Exploring and Predicting the Architecture/Optimising Compiler Co-Design Space

Christophe Dubach, Timothy M. Jones and Michael F.P. O'Boyle
Members of HiPEAC
School of Informatics
University of Edinburgh
christophe.dubach@ed.ac.uk,{tjones1,mob}@inf.ed.ac.uk

ABSTRACT

Embedded processor performance is dependent on both the underlying architecture and the compiler optimisations applied. However, designing both simultaneously is extremely difficult to achieve due to the time constraints designers must work under. Therefore, current methodology involves designing compiler and architecture in isolation, leading to sub-optimal performance of the final product.

This paper develops a novel approach to this *co-design* space problem. For any microarchitectural configuration we automatically predict the performance that an optimising compiler would achieve without actually building it. Once trained, a single run of -O1 on the new architecture is enough to make a prediction with just a 1.6% error rate. This allows the designer to accurately choose an architectural configuration with knowledge of how an optimising compiler will perform on it. We use this to find the best optimising compiler/architectural configuration in our co-design space and demonstrate that it achieves an average 13% performance improvement and energy savings of 23% compared to the baseline, leading to an energy-delay (ED) value of 0.67.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Retargetable compilers*; C.4 [**Computer Systems Organization**]: Performance of systems—*Design studies, modeling techniques*; C.0 [**Computer Systems Organization**]: General—*Hardware/software interfaces*

General Terms

Design, Experimentation, Performance

Keywords

Architecture/compiler co-design, design-space exploration, performance prediction

CASES'08, October 19–24, 2008, Atlanta, Georgia, USA.

1. INTRODUCTION

Embedded system performance is usually achieved via efficient processor design and optimising compiler technology. Fast time-to-market is critical for the success of any new product and therefore it is crucial to design new microprocessors quickly, without sacrificing performance. However, during early design stages, architectural decisions must be taken with only limited knowledge of other system components, especially the compiler. Ideally we would like to consider both architecture and optimising compiler design simultaneously, selecting the best combination.

Unfortunately exploring this combined design or *co-design* space is extremely time consuming. For each architecture to consider we would have to build an optimising compiler, which is clearly impractical. Instead, typical design methodology consists of first selecting an architecture under the assumption that the optimising compiler can deliver a certain level of performance. Then, a compiler is built and tuned for that architecture which will hopefully deliver the performance levels assumed.

Clearly this is a sub-optimal way of designing systems. The compiler team may not be able to deliver a compiler that achieves the architect's expectations. More fundamentally, if we could predict the performance of the eventual optimising compiler on any architecture, then an entirely different architecture may have been chosen. This inability to directly investigate the combined architecture/optimising compiler interactions means we end up designing tomorrow's architectures based on yesterday's compiler technology.

In this paper we propose a novel approach to this co-design space problem. We build a machine-learning model that can automatically *predict* the performance of an optimising compiler across an arbitrary architecture space *without* tuning the compiler first. This allows the designer to accurately determine the performance of any architecture as if an optimising compiler were available. Given a small sample (less than 0.01%) of the architecture and optimisation space, our model can predict the performance of a yet-to-be-tuned optimising compiler using information gained from a non-optimising baseline compiler. This achieves an error rate of 1.6% across all microarchitectures in the co-design space.

The use of predictors, particularly to reduce simulation time, is not new. Several authors have shown that it is possible to predict the performance of a fixed program on an architecture space when compiling with fixed optimisations [1, 2, 3]. Other researchers have shown that it is possible to predict the impact of compiler optimisations on a fixed architecture [4, 5, 6]. In [7] this is taken one step further where a

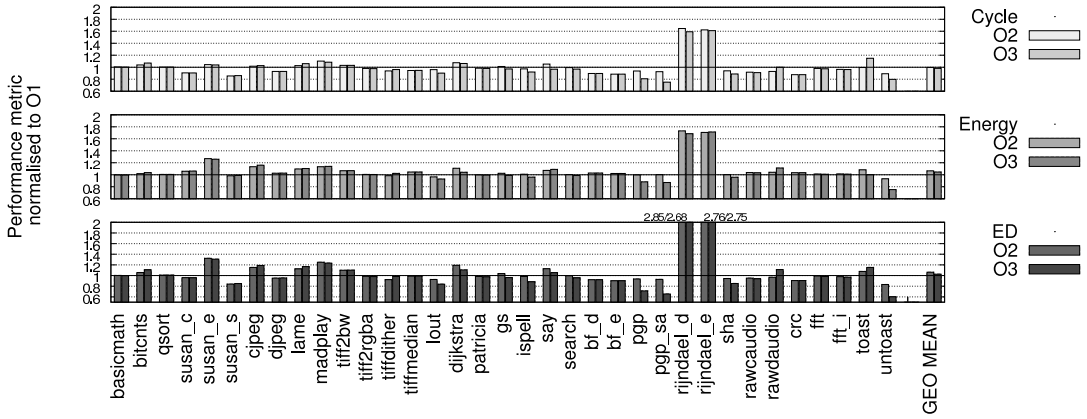


Figure 1: Execution time, energy and ED of each benchmark from MiBench when compiled with -O2 and -O3, normalised by -O1 (so lower is better).

Parameter	Low → High	Baseline
ICache size	4K → 128K	32K
ICache associativity	4 → 64	32
ICache block size	8 → 64	32
DCache size	4K → 128K	32K
DCache associativity	4 → 64	32
DCache block size	8 → 64	32
BTB size	128 entries → 2048 entries	512 entries
BTB associativity	1 → 8	1

Table 1: Microarchitectural parameters and the range of values they can take. Each parameter varies as a power of 2, with 288,000 total configurations. Also shown are the baseline values.

model predicts the performance of compiler settings on different architectures for a fixed program. However, as we will show in section 5, this approach fails to accurately predict the performance of an optimising compiler. To the best of our knowledge, we propose the first model to predict the performance of an optimising compiler across the architecture space before the compiler is tuned.

