Exploring and Predicting the Effects of Microarchitectural Parameters and Compiler Optimisations on Performance and Energy

CHRISTOPHE DUBACH, TIMOTHY M. JONES and MICHAEL F.P. O'BOYLE
School of Informatics
University of Edinburgh, UK
christophe.dubach@ed.ac.uk,{tjones1,mob}@inf.ed.ac.uk

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 our specific design space, we demonstrate that we can automatically predict the performance that an optimising compiler would achieve without actually tuning it for any of the microarchitecture configurations considered. Once trained, a single run of the program compiled with the standard optimisation setting is enough to make a prediction on the new microarchitecture with just a 3.2% error rate on average. 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 19% performance improvement and energy savings of 16% compared to the baseline, nearly doubling the energy-efficiency measured as the energy-delay-squared product (EDD).

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Retargetable compilers*; C.4 [**Computer Systems Organisation**]: Performance of systems—*Design studies, modelling techniques*; C.0 [**Computer Systems Organisation**]: General—*Hardware/software interfaces*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Architecture/compiler co-design, design-space exploration, performance prediction

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 microarchitecture space *without* tuning the compiler first. This allows the designer to accurately determine the performance of any microarchitecture from our design space as if an optimising compiler had been tune for each of them. Given a small sample (less than 0.01%) of the microarchitecture 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 3.2% across all sampled 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 [İpek et al. 2006; Joseph et al. 2006a; Lee et al. 2007]. Other researchers have shown that it is possible to predict the impact of compiler optimisations on a fixed architecture [Vuduc et al. 2004; Cavazos et al. 2006; Cavazos et al. 2007]. In [Vaswani et al. 2007] this is taken one step further where a model predicts the performance of compiler settings on different microarchitectures 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 microarchitecture space before the compiler is tuned.

In this work we separately explore the microarchitectural and compiler optimisation spaces. We 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 architecture from our design space that consists of various cache and branch predictor configurations. 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 of 19%, 16% savings in energy and an energy-delay-squared product (EDD) of 0.55.

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 microarchitecture/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 and define the notion of an *optimising compiler*.

2.1 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 [Triantafyllis et al. 2003; Kulkarni et al. 2004; Cooper et al. 2005; Haneda et al. 2005; Pan and Eigenmann 2006; Cavazos et al. 2007] 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.

2.2 Microarchitecture / Compiler Co-Design Space

To evaluate the effectiveness of co-design space exploration, we chose to use the Intel XScale processor [Intel Corporation] as our baseline architecture. This processor uses the ARM ISA and is typically found in embedded systems. Its configuration is shown in table I, 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 compiler 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 microarchitectural configurations and compiler optimisations. We describe these in more detail in sections 3.1 and 3.3.

2.3 Experimental Framework

We used the Xtrem simulator [Contreras et al. 2004] which has been validated for cycles and energy consumption against the Intel XScale processor. Using Cacti [Tarjan et al. 2006] we accurately modelled the access latencies of each cache configuration to ensure our experiments were as realistic as possible.

We used the full MiBench suite [Guthaus et al. 2001] 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.

