hidden layer and five hidden units, as it is robust to noise in its input and capable of learning real-valued outputs - both characteristics of our problem domain. ANNs are well studied and have been used in a wide range of domains [5]. We have also considered other techniques, such as *Mixtures of Experts* [16] and *Regression Trees* [6]; however, our current experience suggests that ANNs may be particularly practical for the prediction problem.

3.3 ANN

The model is constructed as follows: the inputs are the reactions and the target transformation whose performance impact we want to predict, and the output is the predicted speedup of the target transformation. More precisely, for any given program P, the inputs of the ANN are the speedups $s_1(t_1; P), \ldots, s_K(t_K; P)$ obtained for the K probe transformations, and the identifier t of the target program transformation. The identifier is provided as a sequence of 13 bits, one per possible elementary transformation; since each transformation sequence is composed of elementary transformations, at most 5 such bits are set to 1. The model is not trained specific to a program, it is trained on multiple programs, and can be applied to any unseen program thereafter.

The K probe (or *canonical*) transformations are chosen to be most characteristic of the program behavior, and the resulting performance speedups may be used to *discriminate* between the training programs. These canonical transformations are not defined a priori or in ad-hoc manner; they are learned from data by using a systematic algorithm which we will now describe.

3.4 Selecting the canonical transformations

We wish to determine the smallest set of canonical transformations, or probes, with which to characterize a new program. By observing the canonical speedups for a new program, we want to be able to say how similar this new program is to each of the training programs, which may potentially be a useful characterization for improving predictions. One way to learn such canonical transformations is by using information theory [9], which allows one to select transformations informative about the underlying benchmark.

Let $s_1, \ldots, s_T$ define *random variables* describing the speedups for the T transformations sequences $t_1, \ldots, t_T$ from the training set. Our goal here is to extract the canonical subset of K transformations $\{t_{i_1}, \ldots, t_{i_K}\}$, such that the resulting improvements $\{s_{i_1}, \ldots, s_{i_K}\}$ are most informative about the program P. A formal measure of information is *mutual information* [9]:

$$\begin{aligned} I(P; s_{i_1}, \ldots, s_{i_K}) &\stackrel{\text{def}}{=} H(s_{i_1}, \ldots, s_{i_K}) - H(s_{i_1}, \ldots, s_{i_K}|P) \\ &\stackrel{\text{def}}{=} H(\mathbf{\Delta}) - H(\mathbf{\Delta}|P), \end{aligned} \tag{1}$$

where $\mathbf{\Delta} \stackrel{\text{def}}{=} [s_{i_1}, \ldots, s_{i_K}]$ is the vector of speedups for the canonical transformations, and $H(\mathbf{\Delta})$, $H(\mathbf{\Delta}|P)$ are the marginal and the conditional entropies respectively:

$$H(\mathbf{\Delta}) \stackrel{\text{def}}{=} -\sum_{\mathbf{\Delta}} p(\mathbf{\Delta}) \log p(\mathbf{\Delta}), \tag{2}$$

$$H(\mathbf{\Delta}|P) \stackrel{\text{def}}{=} -\frac{1}{M} \sum_{P=1}^{M} \sum_{\mathbf{\Delta}} p(\mathbf{\Delta}|P) \log p(\mathbf{\Delta}|P) \tag{3}$$

(see e.g. [9]). Our goal is to maximize (1) with respect to the indices of the canonical transformations $i_1, \ldots, i_K$.

It is clear that for a given benchmark P, the speedup obtained on transformation t_i may be found from the training data deterministically. Indeed, it is given by the table look-up, i.e. $\forall i.\ p(s_i|P) \sim \delta(s_i - s(t_i; P))$. Analogously we can compute the marginal distributions $p(s_i)$ by counting the speed-ups across the programs. Additionally, we note that for all transformations, the corresponding speedups are *conditionally independent* given P, i.e.

$$p(\mathbf{\Delta}|P) = \prod_{j=1}^{K} p(\Delta_{i_j}|P) \sim \delta. \tag{4}$$

As there is no uncertainty in determining speedup s_i for transformation t_i and a given program P, the conditional entropy $H(\mathbf{\Delta}|P)$ may be dropped from the objective function (1); in other words, optimization of the mutual information (1) reduces in our case to optimization of the marginal entropy $H(\mathbf{\Delta})$.

In general, computation of $H(\mathbf{\Delta})$ is a difficult task. For example, if the speed-ups are quantized to lie in a discrete space of S bins, the computational complexity of evaluating $H(\mathbf{\Delta})$ is $\sim O(S^K)$, i.e. approximations need to be considered. We use the *as-if Gaussian* approximation $H(\mathbf{\Delta}) \approx (1/2) \log |\text{cov}(\mathbf{\Delta})| + const$, which effectively reduces the optimization problem to maximizing the volume of the covariance of the selected canonical speedups. We also impose an additional constraint such that each canonical transformation should be individually predictive about the program, i.e. $I(s_{i_k}, P) > \epsilon$ for some $\epsilon > 0$ (here $I(s_{i_k}, P)$ is defined similarly to (1)).