In this work we separately explore the microarchitectural and compiler optimisation spaces. We then show that there is the potential for significant improvement over the baseline processor and compiler by exploring the combined co-design space. We also demonstrate that the optimal compiler for one architecture is not the best for all. We build our model that predicts the performance of an optimising compiler on any microarchitectural configuration. We then use it to find the best architectural/optimising compiler configuration and demonstrate that for this architecture, an optimising compiler can deliver the predicted performance. The best design achieves significant performance increases resulting in a 13% improvement in execution time, 23% savings in energy and an energy-delay product (ED) of 0.67.

The rest of this paper is structured as follows. Section 2 describes our experimental methodology. Section 3 characterises the microarchitectural and compiler optimisation spaces in isolation whilst section 4 explores the combined design space. We build a machine-learning model in section 5 and evaluate it against a state-of-the-art alternative approach in section 6. Here we also show how our model is used to select the best architecture/optimising compiler combination and that this configuration does achieve the predicted level of performance. Finally, section 7 describes related work and section 8 concludes.

2. METHODOLOGY

In this section we describe the baseline architecture and compiler infrastructure used as a reference point for the later sections on design space exploration. We also briefly describe the benchmarks used.

Optimising Compiler

This work considers the performance of an optimising compiler across a large microarchitectural design space. Without actually building an optimising compiler for each microarchitectural configuration, it is difficult to verify the performance that it will achieve. However, previous research [5, 8, 9, 10, 11, 12] has shown that using iterative compilation over randomly-selected flag combinations can out-perform an optimising compiler tuned for a specific configuration. This can be considered an upper bound on the performance an optimising compiler can achieve.

Hence, in this paper, we define an optimising compiler as a compiler that uses iterative compilation over 1000 randomly-selected flag combinations on the specific architecture to be tuned. This means that the optimising compiler uses the flags that lead to the best performance after running 1000 flag combinations on the architecture it is compiling for.

2.1 Architecture / Compiler Co-Design Space

To evaluate the effectiveness of co-design space exploration, we chose to use the Intel XScale processor [13] as our baseline architecture. This processor is typically found in embedded systems and its configuration is shown in table 1, column 3. In section 3.2 we show that in fact this is a well balanced design for energy and execution time.

Our benchmarks are compiled with gcc version 4.1.0. This is a well designed compiler that is widely used within industry. In our experiments, all compiler optimisations are enabled from the command line by using the flags available. The co-design space is the combined space of all architectural configurations and compiler optimisations. We describe these in more detail in sections 3.1 and 3.3.

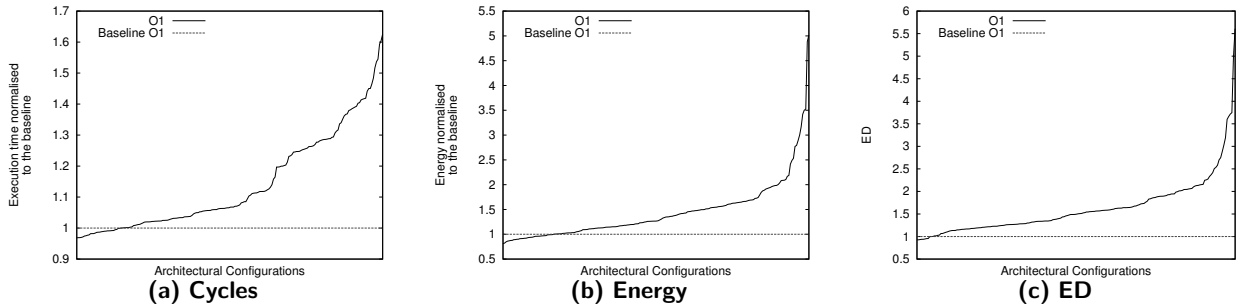


Figure 2: The average execution time, energy and ED of each microarchitectural configuration across the whole benchmark suite compiled with -O1. Each graph is independently ordered from lowest to highest and is normalised by the baseline configuration.

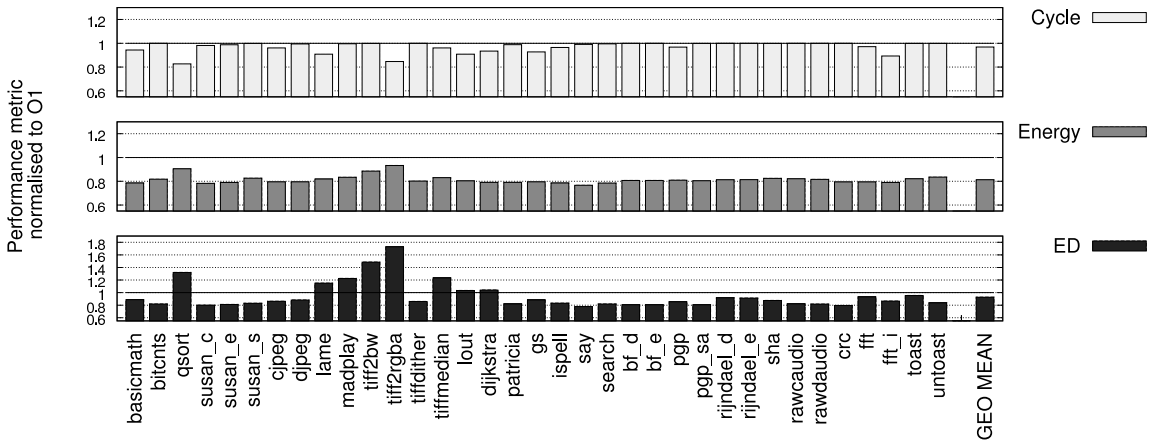


Figure 3: Execution time, energy and ED for each benchmark compiled with -O1 on the microarchitectural configuration performing best for each metric.