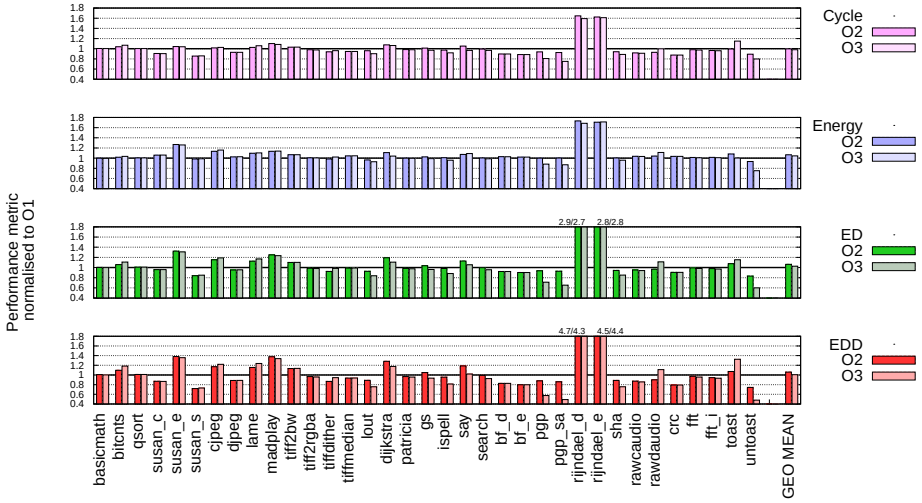


Fig. 1. Cycles, energy, ED and EDD for each MiBench program when compiled with **O2** and **O3**, normalised to **O1** (lower than 1 means better than **O1**).

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. 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) and energy-delay-squared (EDD) products. These two metrics are widely used among architects and represent the trade-offs between performance and energy consumption in a single value, the lower the better.

2.4 Baseline Optimisation Level

We wanted to ensure that our baseline compiler optimisation level was realistic and effective. We therefore considered the three default optimisation levels available in gcc: **O1**, **O2** and **O3**.

Figure 1 shows the performance, energy consumption, ED and EDD 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 trade-off between performance and energy, ED and EDD, 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.

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

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

3. MICROARCHITECTURE AND COMPILER DESIGN IN ISOLATION

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

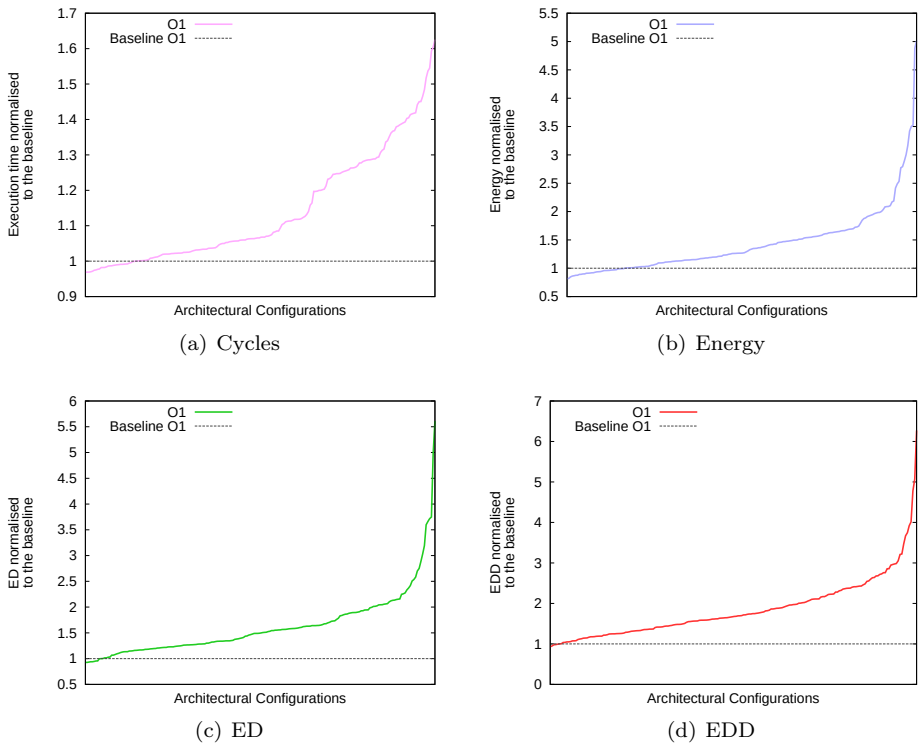
3.1 Microarchitecture Design Space

We have picked a typical microarchitectural design space, whose parameters are shown in table I. 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 [Intel Corporation]. 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 our microarchitectural design space. Each graph shows the performance achieved by each microarchitectural configuration in terms of execution time, energy, ED and EDD 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% and within the top 2% for EDD. 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, ED and EDD 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. 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 (the average saving in energy is 19%). However, the ED value for some benchmarks is over 1 because this configuration actually loses



(a) Cycles (b) Energy

(c) ED (d) EDD

Fig. 2. The average execution time, energy, ED and EDD 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.