Our method has a useful interpretation as an approximate *redundancy reduction*. In our case, equation (1) can be transformed into the following form

$$I(P; \mathbf{\Delta}) = \sum_{j=1}^{K} I(s_{i_j}, P) - \sum_{\mathbf{\Delta}} p(\mathbf{\Delta}) \log \frac{p(\mathbf{\Delta})}{\prod_{j=1}^{K} p(s_{i_j})}. \tag{5}$$

The rightmost term in (5) defines the *redundancy*, which is zero when the speedups are independent, and it is large when they are strongly correlated. In order to select the most informative subset of transformations (in the mutual information sense), we need to optimize (5) with respect to the indices of the canonical transformations $i_1, \ldots, i_K$. The computation of the first term $\sum_{j=1}^{K} I(s_{i_j}, P) = \sum_{j=1}^{K} H(s_{i_j})$, is $\sim O(SK)$, where S is the number of quantization bins. However, computation of the redundancy is difficult and

Label	Transformation
1,2,3,4	Loop unrolling
n	FOR loop normalization
t	Non-perfectly nested loop conversion
k	Break load constant instructions
s	Common subexpression elimination
d	Dead code elimination
h	Hoisting of loop invariants
i	IF hoisting
m	Move loop-invariant conditionals
c	Copy propagation

Table 1: ***The labeled transformations used for the exhaustive enumeration of the space. 1,2,3,4 corresponds to the loop unroll factor.***

Label	Static Feature
LDC	Load a constant value
CVT	Conversion between float/int
LOD	Load from memory
STR	Store to memory
MBR	Multi-way branch
CMPI/CMPF	Comparison using int/float
UJMP/CJMP	Unconditional/Conditional jump
CPY	Copy
SFT	Shift
ROT	Rotation
ARII/ARIF	Arithmetic operation on int/float
MULI/MULF	Multiplication on int/float
DIVI/DIVF	Division on int/float
LOG	Logical operation
CAL	Function call
ARYI/ARYF	Array operation with int/float (address computation)

Table 2: ***Static program features.***

needs to be approximated. To maximize (5), we apply a simple greedy approximation strategy by choosing transformations which lead to a high information content *individually*, and which are approximately maximally independent from one another.

Specifically, by using the Gaussian approximation, we are greedily maximizing the log determinant of the sample covariance of the canonical speedups $\log|\text{cov}(\mathbf{\Delta})|$ for transformations which individually have large marginal entropies $H(s_{i_j})$. In practice, we recalculate the best canonical transformations as we progressively collect more training data, and we experimented with the number of canonicals in the range of 1 to 8. Once we have selected the canonical transformations containing the most information, we apply them to the program to be predicted. Their execution times are the final inputs into our trained proxy model.

4. EXPERIMENTAL METHODOLOGY

This section provides a brief description of the programs, transformations and platforms used in our experiments.

4.1 Benchmarks

The *UTDSP* [19, 24] benchmark suite was designed "to evaluate the quality of code generated by a high-level language (such as C) compiler targeting a programmable digital signal processor (DSP)" [19]. This set of benchmarks contains small, but compute-intensive DSP kernels as well as larger applications composed of more complex algorithms. The size of programs ranges from 20 to 500 lines of code. These programs represent compute-intensive kernels widely regarded most important by DSP programmers and are used indefinitely in stream-processing applications.

4.2 Transformations

In this study, we consider source-to-source transformations (many of these transformations also appear within the optimization phases of a native compiler [2]), applicable to C programs and available within the restructuring compiler SUIF 1 [14]. For the purpose of this paper, we have selected the transformations described and labeled in Table 1. As we (arbitrarily) consider four loop unroll factors, this increases the number of transformations considered to 13. We then exhaustively evaluated all transformations sequences of length 5 selected from these 13 options. So, in theory, the total number of possible transformation sequences is given by the combinational expression $A_{13}^5 = \frac{13!}{(13-5)!} = 154440$, since no transformation can appear twice in the sequence and the order of transformations has an influence on performance. However, since unrolling can only appear once in any sequence (only one possible unroll factor), it decreases the total number of possible sequences we evaluated to 88000 per benchmark.

4.3 Platforms

Most of the results in Section 5 are provided for a TI processor architecture described below. However, in order to show that our approach is not specific to an architecture, we also provide results for an embedded AMD processor architecture in Section 5.5; this AMD processor is based on a MIPS core, so we will later refer to it as `MIPS`. The platforms are detailed below.

TI: The Texas Instrument C6713 is a high-end floating point DSP, running at 300MHz. The wide clustered VLIW processor has 256KB of internal memory. The programs were compiled using the TI's *Code Composer Studio Tools* Version 2.21 compiler with the highest `-O3` optimization level and `-ml3` flag (generates large memory model code).

MIPS: The AMD Alchemy Au1500 processor is an embedded SoC processor using a MIPS32 core (Au1), running at 500MHz. It has 16KB instruction cache and 16KB non-blocking data cache. The programs were compiled with GCC 3.2.1 with the `-O3` compile flag. According to the manufacturer, this version/option gives the best performance - better than later versions of GCC - and hence was used in our experiments.