2.2 Experimental Framework

We used the Xtrem simulator [14] which has been validated for cycles and energy consumption against the Intel XScale processor. Using Cacti [15] we accurately modelled the access latencies of each cache configuration to ensure our experiments were as realistic as possible.

We used the full MiBench [16] suite to evaluate the performance of our system. All 35 programs were run to completion. For each benchmark we chose the inputs leading to at least 100 million executed instructions where possible. Programs *susan_c*, *susan_e*, *djpeg*, *tiff2rgba* and *search* have been run with the large input set whilst all others have been run with the small inputs.

To perform our experiments we chose 200 microarchitectural configurations and 1000 compiler optimisations from our total design space using uniform random sampling. As training data for our predictor, described in section 5, we used 15,185 design points per benchmark, out of a total of 200,000. In total, for 35 benchmarks, we ran 7 million simulations to create this sample space.

We explored the microarchitectural, compiler and co-design spaces using execution time (cycles), energy and the energy-delay (ED) product. This is an important metric which presents the trade-offs between performance and energy consumption in a single value, the lower the better.

2.3 Baseline Optimisation Level

We wanted to ensure that our baseline compiler optimisation level was realistic and effective. We considered three default optimisation levels available in gcc: -O1, -O2 and -O3. Figure 1 shows the performance, energy consumption and ED per benchmark for each of these optimisation levels, normalised to -O1. As can be seen, the optimisation levels -O2 and -O3 affect each benchmark in varying degrees. However, surprisingly, on average they both produce the same execution time as -O1. There is similar variation for energy, although on average there is higher consumption when using -O2 or -O3. When we look at the tradeoff between performance and energy, ED, it is clear that -O1 represents the best choice. Hence, we chose -O1 as our baseline optimisation. Note that the accuracy of our predictor is not affected in any way by this choice.

3. MICROARCHITECTURE AND COMPILER DESIGN IN ISOLATION

Current microprocessor design methodology involves choosing and optimising an architecture whilst developing the compiler independently. In this section we show how these two stages are usually performed.

Flag
-fthread-jumps / ⊘
-fcrossjumping / ⊘
-foptimize-sibling-calls / ⊘
-fcse-follow-jumps / ⊘
-fcse-skip-blocks / ⊘
-fexpensive-optimisations / ⊘
-fstrength-reduce / ⊘
-frerun-cse-after-loop / ⊘
-frerun-loop-opt / ⊘
-fcaller-saves / ⊘
-fpeephole2 / ⊘
-fregmove / ⊘
-freorder-blocks / ⊘
-falign-functions / ⊘

Flag
-falign-jumps / ⊘
-falign-loops / ⊘
-falign-labels / ⊘
-ftree-vrp / ⊘
-ftree-pre / ⊘
-funswitch-loops / ⊘
-fschedule-insns /
/ -fschedule-insns2 / ⊘
-fno-sched-interblock / ⊘
-fno-sched-spec / ⊘
-fgcse / ⊘
-fno-gcse-lm / ⊘
-fgcse-sm / ⊘

Flag	Values
-fgcse-las / ⊘	
-fgcse-after-reload / ⊘	
–param max-gcse-passe =	1, 2, 3, 4
-finline-functions / ⊘	
–param max-inline-insns-auto =	10,30,50,...,190
–param large-function-insns =	1300,1500,1700,...,3300
–param large-function-growth =	20,50,100,200,300,400,500
–param large-unit-insns =	4000,6000,8000,...,20000
–param inline-unit-growth =	10,20,30,...,100,200,300
–param inline-call-cost =	10,12,14,...,30
-funroll-loops / -funroll-all-loops / ⊘	
–param max-unroll-times =	2,4,6,...,20
–param max-unrolled-insns =	50,75,100,...,400

Table 2: Compiler optimisations and the values they can take. There are 642 million combinations. The baseline is -O1 with no further optimisations enabled.

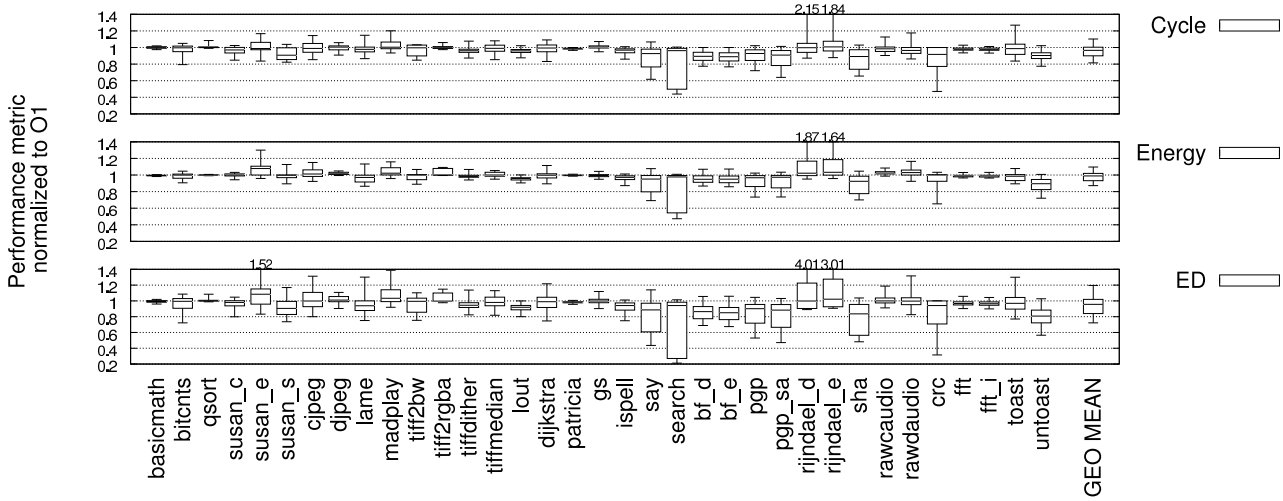


Figure 4: The compiler optimisation design space on a per-benchmark basis for execution time, energy and ED. We show the minimum, maximum, median, 25% and 75% quantiles.