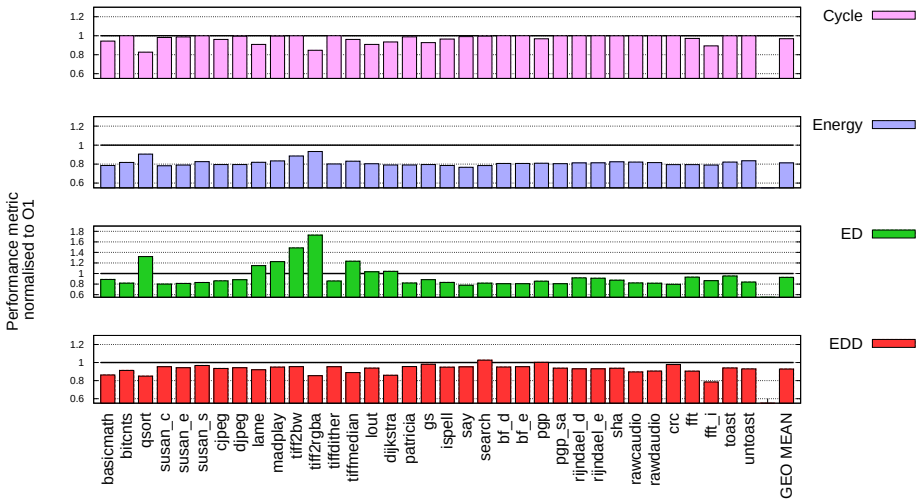


Fig. 3. Execution time, energy, ED and EDD for each benchmark compiled with **O1** on the microarchitectural configuration performing best for each metric.

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

N°	Flag	Values
1	-fthread-jumps/⊘	
2	-fcrossjumping/⊘	
3	-foptimize-sibling-calls/⊘	
4	-fcse-follow-jumps/⊘	
5	-fcse-skip-blocks/⊘	
6	-fexpensive-optimizations/⊘	
7	-fstrength-reduce/⊘	
8	-frerun-cse-after-loop/⊘	
9	-frerun-loop-opt/⊘	
10	-fcaller-saves/⊘	
11	-fpeephole2/⊘	
12	-fregmove/⊘	
13	-freorder-blocks/⊘	
14	-falign-functions/⊘	
15	-falign-jumps/⊘	
16	-falign-loops/⊘	
17	-falign-labels/⊘	
18	-ftree-vrp/⊘	
19	-ftree-pre/⊘	
20	-funswitch-loops/⊘	
21	-fgcse/⊘	
22	-fno-gcse-lm/⊘	
23	-fgcse-sm/⊘	
24	-fgcse-las/⊘	
25	-fgcse-after-reload/⊘	
26	–param max-gcse-passe =	1,2,3,4
27	-fschedule-insns/-fschedule-insns2/⊘	
28	-fno-sched-interblock/⊘	
29	-fno-sched-spec/⊘	
30	-finline-functions/⊘	
31	–param max-inline-insns-auto=	10,30,...,190
32	–param large-function-insns=	1300,1500,...,3300
33	–param large-function-growth=	20,50,100,200,...,500
34	–param large-unit-insns=	4000,6000,...,20000
35	–param inline-unit-growth=	10,20,30,...,100,200,300
36	–param inline-call-cost=	10,12,...,30
37	-funroll-loops/-funroll-all-loops/⊘	
38	–param max-unroll-times=	2,4,6,...,20
39	–param max-unrolled-insns=	50,75,100,...,400

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 and EDD 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.

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 II. It is similar to the optimisations considered by other researchers [Vaswani et al. 2007], 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.

