Portable Compiler Optimization Across Embedded Programs and Microarchitectures using Machine Learning

Abstract

Building an optimizing compiler is a difficult and time consuming task which must be repeated for each generation of a microprocessor. As the underlying microarchitecture changes from one generation to the next, the compiler must be retuned to optimize specifically for that new system. It may take several releases of the compiler to effectively exploit a processor's performance potential, by which time a new generation has appeared and the process starts again.

We address this challenge by developing a portable optimizing compiler. Our approach employs machine learning to automatically learn the best optimizations to apply for any new program on a new microarchitectural configuration. It achieves this by learning a model off-line which maps a microarchitecture description plus the hardware counters from a single run of the program to the best compiler optimization passes. Our compiler gains 67% of the maximum speedup obtainable by an iterative compiler search using 1000 evaluations. We achieve, on average, a 1.16x speedup over the highest default optimization level across an entire microarchitecture configuration space, achieving a 4.3x speedup in the best case. We demonstrate the robustness of this technique by applying it to an extended microarchitectural space where we achieve comparable performance.

1 Introduction

Creating an optimizing compiler for a new microprocessor is a time consuming and laborious process. For each new microarchitecture generation the compiler has to be retuned and specialized to the particular characteristics of the new machine. Several releases of a compiler might be needed to effectively exploit the processor's performance potential, by which time the next microarchitecture generation has been developed and the process starts again. This never-ending game of catch-up means that we rarely exploit a shipped processor to the full and this inevitably delays the time to market. Although this is a general issue for all processor domains, it is particularly acute for embedded systems. Ideally, we would like a portable compiler technology that provides retargetable optimization while fully exploiting the characteristics of the new microarchitecture. In other words, given any new processor generation, deliver a compiler that automatically optimizes for that new target and achieves high performance.

Building such a compiler is, however, extremely challenging. This is primarily due to the complexity of the underlying machine's behavior and the varying structure of the programs being compiled. Iterative compilation, which tunes each new program on a specific architecture [6, 16, 24, 31], has provided a methodology to find good optimizations. Techniques such as genetic algorithms [24], hill climbing [2] or optimization orchestration [31] have been explored, all showing impressive performance improvements. Although useful, these approaches all suffer from the large number of compilations and execu-

tions required to optimize each program. Every time the program or architecture changes, this time-consuming process must be repeated.

In order to overcome these challenges, researchers have developed compilers that learn optimization strategies using prior knowledge of other programs' behavior. Stephenson *et al.* [35] showed that genetic programming can learn good individual compiler optimizations on a fixed architecture, eliminating the need for any iterative compilations of the new program. Cavazos *et al.* [3] showed that this could be used to learning the best set of compiler options on a fixed architecture. Although these approaches dramatically reduce or even eliminate the need for extra compilations and executions of the target program, they suffer from the need to entirely retrain the compiler whenever the platform changes.

In this paper we develop, to the best of our knowledge, the first truly portable optimizing compiler. Given a new microarchitecture, our approach automatically determines the right optimization passes for any new program. Our scheme learns a machine learning model off-line which maps a microarchitecture description plus the hardware counters from a single run of the program to the best compiler optimization passes. The learning process is a one-off activity whose cost is amortized across all future users of the compiler on subsequent variations of the processor's microarchitecture.

Using this approach we can, on average, achieve a 1.16x speedup over the highest default compiler optimization across 200 microarchitectural configurations. We further show that this approach achieves 67% of the maximum performance improvement gained by standard iterative compilation search using 1000 evaluations. Given our approach, a new compiler does not need to be tuned whenever the processor microarchitecture changes or a new program needs compiling. This allows compilers to become fully integrated into the design space exploration of a new processor generation, helping designers to fully evaluate the potential of any new microarchitecture. Overtime, designers may wish to add new microarchitectural features not originally envisaged. We show that our approach adapts to new microarchitectural configurations and is able to deliver the same level of performance.

In summary, this paper makes the following contributions:

- We develop a machine learning model that can predict the best optimization passes to use for any new program when compiling for a new microarchitecture;
- We show how our scheme accurately delivers the performance improvements available across the MiBench benchmark suite and an embedded microarchitectural design space;
- We demonstrate the robustness of our scheme showing that it delivers comparable performance on an extended microarchitecture space.

The next section provides a short example demonstrating the difficulty of achieving portable optimization. Section 3 provides a description of how our compiler is trained and deployed using machine learning. Section 4 describes the experimental setup. Section 5 then evaluates the technique and analyzes the results. Section 6 evaluates this approach on a new extended space. This is followed by a description of related work in section 7. Finally, section 8 concludes the paper.

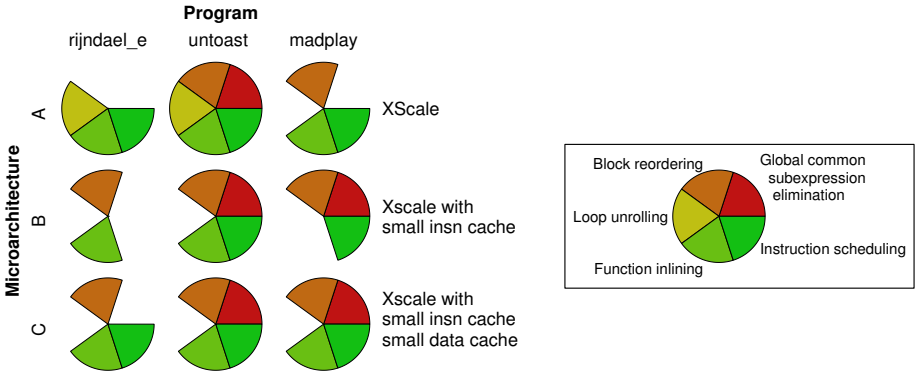


Figure 1. Segment diagrams representing the optimization passes to enable in order to achieve the highest performance for three programs executed on three microarchitectures. A filled segment means that the optimization should be enabled, an empty one means it should be disabled.

2 Example

In this paper we limit our study to selecting the right passes within an existing compiler framework for varying programs and microarchitectures. Although this may seem like a restricted setting, the best optimization passes to apply vary significantly between programs and microarchitectural configurations. Finding the best set of optimization passes across programs and microarchitectures is highly non-trivial.

To illustrate this point, consider figure 1 which shows segment diagrams for three programs (*rijndael_s, untoast* and *madplay*) on three microarchitectures from our design space (described in section 4). For these three programs and microarchitectures, we found the best optimization passes to apply (described in section 4.2.1). These optimizations lead to significant speedups, ranging from 1.16x to 2.62x speedup over the highest default optimization level. In this example we show only five significant optimization passes: *block reordering, loop unrolling, function inlining, instruction scheduling* and *global common sub-expression elimination*, labeled on the right of figure 1. For each program/microarchitecture pair there is a circle of five segments representing the five passes. If the segment is filled, then the corresponding optimization should be enabled for the given program and microarchitecture. If empty, it should be disabled.

What is immediately clear is that the best set of optimization passes to apply changes across programs and microarchitectures. If we consider *madplay* for instance, three optimizations should be enabled for microarchitecture *A*, a different set of three for *B* and four enabled for configuration *C*. If we now consider microarchitecture *B*, two optimizations should be turned on for *rijndael_e*, four when compiling *untoast* and only three for *madplay*. Given the large number of optimizations available in a typical compiler, providing a portable optimizing compiler for programs and microarchitectures is non-trivial.