4.4 Training the model

In our experiments we vary the training set size to consider its impact on performance. In all cases, this is performed using "leave one out" cross-validation, a standard technique for evaluating ANNs and other predictive models. Basically, this means that, if we have N programs, we select one program whose performance we wish to predict with respect to transformations. We then use data from the $N-1$ remaining programs to train our ANN; before testing it on the selected Nth program; we repeat this procedure for the N programs and average results. Thus, we do not train the ANN on data associated with the program whose performance we wish to predict. Our ANNs are trained by standard back propagation on the mean squared error.

The training inputs are the canonical speedups (see Section 3.4) and randomly picked transformation sequences from the transformation space. The training outputs are the corresponding relative improvements over the baseline performance. For statistical significance, we repeat experiments 30 times and show the average performance results and the corresponding variances.

Finally, note that for the reactions approach, in addition to the training set, we need one more simulation/execution of the baseline program to compute the speedups (ratio of baseline over transformed versions) of the canonical transformations, which are used as inputs to the model.

5. EXPERIMENTAL RESULTS

This section first evaluates the accuracy of reaction-based modeling across the program transformation space and also examines its accuracy in predicting high-speed up optimizations. This is followed by a short evaluation of the standard feature-based approach to modeling. Next we consider the trade-off between accuracy of prediction and the number of training examples we use. This is followed by an evaluation of how the number of canonical transformations also affects predictive power. Finally, we try a simple cross-platform study where we train on one platform and use it to predict performance on a new unseen machine.

5.1 Reaction-Based approach

In this section, we use a training set of 64 runs per benchmark, i.e., this means we have randomly selected 64 program transformations and apply them to each of the $N - 1 = 10$ benchmarks to build a predictor. As mentioned before, we repeat this process 30 times to be statistically meaningful. Also, we have set the number of canonical transformations to 4. After training, these 4 canonical transformations are applied to the Nth unseen program. The reactions (speedups) to these canonical transformations, as well as the transformation identifier, are used as inputs to the model, along with the execution of the original, untransformed program. The model is used to predict the speedup of all of the remaining $88000 - 4$ transformation sequences of the unseen Nth benchmark (i.e., "leave one out"). Thus we have a training set of overall size $64 \times 10 + 4$ (644) to predict $88000 - 4$ data points.

We compare our model against a *naive predictor*. The naive predictor simply finds the average performance of the training data (all program-transformation pairs in the samples) and predicts that any new point will be that value, i.e., the naive prediction is $\sum_{i=1}^{640} s_i/640$ where s_i is the real speedup of one of the transformations.

In order to describe the accuracy of the model, we use the *Mean prediction error* defined as $\frac{1}{88000-4}\sum_{i=1}^{88000-4}\frac{|\hat{s}_i - s_i|}{s_i}$, where s_i is the real speedup of one of the transformation points and $\hat{s_i}$ the predicted value. Averaged over all $88000 - 4$ transformations applied to the unseen benchmark, this gives an overall average measure of the error. As mentioned in Section 2, we have collected the actual speedups for all benchmarks and all transformations allowing us to calculate the MAE exactly.

Figure 5 shows the mean prediction error of our predictor for each program, as well as the naive predictor. As can be seen, the overall mean error is 7.3% when using a reaction-based predictor. If we compare this to the naive predictor, whose error is 15.8%, it is twice as accurate. Since our model is trained on random samples, its accuracy can typically vary from one sample set to another. For that reason, we have evaluated the standard deviation of the mean prediction error over the 30 different trials used for evaluating the models in Figure 5, see Section 4. The standard deviation in this figure shows that the model accuracy is quite stable. In fact, we note that by using the reaction-based approach, not only we outperform other predictors we are considering here. We also get significantly lower variances than by using the code feature-based models. Our method is therefore much less sensitive to the specifics of initialization or choices of training data.

While the average error and standard deviation over all program transformations is a good indication of model accuracy, for practical usage, it is important to predict well the best performing transformations most of the time. For that reason, we have evaluated the model accuracy on (1) all transformations which bring more than 5% speedup, and (2) the transformations within 5% of the maximum achievable (according to the actual speedup). The results are reported in Figure 6 and are shown only for those benchmarks where we can obtain over 5% speedup. Even though the error is slightly worse over both transformations bringing more than 5% speedup (8.1% error) and within 5% of max transformations (8.5% error) than over all transformations (7.3%), the model is still fairly accurate on these critical transformations, and again, it is significantly more accurate than the naive predictor on the same transformations. Thus, our method is a practical alternative for fast software exploration.

5.2 Feature-Based approach

In this section, we compare the reaction-based approach against the more standard feature-based approach. Even though the reaction-based approach significantly outperforms the feature-based approach, feature selection for whole-programs is still in its infancy [1] and future improvements in feature selection may improve its accuracy. As a pragmatic starting point we generated many different features used in other research [1], as listed in Table 2. As mentioned before, we collect these features after applying the program transformations in SUIF, in order to capture the static impact of transformations on static program characteristics.