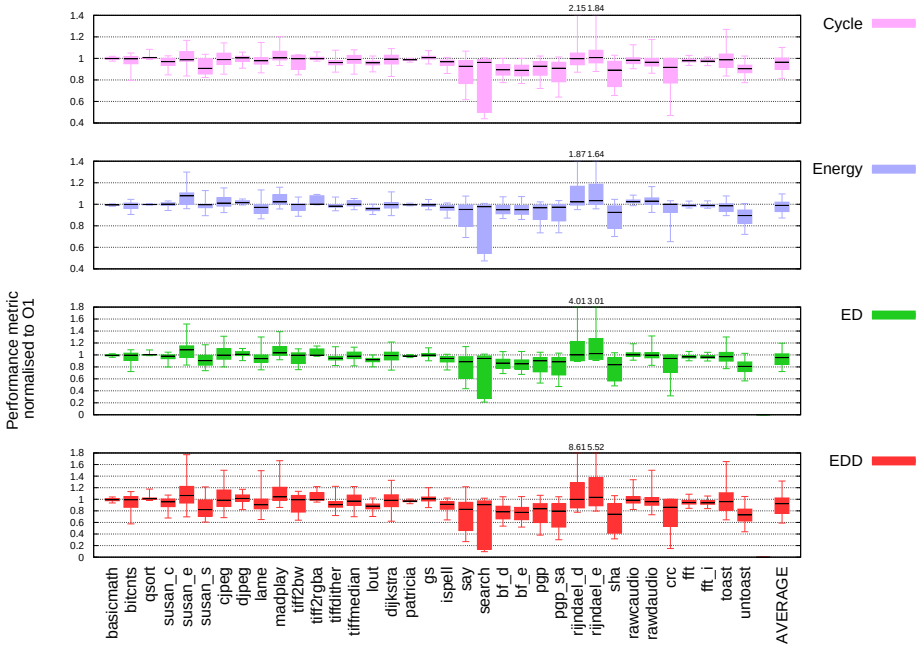


Fig. 4. Distribution of the compiler optimisation design space on a per-benchmark basis for execution time, energy, ED and EDD (the lower the better). The bottom and top of the box represent the 25% and 75% quantiles respectively, the band near the middle the median whilst the bottom and top ends of the whiskers represent the minimum and maximum respectively.

3.4 Compiler Optimisation Exploration

Figure 4 shows the execution time, energy, ED and EDD 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 or EDD, 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 and achieve an ED value of 0.72 or an EDD value of 0.60. This is compared to the best ED value of 0.93 when varying the microarchitectural

space alone. Not surprisingly, there is more room for improvement in the compiler space because the architecture has already been significantly tuned.

3.5 Different Optimisations For Each Program

Knowing the performance of the best flags for each application is meaningful but it is also important to see whether these flags are actually different from program to program. To understand which optimisation parameters have an impact on performance and energy, we have taken each program individually and optimised them for EDD. Then the optimisation settings were ranked from the best to the worst EDD values and those within the top 5% of the best (100% being the performance of **O1**) were retained. The first ten best were always retained independently of their performance to ensure there were enough flag settings to conduct an analysis.

Each boolean flag is marked as important if it is turned on or off at least 90% of the time within the set of the top 5%. For the parameter flags such as *max-unroll-times*, the flags are marked as important if the standard deviation within the top 5% was significantly lower than the standard deviation across all the settings. The value reported in this case is simply the mean of the parameters present in the top 5%.

Figure 5 shows the values of the important flags for each benchmark individually, since the best combination of optimisation flags varies from program to program. As can be seen the importance of the flags and their corresponding values are dependent on the program for some of the flag settings. For instance consider optimisation number 27 (*fschedule-insns*): for program susan_s it is better to disable this optimisation, whereas for program pgp it is better to enable it with instruction scheduling policy 1 and for program rawcaudio with policy 2. These two policies influence when the instruction scheduling will be performed (before register allocation in one case and after in the other). This clearly shows that the flag settings that achieve the best EDD are different from program to program.