However, there are similarities between programs and microarchitectures that we can exploit. The best set of optimizations for *rijndael_e* on configuration *C* and *madplay* on microarchitecture *A* are exactly the same. This is also true for *untoast* on microarchitectures *B* and *C* and *madplay* on configuration *C*. If we can somehow *characterize* the program *madplay* on configuration *A* and relate it to the characteristics of *rijndael_e* on microarchitecture *C*, then we can apply the same

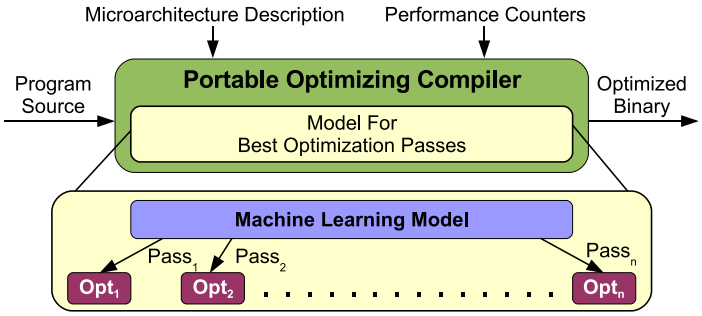


Figure 2. Overview of our portable optimizing compiler. The compiler takes in a program source, some performance counters and a microarchitecture description and outputs an optimized program binary for the microarchitecture. At the heart of the compiler is a machine learning model that predicts the best passes to run, controlling the optimizations applied.

optimization passes to *madplay* as we did to *rijndael_e*. This will allow us to obtain the best speedups on this new program/microarchitecture pair without having ever seen *madplay* or configuration *A* before. In the next section we develop a machine learning model that automatically identifies these similarities. We then use and evaluate it in section 5 to provide portable optimization across programs and microarchitectures.

3 Enabling Portable Optimization

As seen in the previous section, the best performance is achieved by applying different optimizations depending on the program and the underlying microarchitecture. This means that with current approaches to tuning an optimizing compiler [1, 3, 30, 35] a new compiler needs to be developed for each new generation of the microarchitecture. To overcome this problem, we develop a machine learning model that automatically adapts the compiler's optimization strategy for any program and any microarchitecture.

3.1 Overview

Figure 2 gives an overview of our compiler's structure. The tool works like any other compiler, taking as an input the source code of a program and producing an optimized binary. However, in addition to the source code our compiler has two other inputs which it uses internally to optimize the program specifically for the microarchitecture it will run on.

Firstly, our compiler takes in a description of the microarchitecture to target. This is similar to standard compilers where this description is hard-coded in a machine description file; here it is just an input. Secondly, it takes in performance counters derived from a previous run of the program. This is similar to feedback-directed compilers that typically use profiling information from a previous run to generate an optimized version of the program. However, unlike any existing technique, our compiler generates an optimized binary specifically for the target microarchitecture even when it has never seen the program or the microarchitecture before. Therefore, the compiler does not have to be modified or regenerated whenever a new program or microarchitecture is encountered.

Performance Counter		
Instructions per cycle	Insn cache access rate	ALU usage
Decoder access rate	Insn cache miss rate	Mac usage
Register file access rate	Data cache access rate	Shifter usage
Branch pred. access rate	Data cache miss rate	

Table 1. Performance counters used as a representation of program/microarchitecture pairs.

At the heart of our compiler is a model that correlates the behavior of the new input program and microarchitecture with programs and microarchitectures that it has previously seen. Such a model is built using machine learning and our approach can be considered as a three stage process: generating training data, building a model and deploying it. The next three subsections describe each of these activities in detail, allowing us to create the overall compiler shown in figure 2.

3.2 Generating Training Data

In order to build a model that predicts good optimization passes, we need examples of various optimization passes on different programs and microarchitectures as well as a description of each program and microarchitecture. We generate this training data by evaluating N different sets of optimization passes, $\mathbf{y}$, on a set of training program/microarchitecture pairs, $X^1, \ldots, X^M$, and recording their execution times, t. We can characterize a program/microarchitecture pair using a vector of features $\mathbf{x}^1 \ldots, \mathbf{x}^M$. Therefore, for each program/microarchitecture pair X^j we have an associated dataset $\mathcal{D}^j = \{(\mathbf{y}^i, t^i)^j\}_{i=1}^N$, with $j = 1, \ldots M$. Our goal is to predict the best set of optimization passes $\mathbf{y}^*$ whenever a new program/microarchitecture X^* is encountered.

Although the generated dataset may be large, it is only a one-off cost incurred by our model. Furthermore, techniques such as clustering [32] are able to reduce this and is the subject of future work.

Features We characterize program interaction with the processor using 11 performance counters, $\mathbf{c}$, and with the microarchitectures using 8 descriptors, $\mathbf{d}$. The performance counters are shown in table 1 and are similar to those typically found in processor analytic models [12, 22] To capture the features of the microarchitecture we simply record its static description shown in table 2. The performance counters from a program running on a microarchitecture, $\mathbf{c}$, are concatenated together with the microarchitecture description, $\mathbf{d}$, to form a single feature vector for the program/microarchitecture pair, $\mathbf{x} = (\mathbf{c}, \mathbf{d})$.

3.3 Building a Model

Our aim is to build a model, $\mathcal{M}(\mathbf{x}, \mathbf{y})$, that provides the mapping from any set of program/microarchitecture features to a set of good optimization passes: $\mathcal{M} : \mathbf{x} \rightarrow \mathbf{y}$. We approach this problem by learning the mapping from the features, $\mathbf{x}$, to a *probability distribution over good optimization passes*, $q(\mathbf{y}|\mathbf{x})$. Once this distribution has been learned (see next section), prediction of a new program on a new microarchitecture is achieved by sampling at the mode of the distribution. Thus we obtain the predicted set of optimizations by computing:

$$\mathbf{y}^* = \underset{\mathbf{y}}{\operatorname{argmax}}\, q(\mathbf{y}|\mathbf{x}^*). \tag{1}$$

In other words, we find the value of $\mathbf{y}$ that gives the greatest probability of being a good optimization.

3.3.1 Fitting Individual Distributions

In order to learn the model we need to fit a probability distribution over good optimization passes to each training program/microarchitecture. Let $g(\mathbf{y}|X)$ be a *parametric* distribution specific to a program/microarchitecture pair X. Note that whereas $g(\mathbf{y}|X)$ is specific to the identity of a program/microarchitecture pair, $q(\mathbf{y}|\mathbf{x})$ allows generalization across programs and microarchitectures by being conditioned on a set of features $\mathbf{x}$.