For a fair comparison with reactions, we have trained the feature-based model on the same $640 + 4$ samples as for the reactions model. These results are reported in Figure 5, along with the reactions results. The feature-based predictor not only performs worse than reaction-based predictor, but it performs worse than the naive predictor in many cases. Even though static features have proved useful for tuning single program optimizations, such as unrolling, and on small code constructs [26], performance modeling of whole programs, even small ones, is a more difficult task. Nevertheless, in future work, we hope to improve the static features definition to better capture whole program behavior, and also to combine static features and reactions in a hybrid approach. In the remaining sections, we therefore focus our investigations and experiments on the reactions approach.

5.3 Speeding up training: accuracy vs. training size

Considering there are 88000 possible transformations per benchmark, and that we need no more than 64 samples per benchmark to obtain a fairly accurate model, we can reduce transformation space exploration to $\frac{1}{3753}$th of its total size. Note that, once the model is trained with a few benchmarks, then we only need to probe new benchmarks with canonical transformations, speeding up exploration by several orders of magnitude. In fact, while we used 10 benchmarks for our training set in each experiment, it does not mean the model cannot be used after training on a smaller (nor a larger) number of benchmarks. In Figure 7, we have evaluated the model accuracy when varying the number of training benchmarks from 1 to 10. After training on two benchmarks, the model accuracy is not much better than that of the naive predictor, however, after training on 5 benchmarks, the error is down to 10.6%.

Throughout the article, we have used randomly selected training runs. However, in practice, software engineers may decide to do specific runs anyway; these runs can add to or even replace the random training runs, to further improve the performance model. There is naturally a trade off between the number of training runs and the accuracy of the model. The smaller the number of runs, the faster the model is built, but potentially, the lower the accuracy. To evaluate this trade off, Figure 8 shows prediction accuracy against training set size (the number of training runs). As mentioned before, the training runs are randomly selected among the set of all possible program-transformation pairs except for the target program.

Consider the bars corresponding to 4 canonical transformations (the trend is similar for other values). The mean prediction error is large, 13.4%, when allowing only 16 samples per benchmark to build the predictor. However, starting at 32 samples per benchmark, the model accuracy is good enough for practical usage (9.2%). It reduces to just 7.3% when using 64 samples per benchmark. (We

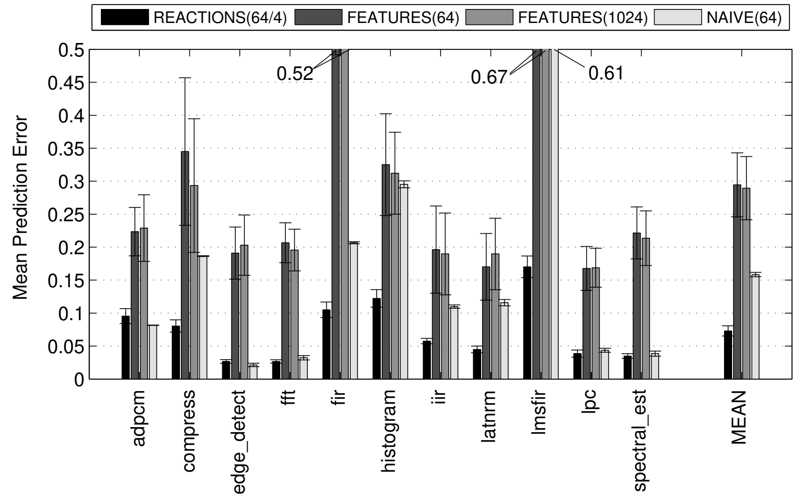

Figure 5: ***Mean prediction error: Reactions, Features and Naive predictors. Averages over 30 trials, 64 training patterns per benchmark, except for FEATURES(1024) which uses 1024 training patterns. 4 canonical transformations were applied for the reaction-based approach.***

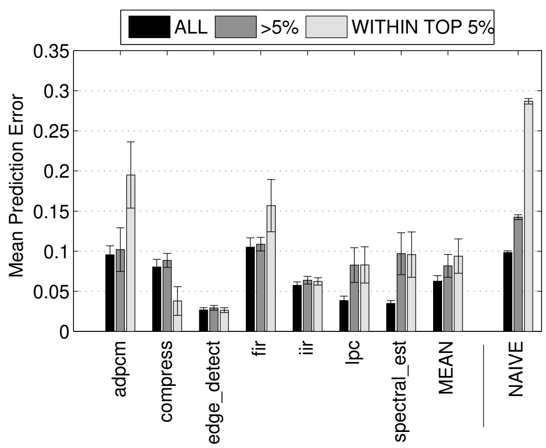

Figure 6: ***Reactions and Naive predictors: transformations with more than 5% speedup, and within 5% of maximum (actual) transformations, 64 training patterns per benchmark.***