3.6 Summary

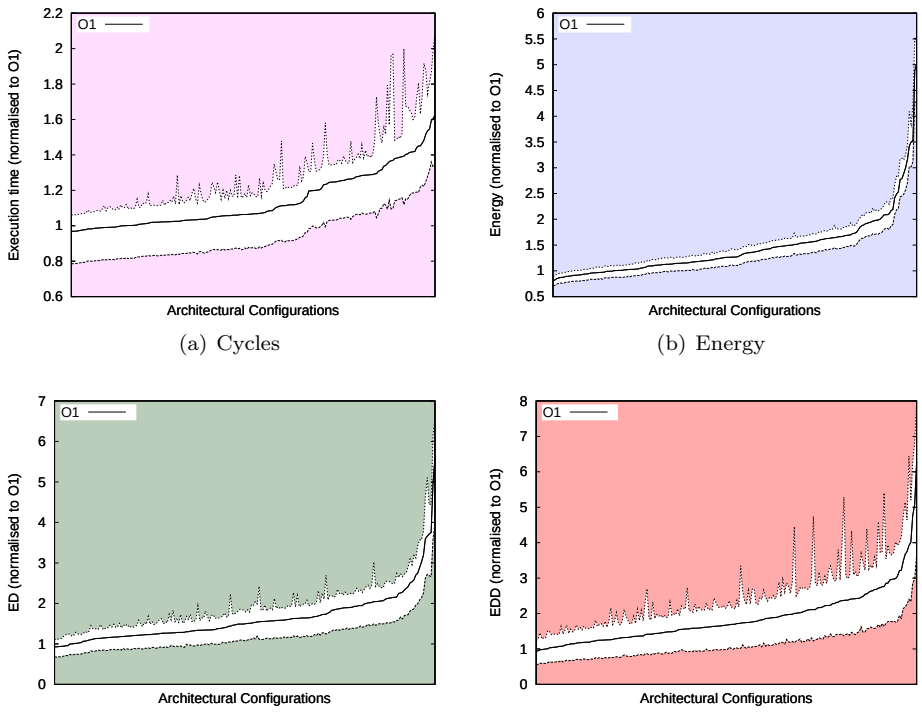
Up to this point we have considered the microarchitecture and compiler optimisation spaces independently. In particular we have shown that the efficiency of the architecture could be improved, reducing the energy consumed by 19%. In the compiler space, we have shown that execution time could be reduced by 19% when selecting the right set of optimisations to apply per program.

The next section combines these spaces, considering their co-design and explores the improvement that is available in terms of execution time and energy.

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 when 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.

Fig. 5. Important flags and their corresponding values that lead to the best EDD value for all the MiBench programs on the baseline architecture. As can be seen the optimisations that lead to the best EDD are different from program to program.



(a) Cycles (b) Energy

(c) ED (d) EDD

Fig. 6. The co-design space for execution time, energy, ED and EDD 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 each architecture. Each graph is independently ordered from lowest to highest and is normalised by **O1** on the baseline configuration.

4.1 Exploration

Figure 6 shows the co-design space across microarchitectural configurations for execution time, energy, ED and EDD. 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, ED or EDD 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 clear that there is large room for improvement over the baseline compiler optimisation across the whole microarchitectural space in terms of execution time, ED and EDD. All four 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 7 we show the execution time, energy, ED and EDD values on a per-