Let $\widetilde{\mathcal{Y}}$ be a set of good optimization passes and $\tilde{p}(\mathbf{y}|X)$ be the empirical distribution over these passes[1] for program and microarchitecture pair X. We wish to fit the parametric distribution $g(\mathbf{y}|X)$ for each program/microarchitecture pair to be as close as possible to the empirical distribution $\tilde{p}(\mathbf{y}|X)$. To do this we can minimize the Kullback-Leibler (KL) divergence:[2]

$$\mathrm{KL}(\tilde{p}(\mathbf{y}), g(\mathbf{y})) = \left\langle \log \frac{\tilde{p}(\mathbf{y})}{g(\mathbf{y})} \right\rangle_{\tilde{p}(\mathbf{y})} = \text{constant} + H(\tilde{p}(\mathbf{y}), g(\mathbf{y})), \tag{2}$$

where $H(\tilde{p}(\mathbf{y}), g(\mathbf{y}))$ is the cross-entropy of $\tilde{p}(\mathbf{y})$ and $g(\mathbf{y})$. Thus, we can maximize the objective function:

$$\mathcal{L} = -H(\tilde{p}(\mathbf{y}), g(\mathbf{y})) = \sum_{\mathbf{y} \in \widetilde{\mathcal{Y}}} \tilde{p}(\mathbf{y}) \log g(\mathbf{y}). \tag{3}$$

In principle, our model's probability distribution $g(\mathbf{y})$ can belong to any parametric family. However, we have selected a very simple IID (independent and identically distributed) model, where the impact of each pass (y_ℓ) is considered to be independent of all others, i.e. $g(\mathbf{y}) = \prod_{\ell=1}^{L} g(y_\ell)$, where L is the number of available passes. If $g(y_\ell)$ is a multinomial distribution we have that:

$$g(\mathbf{y}) = \prod_{\ell=1}^{L} g(y_\ell) = \prod_{\ell=1}^{L} \prod_{j=1}^{|\mathcal{S}_\ell|} (\theta_\ell^j)^{I[y_\ell = s_\ell^{(j)}]}, \tag{4}$$

where $\mathcal{S}_\ell = \{s_\ell^{(1)}, \ldots, s_\ell^{(|\mathcal{S}_\ell|)}\}$ is the set of possible values that the pass y_ℓ can take (*i.e.* on, off or a parameter value); $I[y_\ell = s_\ell^{(j)}]$ is an indicator function that is 1 only when the particular optimization pass y_ℓ takes on the value $s_\ell^{(j)}$ (i.e. $I[y_\ell = s_\ell^{(j)}] = 1$ when $y_\ell = s_\ell^{(j)}$ and zero otherwise); and θ_ℓ^j is the probability of the optimization pass y_ℓ taking on the particular value s_ℓ^j (i.e. $\theta_\ell^j = p(y_\ell = s_\ell^{(j)})$); and, by definition, we have that $\sum_j \theta_\ell^j = 1$.

By using equations (4) in equation (3) and maximizing the latter with respect to each parameter θ_ℓ^j subject to the constraints $\sum_j \theta_\ell^j = 1$ we obtain:

$$\theta_\ell^j = \sum_{\mathbf{y} \in \widetilde{\mathcal{Y}}} \tilde{p}(\mathbf{y}) I[y_\ell = s_\ell^{(j)}]. \tag{5}$$

This result is known as the maximum likelihood estimator. Since we have used the uniform distribution as $\tilde{p}(\mathbf{y})$, this

[1] In our experiments we have chosen the set of "good" optimizations $\widetilde{\mathcal{Y}}$ to be those combinations of passes that are within the top 5% of all training optimizations for the respective program/microarchitecture pair. We have then weighted these optimizations uniformly.

[2] For the following derivation, to simplify the notation, we will omit the conditional dependency of these distributions on a specific program/microarchitecture pair X.

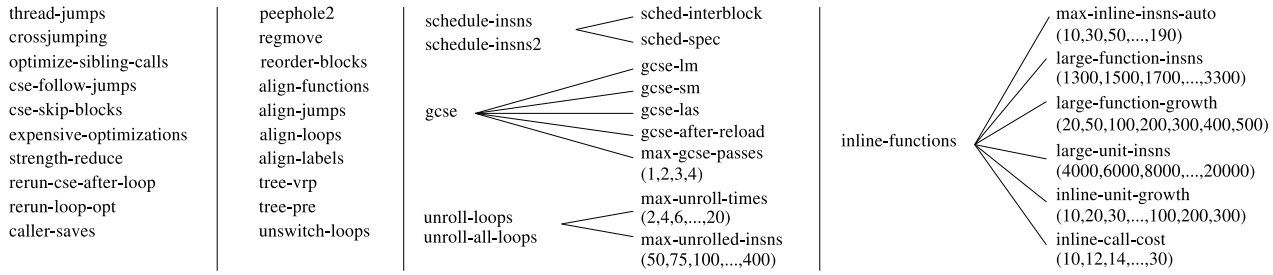


Figure 3. Compiler optimizations and their parameters. Each is a pass within gcc and can be varied independently. In total there are 642 million combinations.

means that the estimation of the parameters of the IID distribution θ_ℓ^j is the number of (selected) optimization passes in which $y_\ell = s_\ell^{(j)}$ divided by the total number of (selected) passes.

Our assumption of statistical independence between compiler optimizations may seem simplistic because it is well known that optimizations interact with each other. However, as we shall see in section 5, this IID model generalizes well across programs and microarchitectures. Our model works on the assumption that although compiler optimizations do interact, these interactions are less important across *good* sets of optimizations. Additionally, more complicated distributions, *e.g.* a Markov model, could be considered without modifying our approach.

3.3.2 Learning a Predictive Distribution Across Programs and Microarchitectures

Once the individual training distributions for each program/microarchitecture pair $g(\mathbf{y}|X)$ have been obtained, we can learn a predictive distribution $q(\mathbf{y}|\mathbf{x})$. This will predict the probability of each optimization pass value being good from the features, $\mathbf{x}$, of a program/microarchitecture pair (performance counters, $\mathbf{c}$, and microarchitecture descriptors, $\mathbf{d}$).

One possible way of learning this distribution is to use memory-based methods such as *K-nearest neighbors*. In other words, we can set the predictive distribution $q(\mathbf{y}|\mathbf{x})$ to be a convex combination of the K distributions corresponding to the training programs and microarchitectures that are closest in the feature space to the new (test) program and microarchitecture. The coefficients of the combination are obtained by using:

$$w^k = \frac{\exp\{-\beta d(\mathbf{x}^{(k)}, \mathbf{x}^{(*)})\}}{\sum_{k=1}^{K} \exp\{-\beta d(\mathbf{x}^{(k)}, \mathbf{x}^{(*)})\}}, \tag{6}$$

where β is a constant and $d(\cdot,\cdot)$ is our evaluation function, *i.e.* the euclidean distance of each corresponding nearest training point to the test point, so that the distributions of the closest training points are assigned larger weights. We have set $\beta = 1$ and $K = 7$ different neighbor programs, although we have found experimentally that the technique is not sensitive to similar values of K.

3.4 Deployment

Once the model is built, it can be used to predict the best optimization passes for any new program on any new microarchitecture, as shown in figure 2. It does this using just one run of the new program compiled with the default optimization level,

Parameter	Values	XScale	Parameter	Values	XScale
IL1 size	4K...128K	32K	DL1 size	4K...128K	32K
IL1 assoc.	4...64	32	DL1 assoc.	4...64	32
IL1 block	8...64	32	DL1 block	8...64	32
BTB entries	128...2048	512	BTB assoc.	1...8	1

Table 2. Microarchitectural parameters and their values. Each parameter varies as a power of 2, meaning 288,000 total configurations. Also shown are the values for the XScale processor.