3.1 Microarchitecture Design Space

We have picked a typical microarchitectural design space, whose parameters are shown in table 1. Also shown is the range of values each parameter can take and our baseline microarchitecture which is based on the configuration of the XScale processor [13]. We have chosen to vary the cache and branch predictor configurations in this work because they are critical components in an embedded processor. The total design space consists of 288,000 different configurations. In our experiments we have used a sample space of 200 randomly selected configurations. To evaluate the architecture space independently from the compiler space, we have compiled each benchmark using the baseline optimisation (-O1).

3.2 Microarchitecture Exploration

Figure 2 shows the microarchitectural design space. Each graph shows the performance achieved by each microarchitectural configuration in terms of execution time, energy and ED across the MiBench suite, normalised to the baseline architecture. The baseline performance is shown with a horizontal line. Each graph is independently ordered from lowest to highest. These graphs show that the baseline is actually a very good choice. For both execution time and energy consumption it is within the top 15% of all configurations, for ED it is within the top 5%. However, there is room for improvement. Selecting a better architecture leads to an ED value of 0.93 compared with the baseline.

Figure 3 shows the best execution time, energy and ED value for each benchmark, normalised to the baseline architecture. We picked the microarchitectural configurations leading to the best performance for each metric over the whole MiBench suite and ran each benchmark compiled with the baseline optimisation -O1 on them. In terms of execution time, three benchmarks achieve a 10% performance gain on this configuration, but the majority perform similarly to the baseline. Considering energy, the majority of benchmarks achieve 20% savings over the baseline configuration. However, the ED value for some benchmarks is over 1 because this configuration actually loses performance for these benchmarks. We cannot specialise the architecture for each program, so this configuration that is the best for ED overall, is not necessarily the best for each program.

On average, we see that we can only achieve a modest ED value of 0.93 over the baseline architecture. This is actually not a surprise, since the baseline architecture corresponds to the XScale processor and has already been highly tuned.

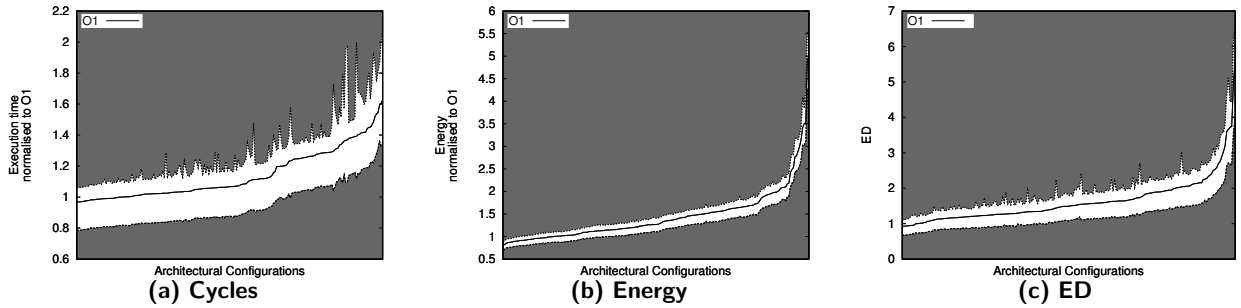


Figure 5: The co-design space of execution time, energy and ED for each microarchitectural configuration across the whole benchmark suite considering the best and worst optimisations for each program. The region in white is the co-design space, with the line showing the performance of -O1 on this architecture. Each graph is independently ordered from lowest to highest and is normalised by -O1 on the baseline configuration.

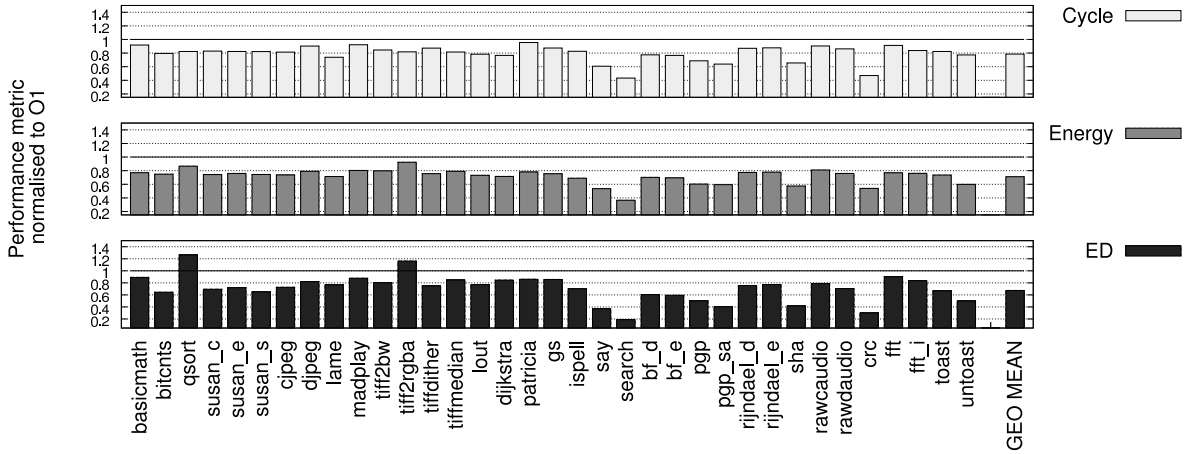


Figure 6: Execution time, energy and ED for each benchmark on the microarchitectural/optimisation configuration performing the best for each metric overall.