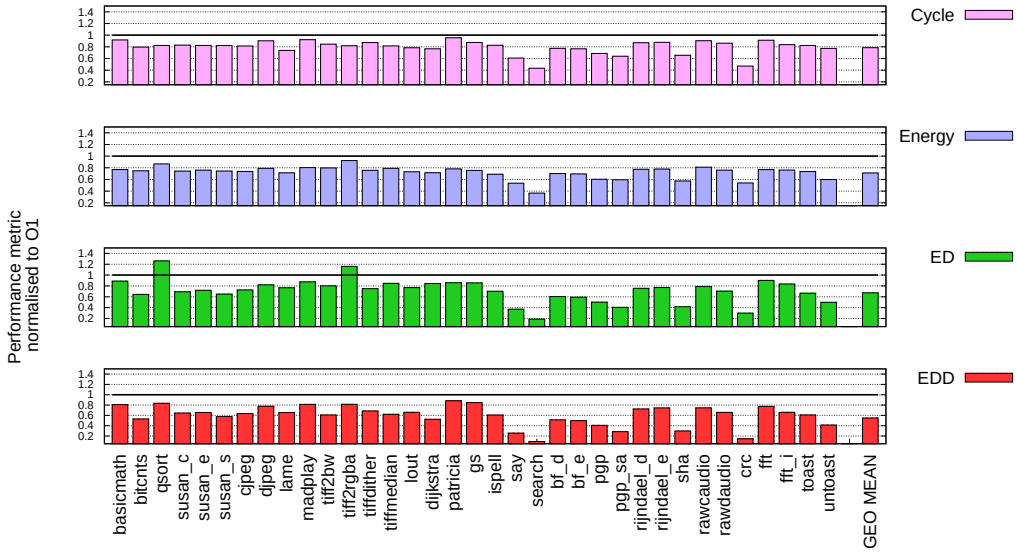


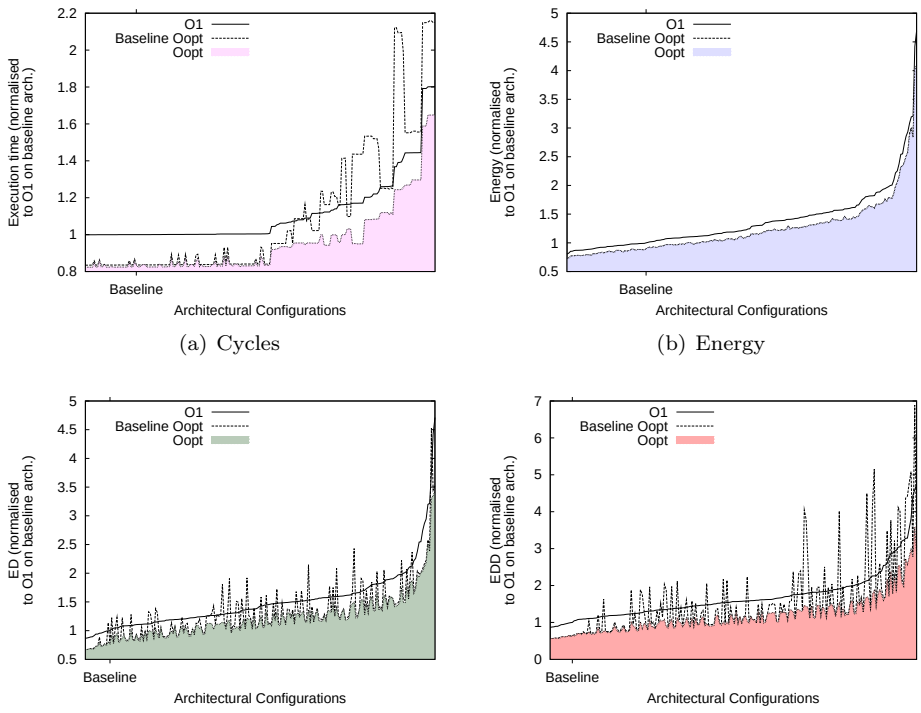
Fig. 7. Execution time, energy, ED and EDD for each benchmark on the microarchitectural/optimisation configuration performing the best for each metric. The microarchitecture is the same across programs however it varies depending on the target metric.

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). Similar conclusions can be drawn for the EDD metric. 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 and an EDD value of 0.55. This is compared with an ED 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

The previous sections have shown that co-design space exploration is beneficial over performing microarchitecture and optimisation space exploration in isolation. This section now considers the effects of finding the best flag settings for a specific program on the baseline architecture and using them across the microarchitectural