O3, on the new microarchitecture. Thus, given a new program/microarchitecture pair, X^*, we extract its features by using the microarchitecture description, $\mathbf{d}^*$, and the performance counters from a run of this program (compiled with O3) on this microarchitecture, $\mathbf{c}^*$, so that we form $\mathbf{x}^* = (\mathbf{c}^*, \mathbf{d}^*)$. We then use equation (1) above to give the predicted-best optimization passes, $\mathbf{y}^*$, compile and execute the program with this new optimization.

4 Design Space

The previous section has developed a machine learning model that automatically predicts the correct optimization passes to apply for any new program on any microarchitectural configuration. This section describes our experimental setup later used to evaluate this model. It also provides a brief characterization of the compiler and microarchitectural design spaces.

4.1 Benchmarks

We chose to use MiBench [15], a common embedded benchmark suite, well suited to the microarchitecture space of this paper. It contains a mix of programs, from signal processing algorithms to full office applications. We used all 35 programs, running each to completion using an input set requiring at least 100 million executed instructions, wherever possible. Therefore, *susan_c*, *susan_e*, *djpeg*, *tiff2rgba* and *search* were run with the *large* input set, all others were run with the *small* inputs.

4.2 Microarchitecture Space

In this paper we consider a typical embedded microarchitectural design space based on the XScale processor shown in table 2. We show the microarchitecture parameters of the XScale that we varied along with the values each parameter can take. To generate our design space we varied the cache and branch predictor configurations because they are important components of an embedded processor. Other significant parameters such as pipeline depth or voltage scaling were not considered, but could be easily added to our experimental setup. We varied the parameters over a wide range of values, beyond those in current systems, to fully explore the design space of this processor's microarchitecture. In total there are 288,000 different configurations some of which give increased performance over the original processor (up to 19%), others give significantly reduced power consumption (up to 21%) which is equally significant for embedded processors. In our experiments we used a sample space of 200 configurations selected with uniform random sampling. Each configuration was implemented in the Xtrem simulator [5] which has been validated for performance against the XScale processor. We also

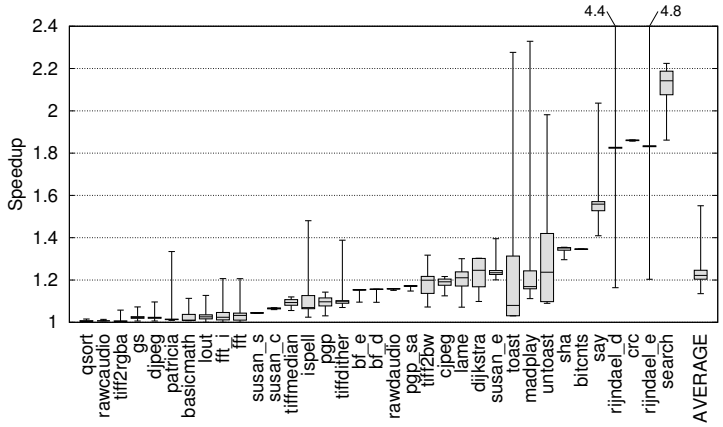


Figure 4. Distribution of the maximum speedup available across all microarchitectures on a per-program basis. The x-axis represents the program and the y-axis the speedup relative to gcc's default optimization level O3. The central line denotes the median speedup. The box represents the 25 and 75 percentile area while the outer whiskers denote the extreme points of the distribution.

used Cacti [36] to accurately model the cache access latencies, ensuring our experiments were as realistic as possible.

Application to other spaces We have considered an embedded microarchitectural space and benchmark suite because this represents a real-world challenge for portable compiler optimization, based around an existing processor configuration. However, the machine learning schemes we develop in this paper are independent of this and can equally be applied to other, more complex spaces. In section 6 we extend the microarchitectural space by varying frequency and issue width and show that our approach adapts to the new space.

4.2.1 Compiler Optimization Space

Finding the best optimization for a specific program on a specific microarchitecture is intractable. To do this, all equivalent programs, of which there are infinitely many, must be evaluated and the best selected. Therefore, in this paper we limit ourselves to finding the best optimization within a finite space of optimizations. The space we have considered consists of all combinations of the compiler passes and their parameters shown in figure 3. These passes are applied within gcc 4.2, an industry standard for the XScale processor. They were found to have a performance impact on the XScale microarchitecture configuration and other researchers have explored a similar space [39], allowing independent comparisons with existing work. Turning passes on or off leads to a design space of 642 million different optimizations. Varying the parameters controlling some of the optimizations, (*e.g. gcse* has five further options) leads to a total of $1.69 * 10^{17}$ unique optimization passes.

Clearly, it is not feasible to exhaustively enumerate this entire space to find the best optimizations for each program on each microarchitecture. However, iterative compilation can be used to quickly find an approximation of the best and has been shown to out-perform other approaches [31]. Hence, to find the best optimizations, we used iterative compilation which evaluated a 1000 different optimizations. These optimizations were selected using uniform random sampling. In our experimental setup we saw almost no additional improvement after 1000 evaluations, showing that it is a useful indicator of

the upper bound on realistic performance achievable by a compiler.

4.3 Characterizing the Compiler Space

Before trying to build a compiler that optimizes across microarchitectures, it is important to examine whether there is any performance to gain. For this purpose, we evaluated the impact of the compiler optimizations on the 35 MiBench programs compiled with the 1000 random optimization passes, each of which was executed on the 200 different architectural configurations, as described earlier. This corresponds to a sample space of 7 million simulations and should provide some evidence of the potential benefits of optimization passes selection across microarchitectures and programs. We then record the best performance achieved on a per program per architecture basis.

Figure 4 shows the sample space's distribution of maximum speedups for each program across the microarchitectural configurations when compiling with the best set of optimizations per program per microarchitecture. What is immediately clear is that there is significant variation across the programs. For some the performance improvement is modest; selecting the best optimizations does not help the library-bound benchmarks *qsort* or *basicmath* for instance. For *rijnadael_e* there is significant performance improvement to be gained, ranging from a 1.2x speedup to 4.8x in the best case, 1.8x being the average. In the case of *search* the extremes are much less but on average selecting the best optimization gives a 2.2x speedup across all configurations. In programs such as *toast*, *madplay* and *untoast*, there are modest speedups to be gained on average but significant improvements available on certain microarchitectures (up to 2.4x for *madplay* as the top whisker shows).

The right-most entry shows that there is an average speedup of 1.23x available across the design space if we were able to select the best optimizations per program per microarchitecture. The challenge is to develop a compiler that can automatically obtain this speedup without having seen the program or target microarchitectural configuration before. Furthermore, it should be able to capture the high performance available on certain microarchitectures and avoid the potential slowdowns found by picking the wrong optimizations. We found that choosing the wrong set of passes can lead to an average speedup of 0.7 across programs and 0.2 in the worst case (*i.e.* 5 times slower). The next section evaluates experimentally the performance of the machine-learning compiler developed in section 3.