3.3 Compiler Optimisation Space

Having considered the microarchitecture design space in isolation, we now wish to explore the compiler space alone to show its characteristics when optimising for the baseline architecture. The optimisation space we have considered is shown in table 2. It is similar to the optimisations considered by other researchers [7], allowing meaningful comparisons with existing work. There are 642 million different combinations of optimisations when considering turning the flags either on or off. We also considered changing the behaviour of the heuristics that control some of the optimisations, leading to a total of $1.69 * 10^{17}$ unique optimisations.

Since exhaustive enumeration of this optimisation space is not feasible, we explored it by choosing 1000 different optimisations using uniform random sampling. We then ran the benchmarks compiled with these flags on the baseline architecture. As stated previously, we define an optimising compiler as an iterative compiler that uses the best of these 1000 randomly-selected flag settings

3.4 Compiler Optimisation Exploration

Figure 4 shows the execution time, energy and ED design spaces on a per-benchmark basis. We show each benchmark individually because the best combination of optimisation flags varies between programs. In these graphs we show the minimum, maximum, median, 25% and 75% quantiles. Also shown in the final column is the geometric mean from using these different optimisations across all benchmarks.

What is immediately clear is that for some benchmarks there is significant improvement to be obtained in execution time over the baseline optimisation (*e.g. search* at 0.44 and *crc* at 0.47). This also shows that picking the wrong optimisations can significantly degrade performance or increase energy consumption. On this baseline architecture, the compiler can do little to improve the performance or save energy on some programs (*e.g. basicmath* and *patricia*). In terms of ED, there is significant scope for improvement for some benchmarks, but on others the compiler optimisations have little impact. For example, *sha* achieves an ED value of 0.48 in the best case but *qsort* does not gain from optimisation. The best case flags chosen on a per-benchmark basis give the performance of the optimising compiler. On average, this can reduce execution time by 19%, save 13% of the energy consumption or achieve an ED value of 0.72. This is compared to the best ED value of 0.93 when varying the microarchitectural space alone. Not surprisingly, there is more

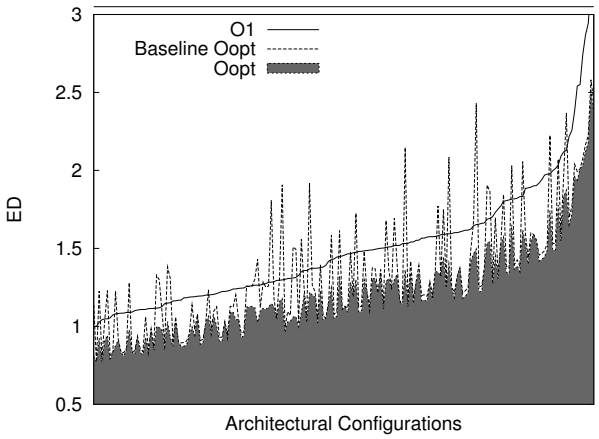


Figure 7: Optimising toast on the baseline architecture and running it on all other microarchitectures.

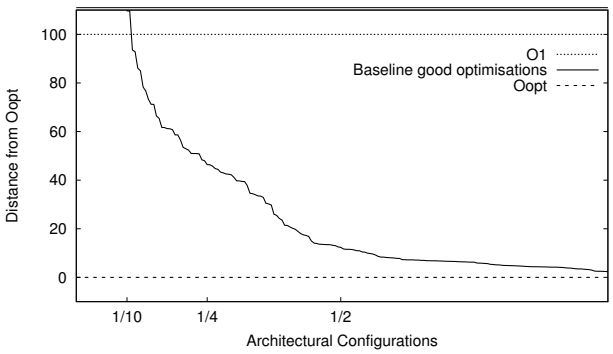


Figure 8: Optimising on the baseline architecture and running on all other microarchitectural configurations. Optimisations that are good on the baseline microarchitecture can perform worse than -O1 on other configurations.

room for improvement in the compiler space because the architecture has already been significantly tuned. However, up to this point we have considered each space independently. The next section combines the microarchitectural and optimisation spaces to consider co-design.

4. CO-DESIGN SPACE EXPLORATION

This section demonstrates that by exploring the co-design space we can find an architectural/optimising compiler configuration that achieves even higher performance than can be achieved by considering the architecture and compiler in isolation. Furthermore we show that tuning the compiler separately to the microarchitecture can lead to sub-optimal performance of the final system.

4.1 Exploration

Figure 5 shows the co-design space across microarchitectural configurations for execution time, energy and ED. The performance of the baseline compiler on each configuration is shown by the solid line. The performance of the optimising compiler on each configuration is also shown. Here we have selected the best compiler optimisations on a per-program basis for each microarchitectural configuration in our sample space. This represents the lower bound on the execution time, energy consumption or ED achievable for each architecture. Hence we have shaded the region below it. Also shown is the performance when selecting the worst compiler optimisations which represents the upper bound and we have shaded the area above this.

It is immediately obvious that there is large room for improvement over the baseline compiler optimisation across the whole microarchitectural space in terms of execution time and ED. All three graphs show that picking the wrong optimisations can lead to significant degradation of each metric. Hence, it is important to know the performance of the optimising compiler on each architecture individually.

In figure 6 we show the execution time, energy and ED values on a per-benchmark basis for the microarchitectural / optimisation configurations that perform the best for each metric. In terms of execution time, 13 benchmarks achieve a 20% improvement whilst the largest energy savings of 63% are achieved by *search*. For ED, the majority of benchmarks achieve a reasonable value of 0.8 or under. It is interesting to compare these results with those achieved when performing microarchitecture space exploration alone (figure 3). We can see that performing co-design space exploration leads to more balanced results across benchmarks in terms of ED (the maximum value is now 1.3, before it was 1.7). This is because the optimising compiler is able to take advantage of the microarchitecture, whereas before all benchmarks were compiled with -O1.