(a) Cycles (b) Energy

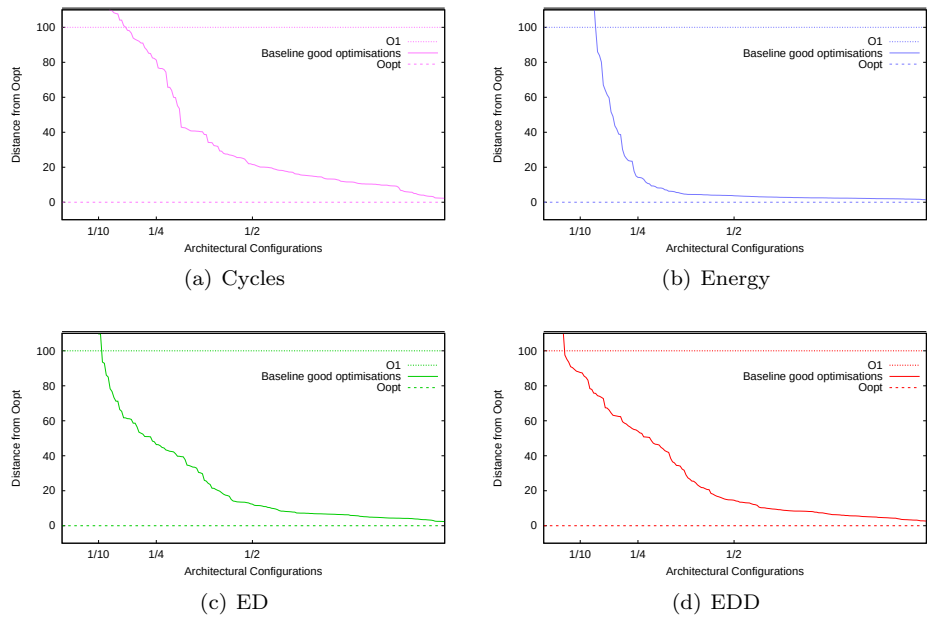
(c) ED (d) EDD

Fig. 8. Optimising program toast on the baseline architecture and running it on all other microarchitectural configurations. All the values are normalised by **O1** on the baseline microarchitecture.

space for the same program. In other words, we wish to examine whether a fixed set of optimisation flags exists for a particular program that can lead to the best performance independently of the microarchitecture.

Figure 8 shows an example for the toast benchmark when optimising the program on the baseline architecture and running it on the other configurations. As can be seen, the optimisations that are the best for the baseline microarchitecture actually perform worse than compiling with **O1** on other configurations for cycles, ED and EDD. Critically, the best compiler optimisations vary across the microarchitecture space. However, for energy it seems to make little difference. As seen earlier, this is due to the fact that the compiler optimisations have very little impact on the energy consumption.

Figure 9 shows this averaged across all benchmarks. Here we have run all programs using 1000 optimisations on the baseline architecture and those optimisations that are within 5% of the best found for each benchmark were selected (a different set for each program). They are called the *baseline good* optimisations. Then, we have run the benchmarks compiled with these baseline good optimisations on the rest of the microarchitectures to determine the average cycle, energy, ED and EDD values that they achieve. For each configuration, the performance of the *baseline good* optimisations were evaluated using the distance from the best value achievable on that configuration. We normalised the distances by the performance of the best



(a) Cycles (b) Energy

(c) ED (d) EDD

Fig. 9. 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. All the results are averaged across all benchmarks.

optimisation (distance=0%) and the performance of **O1** (distance=100%).