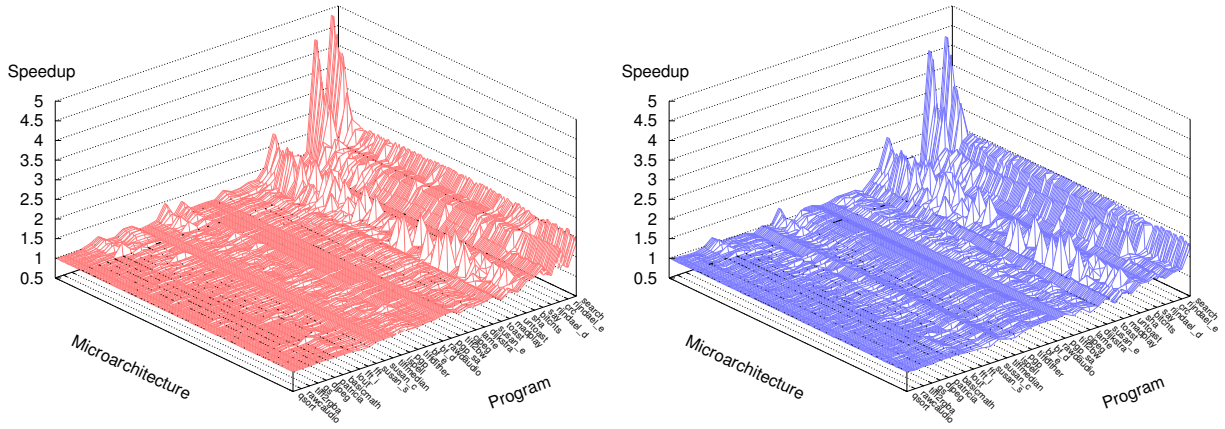
5 Experimental Methodology and Results

We first describe our evaluation methodology and then evaluate our approach across programs and microarchitectures. This is followed by an analysis of the results.

5.1 Evaluation Methodology

This section describes how we perform our experiment and determine the best performance achievable in our space.

Cross-Validation To evaluate the accuracy of our approach we use *leave-one-out cross-validation*. This means that we remove a program and a microarchitecture from our training set, build a model based on the remaining data and then predict



(a) Best passes **(b) Our compiler**

Figure 5. Speedup over O3 for each program/microarchitecture pair. Figure (a) shows the best improvement possible over the programs and microarchitectures. Figure (b) shows the performance of the passes predicted by our compiler scheme. Our portable optimizing compiler can accurately predict the best passes to enable across this space to achieve the speedups available across all programs and microarchitectures.

the best optimizations for the removed program and microarchitecture. Following this, the program is compiled and executed with the predicted optimizations on that microarchitecture and its performance recorded. We then repeat this for each program/microarchitecture. This is a standard evaluation methodology for machine learning techniques and means that we never train using the program or microarchitecture for which we will optimize. Hence, this is a fair evaluation methodology.

Best Performance Achievable In addition to comparison with O3, we want to evaluate our approach by assessing how close its performance is to the maximum achievable. Although it is intractable to determine the best performance that can be achieved by any set of optimization passes, as stated in section 4.2.1, we consider the performance of an iterative compiler with 1000 evaluations as being an appropriate upper bound for a compiler using a single profile run.

5.2 Program/Microarchitecture Optimization Space

We first consider the performance of our compiler compared to the maximum speedups available. Figure 5(a) shows the maximum speedups achievable, when selecting the best optimizations, relative to the default optimization level, O3, across the program and microarchitectural spaces. The microarchitectures are ordered so that those with large speedups available over O3 are on the left. The benchmarks are ordered so that those with large performance increases (such as *search*) are on the right (as in figure 4). In the back corner, the maximum speedup achievable with the best compiler passes is obtained by *rijndael_e*. This benchmark achieves a 4.85x speedup on a microarchitecture with a small instruction cache size. The optimizations leading to this result do not include any loop optimizations (apart from moving loop-invariant code out of the loops). In particular, no loop unrolling is performed because there is already extensive, optimized software loop unrolling programmed into the source code.

The performance of our compiler across the programs and microarchitectures is shown in figure 5(b). As is immediately apparent, it is almost identical to the performance achieved when using the best optimizations, figure 5(a). Our model is highly accurate at predicting very good compiler passes across the programs and microarchitecture space. For all program/microarchitecture pairs with large performance available, our approach is able to achieve significant speedups, as is shown by the peaks for programs *ispell*, *madplay*, *rijndael_d* and *rijndael_e*. These graphs clearly demonstrate that our model is able to capture the variation in speedups available across the program and microarchitecture spaces. The next two sections conduct further evaluations of our model.

5.3 Evaluation Across Programs

This section focuses on the performance of our compiler on each program rather than examining the microarchitectural space. Figure 6 shows the performance of each program when optimized with our portable optimizing compiler, relative to compiling with O3, averaged across all microarchitecture configurations. The second bar, labeled *Best*, is the maximum speedup achievable for each program. On average, our technique obtains a 1.16x performance improvement across all programs and microarchitectures with just one profile run, achieving a 1.94x speedup for *search* on average.

For three benchmarks in particular (*search*, *rijndael_e* and *rijndael_d*), our scheme achieves significant speedups, approaching the best performance available. Figure 6 shows that our model is able to correctly identify good optimizations, allowing these programs to exploit the large performance gains when available.

However, figure 6 also shows that some programs experience minor slowdowns compared with O3. Considering *rawcaudio* for example, our approach achieves only a 0.97x speedup. This can be explained if we refer back to figure 4 where we can see that there is negligible performance improvement to be gained over O3 for this program, even when picking the best optimizations per microarchitecture. Unfortunately, for this benchmark, the majority of optimizations are detrimental to performance and being less than 100% accurate in picking optimizations means that compiling with our scheme causes a small amount of performance loss.

Considering our technique compared to the maximum speedup achievable, we approach *Best* in most cases. For some programs, such as *susan_e*, we obtain over 95% of the maximum performance. However, for *crc* we achieve only 30%. The reason for this shortfall is due to a subtlety in the source code of *crc*. The main loop within this benchmark updates a pointer on every iteration, resulting in a large number of loads and stores. By performing function inlining and allowing a large growth factor (parameter *max-inline-insns-auto*), this pointer increment is reduced to a simple register addition which in turn reduces the number of data cache accesses. The performance counters are not sufficiently informative to enable our machine learning model to capture this behavior. This prevents our model from selecting the best passes. However, the addition of extra features, in particular code features [9], would enable us to pick this up and will be considered in future work.

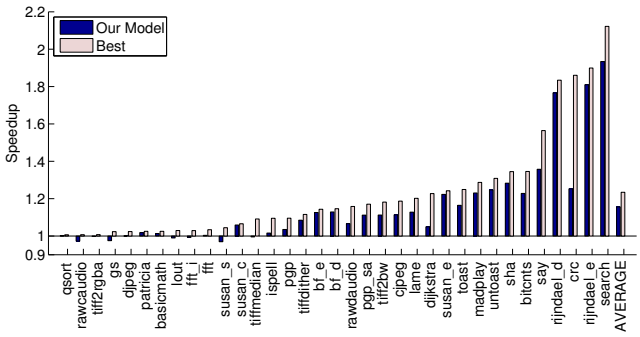


Figure 6. The performance of our approach and the best optimizations achieved by iterative compilation for each program, normalized to O3 and averaged over all microarchitectures.