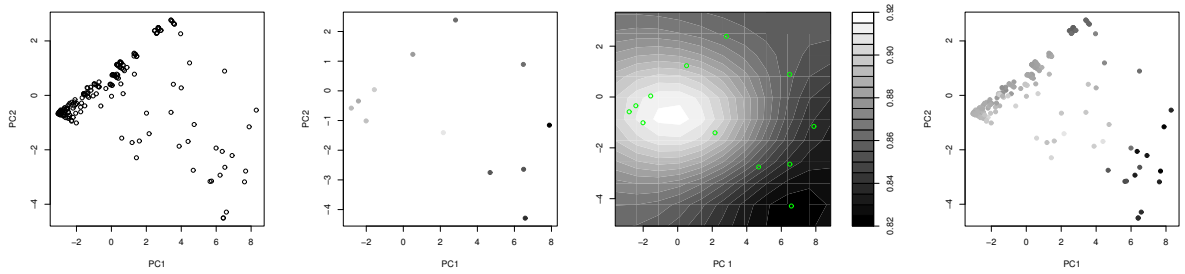
On average, considering the co-design space of compiler optimisations and microarchitectural configurations brings significant benefits. We can reduce execution time by 21%, save 29% of energy or achieve an ED value of 0.67. This is compared with an ED value of 0.93 when varying the microarchitecture alone, or 0.72 when only considering compiler optimisations. This shows the benefits of performing co-design space exploration compared with independent design of compiler and architecture.

4.2 Optimisation Sensitivity to Microarchitecture

Having shown that co-design space exploration can lead to benefits over both microarchitecture and optimisation space exploration alone, we now show that it is important to explore both spaces simultaneously. Figures 7 and 8 show the results from tuning the compiler on the baseline microarchitecture and then optimising on a new configuration. Figure 7 shows the results for the *toast* benchmark for ED. As can be seen, the optimisations that are best on the baseline architecture are actually worse than compiling with -O1 on other configurations. Critically, the best compiler optimisations vary across the architecture space.

Figure 8 shows this across all benchmarks. Here we have run 1000 optimisations on the baseline architecture and selected those that are within 5% of the best found for each benchmark. We call these the *baseline good* optimisations. Then, for each other microarchitectural configuration, we have run the benchmarks compiled with these baseline good optimisations again to determine the average ED value that they achieve. We have evaluated how far this average is from the best ED value achievable on that configuration within our sample space.

From this graph we can see that on half the architectures



(a) Features space (b) Training samples (c) Trained model (SVM) (d) Real space

Figure 9: Example use of our technique for the program fft when considering ED. First we collect performance counters, then use PCA to select two components (a). We choose training configurations and search their optimisation spaces for the best performance (b). Finally, we use SVM to determine the contour map around configurations (c). This map provides a prediction of the real performance of the optimising compiler (d).

the good optimisations for the baseline are at least 15% away from the best. For a quarter of the architectures, these good optimisations are at least 40% away from the best. Crucially, the good optimizations on the baseline architecture are actually worse than -O1 for 1/10 of the other architectural configurations. This shows that good optimisations for one architecture are not suitable for others. In essence, the optimal compiler for one architecture is not the best for all. Therefore it must be tuned on each configuration and cannot be developed independently of the microarchitecture.

4.3 Summary

This section has shown the importance of performing co-design space exploration. When the optimisation space is explored at the same time as the microarchitecture space, significant improvements can be gained over the baseline. However, designing the architecture without considering the optimisation space can result in sub-optimal performance on the final system. Considering both spaces together, we can achieve an ED value of 0.67, compared with 0.93 when designing the microarchitecture alone, or 0.72 when exploring only the compiler optimisation space.

5. PREDICTING THE PERFORMANCE OF AN OPTIMISING COMPILER

In previous sections we have presented the characteristics of our design spaces and shown that the optimal compiler for one architecture is not the best for all. To do this we have explored a sample of the total design space, considering 200 microarchitectural configurations and 1000 compiler optimisations over 35 benchmarks. In practise, however, it is not desirable to conduct such a costly co-design space exploration. We now address this issue by building a machine-learning model to predict the performance of the optimising compiler on any microarchitectural configuration.

5.1 Overview

Our model is built in three steps, with an example on the benchmark *fft* shown in figure 9. A new model is created for each benchmark we wish to predict for. First we run the program compiled with -O1 on a number of randomly-selected microarchitectures (200 in our case). We gather performance counters which allow us to characterise its behaviour (figure 9(a)). From this we can select a number of architectural configurations for training (figure 9(b)). We then explore the optimisation space of these configurations by running the program using different compiler settings in order to estimate the best performance achievable. Finally the model is trained with the results of this exploration (figure 9(c)) and predictions can be made for the entire space. These predictions can be directly compared with the real space (figure 9(d)). The next sections describe these three steps in more detail.

5.2 Characterisation of Microarchitectures with Performance Counters

To characterise each microarchitecture in the co-design space, we gather features that can be used as an input to our model. Our model can use these features to determine the performance improvement an optimising compiler can achieve over a standard baseline compiler for any microarchitecture. The features we use are performance counters extracted from a single run of the program with the default optimisation level (-O1) on each architecture.

We have chosen 9 performance counters to extract. They are the IPC; resource utilisation of the caches, branch predictor, ALUs and register file; cache miss rates and the branch misprediction rate. Performance counters like these are typically found in processor analytic models [17, 18]. We then use Principal Component Analysis (PCA) to summarise the 9 features into two values, or principal components. Figure 9(a) shows these components (PC1 & PC2) over 200 microarchitectures for the benchmark *fft*.