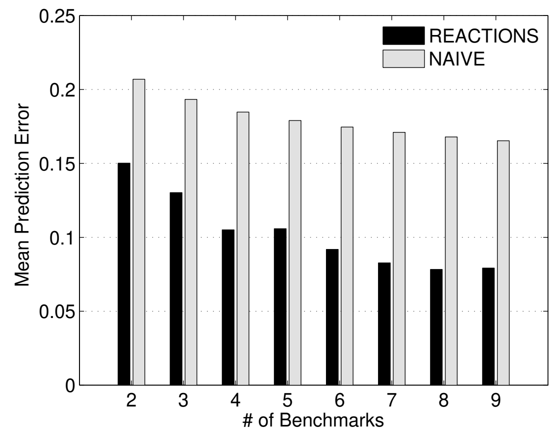

Figure 7: ***Impact of number of training benchmarks, 64 training patterns per benchmark.***

note here that termination of training conditions does not depend on the size of the training set.) Interestingly, the accuracy of the predictor does not increase significantly beyond this sample size. This confirms the remark of Section 2 that there are only few performance "plateaus" for each benchmark, and once all plateaus have been covered by training samples, additional training is unnecessary.

We also note that there are interesting effects when varying the size of training data on accuracy of the predictions. This may indicate the performance limits of the considered predictive models. It is also possible that our model may be "overfitting" to the training benchmarks, which may inhibit the performance on new benchmarks. In the future we are planning to formally investigate these effects for our approach to learning.

5.4 Impact of canonical transformations

5.4.1 Number of canonical transformations vs. prediction error

As stated earlier, the canonical transformations, which enable program discrimination by focusing on the most meaningful reactions for a given model/platform, are selected automatically. Figure 8 shows the impact of selecting 1 to 8 canonical transformations for the predictor. As expected, increasing the number of canonical transformations generally improves the prediction accuracy across the various training sizes. Also, there is a limit (4) beyond which increasing the number of canonical transformations brings little benefit. We also note that models using 8 canonical transformations sometimes perform slightly worse than models with 4 transformations or less. This may indicate that some of the extra features provide little additional information about the underlying program. At the same time, these extra features may be sensitive to the noise or systematic bias in training data, which may potentially adversely affect performance on new programs. Nevertheless, we can see that

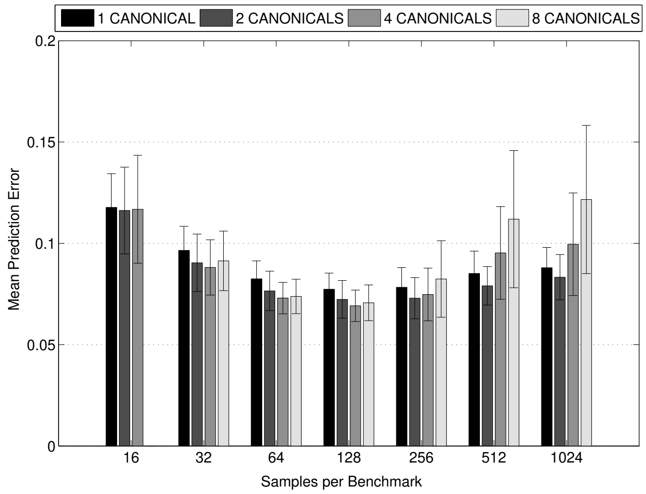

Figure 8: ***Impact of training size and number of canonical transformations for reaction-based model.***

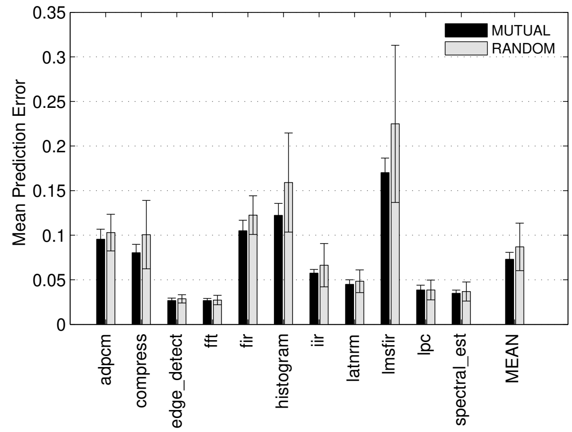

Figure 9: Mutual Information ***selection vs. random selection of canonical transformations for reaction-based model, 64 training patterns per benchmark.***

our reaction-based models significantly outperform predictors (of similar complexity) which only use code features – see Figure 5.

Note however that, even with 1 reaction, the model already performs quite well (consider the bar for 64 samples and 1 canonical transformation). It turns out that, since our set of benchmarks is much smaller than the set of possible transformations, it is easy to find one transformation which almost fully discriminates benchmarks. Still, some benchmarks, such as `fir`, significantly benefit from 4 rather than 1 or 2 canonical transformations.

Finally, one can see that 32 samples/8 canonical transformations performs about as well as 64 samples/2 canonical transformations. So one can choose, depending on the experimental constraints, to either minimize the training set or the number of probes.

5.4.2 Mutual Information selection vs. random selection

In order to evaluate the benefit of the *Mutual Information* approach, we have compared it with a random selection of canonical transformations, in Figure 9. We note that using mutual information features is consistently better than using random features, and sometimes significantly so. Note that a careful choice of canonical transformations appears to be particularly important for difficult benchmarks, i.e. those where the prediction errors are relatively high for both code feature-based and reaction-based approaches (e.g. `fir`, `lmsfir`, `compress`, `histogram`). From Figure 9, we can also see that by using the mutual information (rather than random) features, we get much lower variances of the prediction errors. This difference in variances is consistent across all the benchmarks, and is particularly large for the more complicated programs.