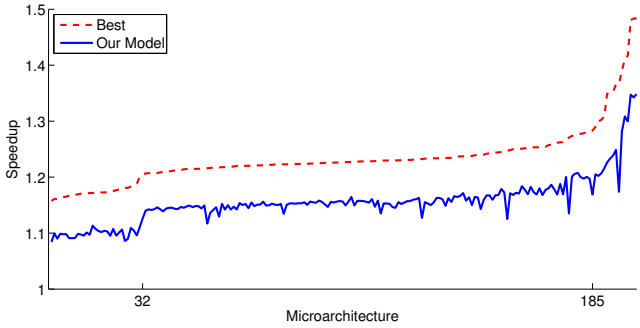


Figure 7. The performance of our approach and the best optimizations achieved by iterative compilation for each microarchitecture, normalized to O3 and averaged across programs.

5.4 Evaluation Across Microarchitectures

We now turn our attention to the performance of our compiler across the microarchitecture space rather than across programs. Figure 7 shows the performance of our compiler compared the best performance available for each microarchitecture, labeled *Best*. The microarchitectural configurations are ordered in terms of increasing speedup available over O3 (*i.e.* the *Best* line). Those on the left have little speedup available whereas those on the right can gain significantly.

For our portable optimizing compiler we see that the amount of improvement over O3 varies from 1.08x to 1.35x. This gives an average speedup of 1.16x across all programs and microarchitectures. It is important to see that our scheme closely follows the trend of the *Best* optimizations, showing how our approach captures the variation between configurations, exploiting architectural features when performance improvements can be achieved.

Looking at figure 7 in more detail we can see that it is divided into roughly three regions. On the left, up to configuration 32, the first region has little performance improvement available. All microarchitectures in this area have a small data cache. Unfortunately gcc has very few data access optimizations, meaning the available speedups are relatively small. Following this is the second region where the *Best* optimizations gain an average 1.2x speedup and our scheme manages to capture a respectable 1.16x.

Finally in the third section, after configuration 185, the available performance improvement increases dramatically. These microarchitectures on the right have a small instruction cache, meaning that it is important to prevent code duplication wherever possible. This is typical of embedded systems where code size is frequently an important optimization goal. The performance counter specifying the instruction cache miss rate enables our model to learn this from the training programs. In particular, our compiler learns that instruction scheduling (*schedule-insns*) and function inlining (*inline-functions*) must be disabled to prevent code size increases. In the case of instruction scheduling, this increase is due to a subsequent register allocation pass which emits more spill code for certain schedules. Here we can see an effect of the complex relationships

between passes within the compiler. Nonetheless, our model is able to cope with these interactions and achieve the majority of the speedups available in this area.

Summary We have shown that our portable optimizing compiler achieves an average 1.16x speedup over O3 across the entire microarchitecture space for the MiBench benchmark suite. This is equivalent to 67% of the speedup achieved by the *Best* optimization passes and is roughly consistent across the architecture configuration space. In addition, our approach is able to achieve higher levels of performance whenever they are available, accurately following the trends in the optimization space across programs and microarchitectures. The next sections analyzes our results, describing the passes that are important in our space and how our model selects good optimization passes for new programs and microarchitectures.

5.5 Program Impact on Optimization Passes

Section 5.2 showed that our compiler's performance closely follows the speedups achieved by the best optimizations for each program/microarchitecture pair. We now consider how it achieves this by focusing on those optimizations that are most likely to affect performance. Note that this is a post-hoc analysis and, in general, we cannot know in advance whether an optimization will be likely to affect performance for a specific program and microarchitecture.

Figure 8 shows a Hinton diagram of the normalized mutual information between each optimization and the speedups obtained on each program. Intuitively, mutual information gives an indication of the impact (good or bad) of a specific compiler pass on each program. The larger the box, the greater the impact of the pass. However, as this is a summary across all architectures an optimization may be important for just a few microarchitectures but not for the others, leading to a small box being drawn.

It is clear from figure 8 that some optimizations are important across all programs, whereas others are only important to a few benchmarks. For example, instruction scheduling (*schedule-insns*) is important for almost all benchmarks. As discussed in section 5.4, in some cases this optimization has a negative impact on microarchitectures with a small instruction cache. Loop unrolling (*unroll-loops*) is also an important optimization for many programs. For programs such as *search*, which contains loops with a known number of iterations, it is important to consider this optimization to achieve good performance. However, for others, such as *rijndael_e*, this optimization does not play a crucial role in achieving good performance because extensive unrolling is already implemented in the source code.

The optimization passes affecting function inlining (*inline-functions* to *param-inline-call-cost*) have little impact on most programs. However, for four programs, *ispell, pgp, pgp_sa* and *say*, these are the most important passes. By using the mutual information shown in figure 8 our model focuses on those optimizations that are most likely to affect performance on a per program/architecture basis.

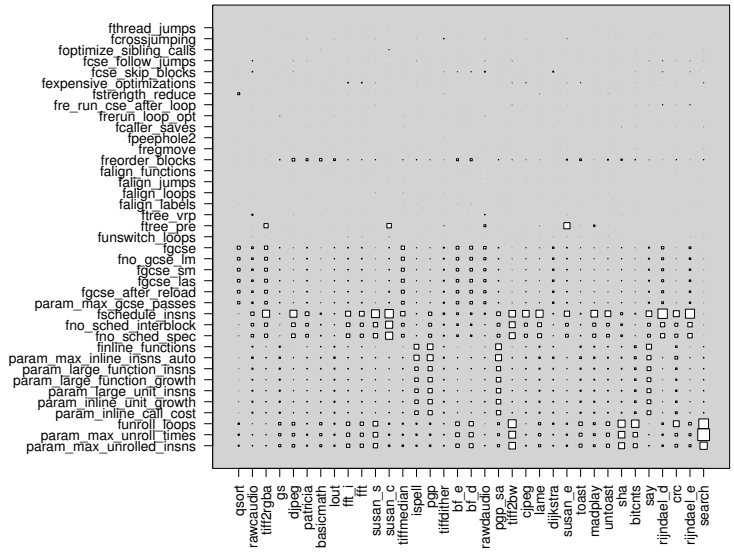


Figure 8. A Hinton diagram showing the optimizations that our model considers most likely to affect performance for each benchmark. The larger the boxnew, the more likely an optimization affects the performance of the respective program.

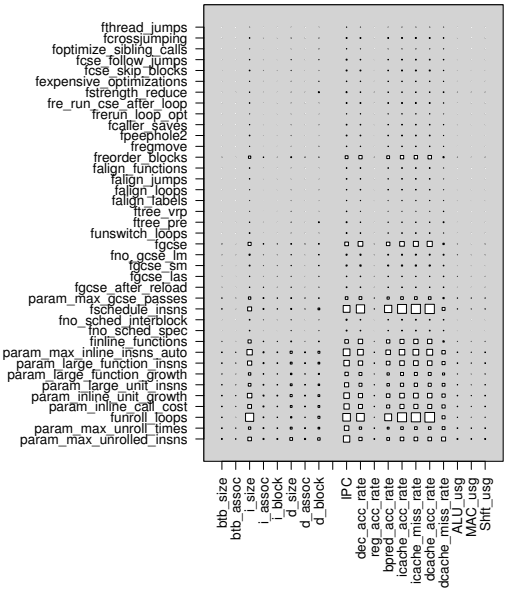


Figure 9. A Hinton diagram showing the relationship between optimizations and features. The larger the box, the more informative a feature is in predicting whether to apply the optimization.