5.3 Gathering Training Data

Using the results from PCA, we pick microarchitectural configurations for training. To maximise the coverage of the principal components space, we divide it into a 5×4 grid, picking one training point per tile. This equates to 15 microarchitectures per program on average. Figure 9(b) shows the configurations we would pick for our *fft* example. To gather our training data we perform a search of the compiler optimisation space on each of the 15 architectures using iterative compilation with 1000 randomly-selected optimisations. This search could be made more efficient by using any specific search technique [8, 9, 10, 19]. However, this is orthogonal to the focus of this paper.

After performing our search we have an estimation of the

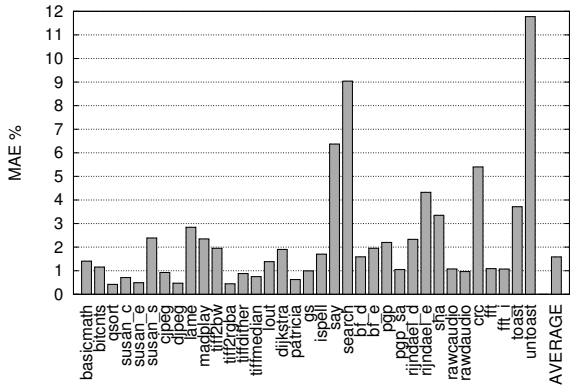


Figure 10: Prediction error for our model broken down by program when predicting the ED value achievable by the optimising compiler. The average error is only 1.6%.

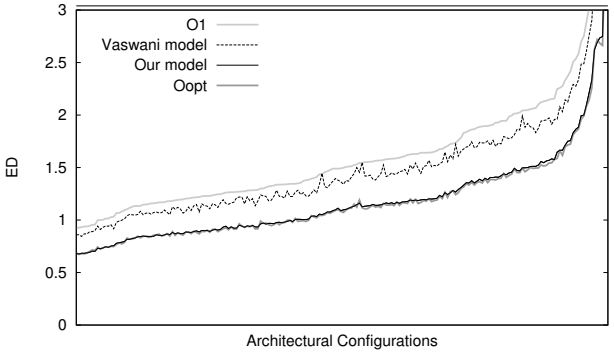


Figure 11: Predicting the performance of the optimising compiler across the microarchitectural space for the whole of MiBench. Also shown are the predictions made by Vaswani's model. Note that our model is highly accurate and overlaps significantly with -Oopt.

maximum performance achievable on each of the 15 training architectures. This is shown in figure 9(b) where darker points lead to better performance.

5.4 Training our Model

Having collected our training data, we can now train our model which is based on Support Vector Machines (SVM), adapted for the regression problem [20]. This model can distinguish between data points that behave differently. In our case, the model learns the difference between microarchitectural configurations based on the performance an optimising compiler can achieve on them. In our example following *fft*, the results from training can be seen visually in figure 9(c). Here we have circled the training configurations. The model learns the areas of similar colour based on the best performance seen on the 15 microarchitectures we selected in the previous step. Architectural configurations that lie in the same colour region are predicted to have similar optimising compiler behaviour. In other words, the model predicts that the optimising compiler has little effect in the light areas and can achieve high performance gains in the dark areas. For *fft*, this can be compared to the real space of 200 microarchitectures in figure 9(d).

Having trained our model, we can predict the performance of the optimising compiler on any new microarchitectural configuration. To do this we run -O1 on the architecture and gather performance counters. The model uses PCA to reduce the number of features to 2 and then makes a prediction based on the colour of the region where it lies.

5.5 Summary

This section has described our model used to predict the performance of the optimising compiler on any microarchitectural configuration. We first run -O1 on 200 architectures and gather performance counters. We use PCA to reduce these and then pick 15 configurations to train our model. On each of these we perform a random search of the optimisation space and then use an SVM to model the entire co-design space. To predict for any new microarchitecture we need performance counters from one run of -O1 on it.

6. MODEL EVALUATION AND COMPARISON

Having built our machine-learning model to predict the performance of the optimising compiler, this section evaluates its accuracy. We compare with a previously-proposed scheme and show a real-world use of our model in being able to predict the best configuration in the co-design space.

6.1 Prediction Error

We use the mean absolute error to evaluate our predictor, defined as $\frac{1}{N}\sum_i^N \left|\frac{predicted\ \ value_i - real\ \ value_i}{real\ \ value_i}\right|$. We used 15 microarchitectural configurations to train our model and then validated it on 185 microarchitectures.

Figure 10 shows the error of our scheme for each benchmark. It also shows the average error across the benchmark suite. As can be seen, our model achieves an error rate of 2% or under for the majority of benchmarks. In fact, for some benchmarks (such as *tiff2rgba*), the error is just 0.5%. The average error rate is 1.6% for the whole of MiBench. This shows that we have an accurate model and can correctly predict the performance of the optimising compiler.

6.2 Comparison

We wish to compare the accuracy of our model with the only other technique (to the best of our knowledge) that has considered the joint microarchitecture and compiler space. We have built the model proposed by Vaswani *et al.* [7] using an Artificial Neural Network. Their model does not directly predict the performance of an optimising compiler but instead predicts the performance of a set of compiler flags for each microarchitecture. In addition, their model does not attempt to predict energy consumption or the ED value achievable, therefore this comparison could be considered unfair. However, a comparison serves as an indication of how our approach performs against an existing machine-learning technique. To evaluate this model we used it to predict our sample compiler space of 1000 optimisations for each architecture. We then picked the best as their predicted value. We trained their model with exactly the same data as ours.

Figure 11 shows the ED value achieved by the baseline

Parameter	Value
ICache size	32K
ICache associativity	64
ICache block size	16
DCache size	16K
DCache associativity	32
DCache block size	16
BTB size	128 entries
BTB associativity	4

Table 3: Microarchitectural parameters resulting in the best ED value.

compiler on each microarchitectural configuration averaged over the whole of MiBench (labelled O1). It also shows the ED value achieved by the optimising compiler on each configuration (Oopt). A third line shows the prediction made by the model proposed by Vaswani *et al.* (Vaswani model) and a final line shows our prediction (Our model).