5.5 Modeling another architecture

Figure 10 shows the main accuracy results for another architecture platform, a MIPS core described in Section 4. Note that the average accuracy of our model is almost the same as the TI platform. We also see that the naive predictor performs better for MIPS than for TI, in large part because the MIPS core (and the underlying compiler) is significantly simpler than the TI core (simple scalar versus large VLIW processor).

Finally, since a significant number of transformations can have a similar impact on different architecture platforms, we have explored whether we could apply on a new platform what was learned on another platform. Figure 11 shows the accuracy of a model trained on the MIPS, on 10 benchmarks (64 samples per benchmark, 4 canonical transformations), and used to predict performance of the left out benchmark running on the TI platform. While not as good as the native TI model, it still outperforms the naive predictors.

6. RELATED WORK

6.1 Speeding up simulators and alternative approaches

Recently, there have been proposals to speed up simulation using sampling, e.g., SimPoint [25] and SMARTS [31]. Even more recently, TurboSMARTS [29] could drastically reduce overall simulation time through a combination of sampling and checkpointing. These techniques are actually orthogonal and complementary with our technique: by reducing the time necessary to perform one run, they can reduce our training and characterization time. However, sampling techniques like SimPoint or SMARTS are ill-suited for software design-space exploration because they require a significant pre-processing effort for each benchmark (at least one full functional simulation), so that, after any program transformation, the whole pre-processing must be replayed, voiding part or all of the speed benefits of sampling.

Karkhanis *et al.* [17] propose an analytical model for hardware exploration that captures the key performance features of superscalar processors. This model can potentially be used for software exploration, but the construction of the model is ad hoc and a complex process, which makes it difficult to generalize and replicate. Eeckhout *et al.* [10] use statistical simulation to similarly capture processor characteristics, and generate synthetic traces that are later run on a simplified superscalar simulator. After any program transformation, a new trace needs to be generated if this approach were to be used for software exploration, requiring a full functional simulation.

Recently Ipek *et al.* [15] proposed a distinct method for both considerably speeding up and automating the hardware design-space exploration process. The principle is to train an ANN (Artificial Neural Network) to predict the impact of hardware parameter variations (e.g., cache size, memory latency, etc) on the performance behavior of a target architecture. After training on less than 1%

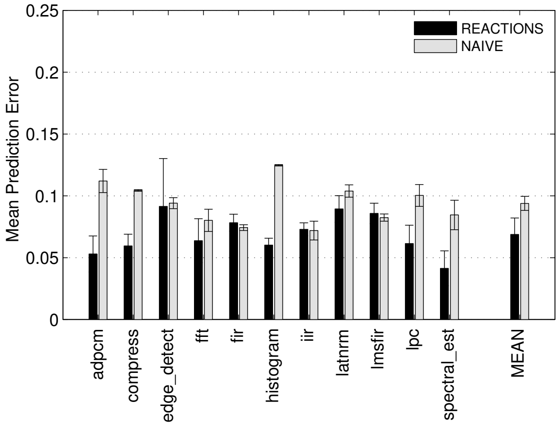

Figure 10: ***MIPS mean prediction error: Reactions and Naive predictors, 64 training patterns per benchmark.***

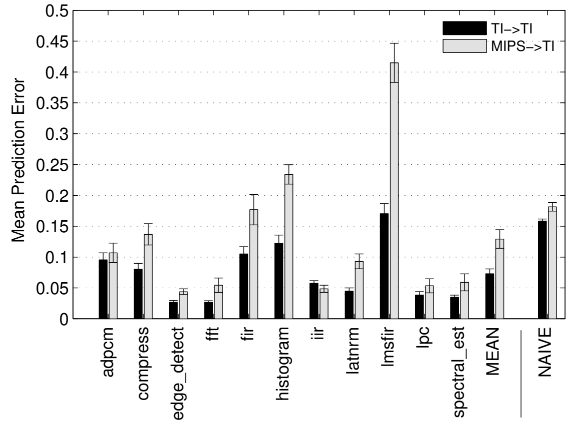

Figure 11: ***Learning across architectures, using the reaction-based model, 64 training patterns per benchmark.***

of the design space, the model can still accurately predict performance variations with less than 5% error. One model is built for each benchmark, i.e., the model does not learn *across* benchmarks as we do. Moreover, the approach is limited to hardware because the model is specific to a program binary, just like the sampling approaches. Therefore, any modification of the program binary, such as applying a program transformation, requires training a new model using several thousands simulations. As a result, the approach is not suitable for software exploration. Our approach also uses machine-learning to build a performance model, but it can accommodate any program transformation without retraining.

Recently, IBM has highlighted the issue of software tuning early on in the design cycle. For the Blue Gene/L, it was shown that an approximate but fast performance model, as a replacement for more detailed but slow simulators, can be very useful in practice [3]; still, the approximate model proposed was designed manually and in an ad-hoc manner.