5.6 Microarchitecture Impact on Compiler Passes

Having analyzed the optimizations that have most impact on different programs, we now turn our attention to the relationship between the microarchitecture and compiler optimizations. Figure 9 presents another Hinton diagram showing the microarchitectural impact on the best optimizations to apply. These results are averaged over all programs. The features are separated into two groups: the first contains the eight architectural parameters **d** whilst the second contains the 11 performance counters **c**.

Of all the micro architectural parameters, the size of the instruction cache (denoted *i_size*) has the biggest impact on compiler optimization. In particular, it strongly influences the optimizations that control function inlining (*inline_functions*) and loop unrolling (*unroll_loops*) It is therefore critical to predict these optimizations correctly (based on *i_size*) to avoid increasing the cache miss rate on small cache configurations. Furthermore, on larger cache configurations, it is important to perform aggressive inlining and unrolling to exploit the full potential of the cache.

Now considering the performance counters, **d**, we can see that IPC has significant impact. This is used by the model in conjunction with the other features to predict the most important optimizations to apply, such as block reordering (*reorder_blocks*), global common subexpression elimination (*gcse*), instruction scheduling (*schedule_insns*), function inlining (*inline_functions*) and loop unrolling(*unroll_loops*). The performance counters that record cache and branch predictor access/miss rates also have significant impact on choosing the best optimization flags. Surprisingly, knowledge of register and

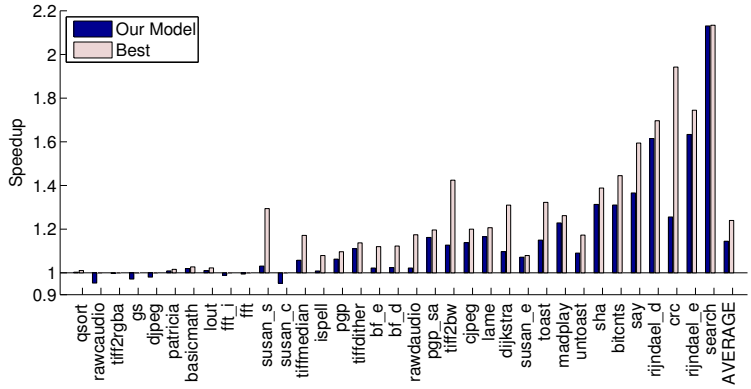


Figure 10. The performance of our approach and the best optimizations achieved by iterative compilation for each program, normalized to O3 and averaged over all microarchitectures on an extended space.

functional unit usage has little importance in determining the correct compiler optimizations to apply.

While some of these observations may seem rather intuitive, current production compilers, such as gcc, always use the same strategy when applying optimization passes, independently of the architectural parameters. One immediate recommendation would be to make gcc's unrolling and inlining optimizations sensitive to the instruction cache size and to make use of branch predictor and cache access performance counters. However, this is just a post hoc analysis based on the results of *this* space. The technique developed in this paper is micro-architectural space neutral enabling a compiler to automatically adapt to any underlying microarchitecture, as shown in the next section.

6 Extending the Microarchitectural Space

While our approach works well on a predefined architecture space, it is reasonable to ask how would it perform if the architecture space was changed at a later date. We therefore extended our space by varying two microarchitectural parameters not considered in section 4.2.1, namely frequency and processor width. Frequency ranges from 200 to 600 MHz while issue width is either 1 or 2. As a reference, the corresponding XScale values are 400 MHz and issue width 1.

Given this extended space, we then applied our approach, predicting the best optimization passes for it. Figure 10 shows the resulting performance across programs, compared to the best performance available. In this new space, selecting the correct compiler optimization passes has a similar impact as before. The *Best* optimizations give an average 1.25x improvement over O3 compared to 1.23x in the previous space. Our approach is able to achieve an improved average of 1.18x speedup. This is comparable to the performance achieved on the previous space without any modification to our approach. If we were to include new features that capture the behavior of the additional architectural parameters, the performance of our model would be further improved.

7 Related Work

There is a significant volume of prior work related to this paper which we discuss in the following seven sections.

Domain-Specific Optimizations Yotov *et al.* [41] investigated a model-driven approach for the ATLAS self-tuning linear algebra library that uses the machine description to compute the optimal parameters of the optimizations. SPIRAL [33] is another self-tuned library. It automatically generates high-performance code for digital signal processing by exploiting domain-specific knowledge to search the parameter space at compile-time. These two systems both required domain-specific knowledge and the use of iterative compilation to optimize themselves on the target system. They have to be retuned for each new platform. This contrasts with our work where the compiler is built only once and optimizes across a range of microarchitectures using just one profile run for any new program.

Iterative Compilation Iterative compilation optimizes a single program on a specific microarchitecture by searching the optimization space. Cooper *et al.* [7] were amongst the first to use a genetic algorithm to solve the phase ordering problem, achieving impressive code size reductions. Later, an extensive study of this problem was conducted, advocating the use of multiple hill-climber runs [2]. Vuduc *et al.* [40] looked at the problem of optimizing a matrix multiplication library using a statistical criterion to stop search. Kulkarni *et al.* [24] used their previously developed VISTA compiler infrastructure [43] to search for effective optimization phases at a function level. They build a tree of effective transformation sequences and use it to limit the search of the optimization space with a genetic algorithm.

Orthogonally to this, other researchers have focused on finding the best optimizations settings to apply. Triantafyllis *et al.* [37] concentrated on a small set of optimizations that perform well on a given set of code segments. These are placed in a search tree which is traversed to search for good optimization combinations for a new application. Pan and Eigenmann [31] evaluated this techniques with their own algorithm that iteratively eliminates settings with the most negative effect from the search space. Finally the Acovea tool [25] uses a genetic algorithm to find the best set of optimizations flags from gcc given a program. Compared to our approach, all these techniques specifically tune each program on a per-program, per-architecture basis by searching its optimization space. Conversely, our technique avoids search and recompilation by directly predicting the correct set of compiler optimizations to apply on a new micro-architecture.

Analytic Models for Compilation The use of analytic models has also been investigated to speedup iterative compilation. Triantafyllis *et al.* [37] used an analytic model to reduce the required time to evaluate different compiler optimizations for different code segments. Zhao *et al.* [42] developed an approach named FPO to estimate the impact of different loop transformations. To overcome the high cost of iterative compilation, Cooper *et al.* [6] developed ACME which uses the concept of virtual execution; a simple analytic model that estimates the execution time of basic blocks. Analytic models have proved to be useful for searching the optimization space quickly. However, since our model does not perform any search but directly predicts the best optimization passes to apply, they are not applicable in this context.