It is immediately obvious that our predictions follow the curve of the optimising compiler with great accuracy. More specifically, our model accurately predicts the peaks and troughs in ED as well as the stable areas. This shows the ability of our model to predict the design points that behave significantly differently from the baseline. The *Vaswani* model, however, fails to accurately predict the performance of the optimising compiler. In particular, it predicts peaks in ED where there are none and follows the -O1 line closely. This predictor, therefore, is inappropriate for finding the performance of the optimising compiler.

6.3 Predicting the Best Architectural/ Optimising Compiler Configuration

Having built and evaluated our machine-learning model, this section considers its real-world use in allowing designers to determine the optimising compiler/architectural configuration that achieves the best ED value in our space. To do this we used our model to predict on 200 microarchitectures, chosen by uniform random sampling. Our model predicted that the optimising compiler would be able to achieve the minimum ED value in the co-design space of 0.677 on the configuration shown in table 3.

To verify the prediction accuracy, we used iterative compilation on this architecture with 1000 randomly-selected optimisation settings. We found that the best ED value achievable is 0.673. This is just 0.6% away from our prediction, showing that our model is very accurate. Had we used the *Vaswani* model, it would have predicted an ED value of 0.860 for this configuration, which is an error of 27%.

In addition to this, we wanted to verify that this prediction is actually the best ED value in the sample co-design space. To do this, we used iterative compilation with 1000 optimisation settings on each of the 200 architectures we predicted for and found that this configuration does actually achieve the best ED value in the sample space.

This best microarchitectural configuration found by our model achieves a performance increase of 13% and energy savings of 23% compared to the baseline. It produces the smallest ED value because it is well balanced. The instruction and data caches have high associativity to avoid conflicts. The data cache is half the size of the instruction cache to save energy without significant loss of performance. For the benchmarks we have studied, the majority of misses are cold misses because the programs analyse streamed data. For other benchmarks which have higher data reuse, a larger data cache might be required.

6.4 Summary

This section has evaluated our model and shown that it has an error rate of just 1.6% on average. We have compared with the only other technique to predict the joint compiler/architecture space and found that our model is more accurate at predicting the performance of the optimising compiler on any microarchitectural configuration. Furthermore, we have shown that our model can predict the best optimising compiler/architectural configuration for ED within our space with just a 0.6% error. This microarchitecture achieves a 13% performance increase and energy savings of 23%, leading to an ED value of 0.67. Using our model, designers can accurately predict the impact of the optimising compiler across the co-design space.

7. RELATED WORK

Several different types of machine-learning model have been proposed to predict the design space of a microprocessor. The first type are models predicting the performance of just a single program. Techniques include the use of linear regressors [2], artificial neural networks [1, 21], radial basis functions [22] and spline functions [23, 24]. All have similar accuracy [3]. The second type of model makes use of prior-knowledge, learning across programs. Linear regression [25], artificial neural networks [26] and program-features based predictors [27] have been proposed. However, since none of these models consider the compiler optimisation space, sub-optimal microarchitectural designs could be chosen.

Searching the compiler optimisation space has been extensively explored in the literature. Feedback-directed optimisation [8, 9, 10, 11, 12, 19, 28] uses different algorithms to search the optimisation space. Agakov *et al.* [28] built a model offline that is used to guide search. Haneda *et al.* [9] make use of statistical inference to select good optimisations. Cooper *et al.* [8, 19] explore the optimisation space using hill climbing and genetic algorithms. Other researchers have used analytical [29] or empirical [4, 5, 6, 30] models to explore the optimisation space. In fact, these techniques are orthogonal to our approach and can be used to reduce training costs.

Finally, co-design space predictors [7] use one model to predict the compiler optimisation and architecture spaces. This model takes as an input the microarchitectural configuration and the desired optimisation flags and produces a prediction. However, as we have shown in section 6.2, this model fails to capture interactions between compiler optimisations and microarchitecture and cannot be used to accurately predict the performance of an optimising compiler.

8. CONCLUSION

This paper has addressed the co-design space problem by automatically predicting the performance of an optimising compiler on any microarchitectural configuration, without needing to build the compiler first. We have explored the microarchitectural, compiler and co-design spaces, showing that the optimal compiler for one architecture is not the best for all. We then built a machine-learning model to

predict the performance of an optimising compiler on any architecture. Our model achieves an error rate of just 1.6%. We used this predictor to find the best optimizing compiler/architectural configuration for ED in our design space and found it achieves a 13% performance increase and 23% energy savings, leading to an ED value of 0.67.

Acknowledgements

This work has been supported by Milepost [31] and has made use of the resources provided by the Edinburgh Compute and Data Facility (ECDF) [32]. The ECDF is partially supported by the eDIKT initiative [33].

9. REFERENCES

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

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

[3] 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*, 2007.

[4] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. P. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *CASES*, 2006.

[5] 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.

[6] 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), 2004.

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

[8] 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.

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

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

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

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

[13] Intel Corporation. Intel XScale microarchitecture. http://www.intel.com/design/intelxscale/.

[14] G. Contreras et al. XTREM: a power simulator for the Intel XScale core. In *LCTES*, 2004.

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

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

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

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

[19] 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.

[20] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3), 2004.

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

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

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

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

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

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

[27] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *PACT*, 2006.

[28] 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.

[29] M. Zhao, B. R. Childers, and M. Lou Soffa. A model-based framework: An approach for profit-driven optimization. In *CGO*, 2005.

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

[31] Milepost. http://www.milepost.eu.

[32] The edinburgh compute and data facility (ECDF). http://www.ecdf.ed.ac.uk.

[33] The eDIKT initiative. http://www.edikt.org.