6.2 Predicting the impact of program transformations

Machine-learning has recently been investigated by a number of researchers in the area of compiler optimization. The goal is usually to improve or replace one or more hand-tuned compiler heuristics. Such machine learned heuristics attempt to predict a good transformation, but do not predict their actual performance.

Stephenson *et al.* [27] used genetic programming to tune heuristics for three compiler optimizations within the Trimaran's IMPACT compiler: hyperblock selection, data prefetching and register allocation. Cavazos *et al.* [7] describe using supervised learning to control whether or not to apply instruction scheduling. Monsifrot *et al.* [20] use decision trees to determine which loops to unroll: they looked at the performance of compiling Fortran programs from the SPEC benchmark suite using g77 for two different architectures. Stephenson *et al.* [26] use machine-learning to characterize good unroll loop factors for a given loop nest, and improve over the ORC compiler heuristic. All of these approaches are successful in automatically generating compiler heuristics for code segments rather than in predicting the eventual performance of selected optimizations for whole programs.

Rather than predicting the impact of a single transformation, others have looked at searching [28, 12, 2, 8, 18, 21] for the best set or sequence of optimizations for a particular program.

Almagor *et al.* [2] propose a number of algorithms to solve the compilation phase ordering problem. Their techniques search for the best phase order of a particular program. Such approaches gives impressive performance improvements, but has to be performed each time a new program is compiled. In contrast, our models are constructed on a training set of programs and then can be used to accurately predict transformations to "unseen" programs.

Cooper *et al.* [8] present a system called ACME which speeds up the search of iterative compilation. ACME utilities a technique called estimated virtual execution (EVE) which estimates changes to the execution counts of basic blocks when an optimization that changes the control flow graph (CFG) is applied. This technique then simply models the benefits and disadvantages of applying optimizations by estimating all the changes to the CFG. EVE works well for estimating code size or for estimating the performance of simple processors, but it is too inaccurrate to model the performance of real machines. This technique also requires substantial changes to a compiler to understand the impact optimizations have on the control flow graph.

Kulkarni *et al.* [18] introduce techniques to allow exhaustive enumeration of all distinct function instances that would be produced from the different phase-orderings of 15 optimizations. This exhaustive enumeration allowed them to construct probabilities of enabling/disabling interactions between the different optimization passes. Using these probabilities, they constructed a *probabilistic batch compiler* that determined which optimization should be applied next depending on which one had the highest probability of being enabled. This method however does not consider the benefits each optimization can potentially provide. In contrast, we train our models to obtain the impact of optimizations applied, and therefore our technique learns which optimizations are beneficial to apply to "unseen" programs with *similar* characteristics. However, the techniques presented in this work would allow a larger exploration of the optimization space than we attempted. By exploring a larger part of the search space, we would likely improve the training data used for our models.

Pan *et al.* [21] partitioned a program into *tuning sections* and then developed fast techniques to find the best combination of optimizations for each of these tuning sections. They are able to reduce the time to find good optimization settings from hours to minutes. Again, these techniques could also be beneficial during the training data generation stage of our models. Specifically, the technique to test different optimization settings on a tuning section during a single run of the program would allow us to increase the number of optimization settings we evaluate.

Agakov *et al.* [1] build models of good transformation sequences from training data on a per program basis. This is then used to guide iterative search on a new program. Unlike this paper, they only attempt to predict good transformations to apply rather than predicting the *performance* impact of any particular transformation.

In the area of predictive modelling, Zhao *et al.* use manually constructed cost/benefit models to predict whether to apply PRE or LICM [34]. They achieve 1% to 2% improvement over always applying an optimization, but at a cost of greatly increasing compilation time (by up to 68%). However, their models appear to be quite complicated and have to be manually constructed. On the other hand, our models are simple and automatically constructed using machine learning.

Iterative optimization has also been employed in well-known library generators in such systems as FFTW [13], ATLAS [30], and SPIRAL [23]. These systems obtain excellent performance on the particular set of applications they tune, e.g., linear algebra libraries or DSP codes, however they require a large amount of search every time a new processor is targetted.

Yotov *et al.* [32] describe a model-based approach for optimizing BLAS libraries. They show that using a model-based approach to evaluate the performance of an optimization can be as effective as empirical evaluation. Epshteyn *et al.* [11] present a hybrid approach that uses this analytical model approach to quickly obtain information about individual search points. The search points that the model predicts will have the highest expected performance are evaluated on the real machine and used to refine an empirical model being constructed. The approach obtains performance comparable to expensive empirical search techniques and significantly outperforms techniques based solely on analytic models. Yotov *et al.*[33] present a technique of refining analytical models based on the results of empirical search. That is, the authors analyze the code and optimization parameters found by ATLAS through its global search and then make refinements to their analytical models. The authors advocate the use of local neighborhood search around the points suggested by the analytical model to further improve the solutions. Results show the refined analytical models and models with local search perform comparable to the global search strategy found in ATLAS. However, the analytical models presented in these three approaches are complicated and require extensive manual tuning. In contrast, our models are automatically constructed and have the potential to outperform hand-tuned models.