Machine-Learning Compilers Some of the first researchers to incorporate machine learning into optimizing compilers were McGovern and Moss [30] who used reinforcement learning for the scheduling of straight-line code. Stephenson and Amarasinghe [34] looked at tuning the unroll factor using supervised classification techniques such as K-Nearest-Neighbor and Support Vector Machines. All these approaches only consider one compiler optimization and, furthermore, are specific to the target architecture.

Subsequent researchers have considered predictive models to automatically tune a compiler for an existing microarchitecture. These models use program's features to focus the search of the optimization space in promising areas. Agakov *et al.* [1] used code features to characterize programs while Cavazos *et al.* [3] investigated the use of performance counters. However, both still require a search of the space and as such are comparable to iterative compilation. To tackle this problem, Cavazos [4] developed a logistic regressor that predicts which optimizations to apply at a method level within the Jikes RVM. Recently the Milepost-gcc has been developed to drive the compiler optimization process based on machine learning [14]. Each of these approaches, however, has to be entirely retrained for any new platform and cannot be used for "compiler in the loop" architecture design-space exploration. In a similar direction, Stephenson *et al.* [35] investigated the use of meta optimizations by tuning the compiler heuristics using genetic programming and Hoste and Eeckhout [17] used genetic algorithms to search for the best static compiler flags across various programs. In contrast to these static heuristics, we have developed a model that predicts the best optimizations to apply based on the characteristics of any new program or microarchitecture.

Retargetable Compilers Integration of compiler and microarchitecture development is not new. Frameworks such as Buildabong [13] and Trimaran [38] allow automatic exploration of both compiler and microarchitecture spaces. Other researchers have focused on creating portable compilers such as LLVM [26]. However, these infrastructures focus purely on portability from an engineering point of view: developing tools and optimizations that can be reused across many microarchitectures.

Microarchitectural Design Space Recently there has been significant interest in predicting the performance of different programs across a microarchitectural design space. Schemes include linear regressors [20], artificial neural networks [18, 19], radial basis functions [21, 39] and spline functions [27, 28]. These models obtain similar accuracy to each other [29]. Other researchers have since propose new models that learn across programs [11, 23]. However, all these models are limited to microarchitectural exploration and have not considered compiler optimizations.

Co-design Space Exploration Finally, other researchers have explored the microarchitecture and compiler optimization co-design space on a per program basis. Vaswani *et al.* [39] focused primarily on allowing exploration of this space. They built a model for a specific program that predicts the performance of compiler flags on microarchitecture configurations for that program. However their model cannot handle unseen programs and its use is therefore limited and cannot be used for portable optimization. Dubach *et al.* [10] and Desmet *et al.* [8] independently also explored the microarchitectural and

compiler optimization co-design space. In addition, Dubach *et al.* [10] developed models that predict the performance that the best set of compiler flags could achieve for a given program on any microarchitecture, without actually searching the optimization space. However, these models are program-specific and predict program performance, rather than the actual optimizations to apply. In contrast, our technique directly predicts the optimization passes to apply for any unseen program on any unseen microarchitecture.

8 Conclusions and Future Work

This paper has presented a portable optimizing compiler that automatically learns the best optimization passes to apply for any new program on any new microarchitecture. Using a machine learning approach, we can achieve on average a 1.16x speedup over the default best optimization pass after just one profile run. This corresponds to 67% of the maximum speedup available if we were to use iterative compilation with 1000 evaluations. We achieve this after a one-off training cost which is amortized across all generations of the processor. We also show that similar performance is achieved when applied to a new extended micro-architectural space. Future work will consider fine-grained optimizations at a function level and the ability of the compiler to alter its optimization pass orderings. We will remove the single profile run we currently require by considering abstract syntax tree features to characterize programs. Furthermore, we will look at reducing the training cost of our approach by using clustering techniques which can dramatically reduce the amount of training data needed.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO*, 2006.

[2] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. *SIGPLAN Not.*, 39(7), 2004.

[3] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO*, 2007.

[4] J. Cavazos and M. F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *OOPSLA*, 2006.

[5] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G. Lueh. XTREM: a power simulator for the Intel XScale core. In *LCTES*, 2004.

[6] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: adaptive compilation made efficient. *SIGPLAN Not.*, 40(7), 2005.

[7] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *LCTES*, 1999.

[8] V. Desmet, S. Girbal, and O. Temam. Archexplorer.org: Joint compiler/hardware exploration for fair comparison of architectures. In *INTERACT workshop at HPCA*, 2009.

[9] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *CF*, 2007.

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

[11] C. Dubach, T. M. Jones, and M. F. P. O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *MICRO*, 2007.

[12] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate cpi components. In *ASPLOS*, 2006.

[13] D. Fischer, J. Teich, R. Weper, U. Kastens, and M. Thies. Design space characterization for architecture/compiler co-exploration. In *CASES*, 2001.

[14] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O. Boyle. MILEPOST GCC: machine learning based research compiler. In *GCC Summit*, 2008.

[15] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC*, 2001.

[16] M. Haneda, P. Knijnenburg, and H. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. *PACT*, 2005.

[17] K. Hoste and L. Eeckhout. Cole: compiler optimization level exploration. In *CGO*, 2008.

[18] E. İpek, B. R. de Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Euro-Par*, 2005.

[19] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS-XII*, 2006.

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

[21] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. A predictve performance model for superscalar processors. In *MICRO-39*, 2006.

[22] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, 2004.

[23] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra. Using predictive modeling for cross-program design space exploration in multicore systems. In *PACT*, 2007.

[24] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *PLDI*, 2004.

[25] S. R. Ladd. Acovea. http://www.coyotegulch.com/products/acovea/.

[26] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

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

[28] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS-XII*, 2006.

[29] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *PPoPP-12*, 2007.

[30] A. McGovern and J. E. B. Moss. Scheduling straight-line code using reinforcement learning and rollouts. In *NIPS*, 1998.

[31] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO*, 2006.

[32] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with spec cpu benchmark suites. In *ISPASS*, 2005.

[33] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, Feb. 2005.

[34] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, 2005.

[35] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *PLDI*, 2003.

[36] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. Cacti 4.0. Technical Report HPL-2006-86, HP Laboratories Palo Alto, 2006.

[37] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *CGO*, 2003.

[38] Trimaran: An infrastructure for research in instruction-level parallelism. http://www.trimaran.org/, 2000.

[39] K. Vaswani, M. J. Thazhuthaveetil, Y. N. Srikant, and P. J. Joseph. Microarchitecture sensitive empirical models for compiler optimizations. In *CGO*, 2007.

[40] R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical models for empirical search-based performance tuning. *Int. J. High Perform. Comput. Appl.*, 18(1):65–94, 2004.

[41] 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. *SIGPLAN Not.*, 38(5):63–76, 2003.

[42] M. Zhao, B. Childers, and M. L. Soffa. Predicting the impact of optimizations for embedded systems. *SIGPLAN Not.*, 38(7), 2003.

[43] W. Zhao, B. Cai, D. Whalley, M. W. Bailey, R. van Engelen, X. Yuan, J. D. Hiser, J. W. Davidson, K. Gallivan, and D. L. Jones. Vista: a system for interactive code improvement. In *LCTES/SCOPES*, 2002.