6.3 Program characterization for prediction

In order to predict the effect of a transformation on a given program, the predictor is fed some characteristics of the target program. Much of the prior work in machine learning based compilation relies on *program feature*-based characterization. For instance, Monsifrot *et al.* [20], Stephenson *et al.* [26] and Agakov *et al.* [1] all use static loop nest features. Features may capture those characteristics of the static program that are best at predicting program transformations to apply.

However, we have shown that feature-based characterization may not be well suited to the complex task of predicting whole program performance and the impact of many different transformations. Triantafyllis *et al.* [28] bears some similarity with our *reaction*-based characterization method. They augment the Intel Itanium compiler with the ability to iteratively search combinations of compiler options across runs for a given program, and they especially focus on the interactions among compiler options. As part of their technique, they collect the good combinations of compiler optimizations by noticing that how a program behaves for one transformation can be an indication of how it would behave for some other transformations. Our reaction characterization method is based on a similar intuition, however they attempt to find appropriate optimizations, while we also attempt to estimate the associated speedups and to scan the whole transformation space very rapidly.

7. CONCLUSIONS AND FUTURE WORK

This article proposes a method for building a performance model of a target machine which is accurate enough to estimate the speedup of any known program transformation. One of the key assets of our approach is that the model construction is entirely automatic; the main construction cost is the training phase, though training runs can be either randomly selected or resulting from past/useful experiments. The performance model is based on characterizing programs by their reactions to a set of automatically selected canonical transformations. This approach has been shown to accurately capture the complex interplay between the program and the architecture.

Hybrid reactions+features approach. Even though the reaction-based approach proved superior to the feature-based approach, and required only a few probing runs on the new target program, the feature-based approach still has the potential practical advantage of not requiring any probing run of the new target program, since the characterization is static. As a result, we intend to investigate a combined reactions+features approach in order to achieve the accuracy of the reactions approach with the practical advantages of the features approach.

Extensions to other combined software+hardware predictions. Our reactions approach makes no assumption on the *nature* of reactions. In this study, reactions are the impact of software transformations on performance, but they could also be the impact of hardware modifications on performance. As a result, we will extend our approach to combined software+hardware design-space exploration by simultaneously studying software and hardware reactions.

Acknowledgements: We thank Chris Williams for helpful discussions.

8. REFERENCES

[1] F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[2] L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 231–239, 2004.

[3] L. Bachega, J. Brunheroto, L. DeRose, P. Mindlin, and J. Moreira. The bluegene/l pseudo cycle-accurate simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.

[4] H. Berry, D. G. Prez, and O. Temam. Chaos in computer performance. *Chaos*, 16(1), Dec. 2005.

[5] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 2005.

[6] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Chapman and Hall, 1984.

[7] J. Cavazos and J. E. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[8] K. D. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, , and T. Waterman. Searching for compilation sequences. Tech. report, Rice University, 2005.

[9] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, New York, 1991.

[10] L. Eeckhout, R. H. B. Jr., B. Stougie, K. D. Bosschere, and L. K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 350–363, 2004.

[11] A. Epshteyn, M. Garzaran, G. DeJong, D. Padua, G. Ren, X. Li, K. Yotov, and K. Pingali. Analytic models and empirical search: A hybrid approach to code optimization. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Hawthorne, NY, USA, 2005.

[12] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.

[13] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[14] M. Hall, L. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, M., and Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1999.

[15] E. Ipek, S.A. McKee, B.R. de Supinski, M. Schulz, and R. Caruana. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, San Jose, CA, USA, 2006. ACM Press.

[16] R. A. Jacobs, M. I.Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive Mixtures of Local Experts. *Neural Computation*, 3, 1991.

[17] T. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 338–349, 2004.

[18] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 12–23, 2003.

[19] C. Lee. Utdsp benchmark suite. In *http://www.eecg.toronto.edu/˜corinna/DSP/infrastructure/UTDSP.html*, 1998.

[20] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications*, LNCS 2443, pages 41–50, 2002.

[21] Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *IEEE PACT*, Seattle, WA, September 2006. IEEE Computer Society.

[22] D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Toward a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *Proceedings of the International Conference on Supercomputing*, 2004.

[23] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[24] M. Saghir, P. Chow, and C. Lee. A comparison of traditional and vliw dsp architecture for compiled dsp applications. In *Proceedings of the International Workshop on Compiler and Architecture Support for Embedded Systems (CASES)*, Washington, DC, USA, 1998.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM Press.

[26] M. Stephenson and S. P. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, pages 123–134, 2005.

[27] M. Stephenson, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–90, 2003.

[28] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.

[29] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Turbosmarts: accurate microarchitecture simulation sampling in minutes. In *Proceedings of the ACM SIGMETRICS*, pages 408–409, 2005.

[30] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[31] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 84–95, 2003.

[32] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 63–76, 2003.

[33] K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 141–150, New York, NY, USA, 2005. ACM Press.

[34] M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: an approach for profit-driven optimization. In *Third Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, pages 317–327, 2005.