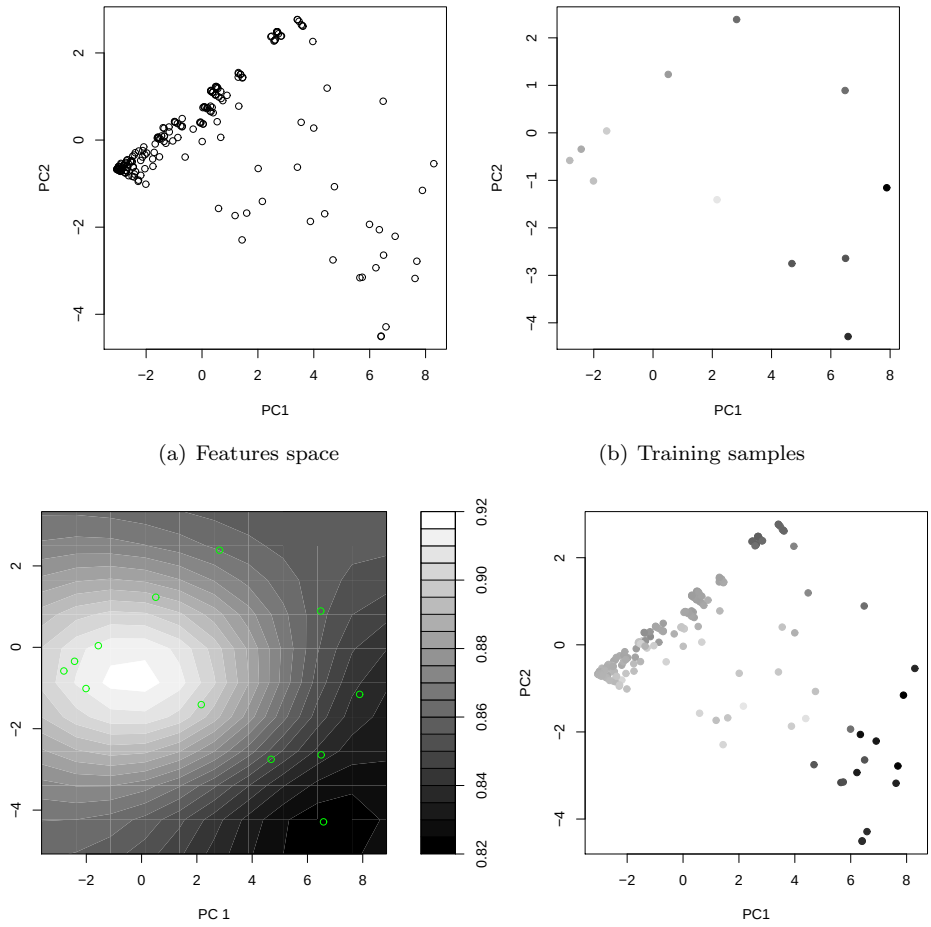
For ED and EDD we can see that on half (1/2) the architectures the good optimisations for the baseline are at least 15% away from the best. For a quarter (1/4) of the architectures, these good optimisations are at least around 50% away from the best. Crucially, the good optimisations on the baseline architecture are actually worse than **O1** for one tenth (1/10) of the microarchitectures. This shows that good optimisations for one architecture are not necessary suitable for others. In essence, the optimal compiler optimisations to apply for one architecture are not the best for all. Therefore the compiler has to 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



(a) Features space

(b) Training samples

(c) Trained model (SVM)

(d) Real space

Fig. 10. Example use of the model for the program fft when considering ED (the darker a dot, the better the ED value is over the baseline **O1**). First performance counters are collected, then PCA is used to select two components (a). Training configurations are then selected and a search of their optimisation spaces for the best performance is conducted (b). Finally, the SVM model is trained to determine the contour map around configurations (c). This map provides a prediction of the real performance of the optimising compiler (d).

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 variation of the microarchitectural space considered in section 3.1.

Table III. The performance counters used to characterise the microarchitectures.

Parallelism	Resource usage	Cache information
Instructions per cycle	ALU / MAC / Shifter usage	Insn cache access rate
	Decoder access rate	Insn cache miss rate
	Register file access rate	Data cache access rate
	Branch pred. access rate	Data cache miss rate

5.1 Overview

Our model is built in three steps, with an example on the benchmark fft shown in figure 10. 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 10(a)). From this we can select a number of architectural configurations for training (figure 10(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 10(c)) and predictions can be made for the entire space. These predictions can be directly compared with the real space (figure 10(d)). The next sections describe these 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 from our space. 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 nine performance counters to extract shown in table III. Performance counters like these are typically found in processor analytic models [Karkhanis and Smith 2004; Eyerman et al. 2006]. We then use Principal Component Analysis (PCA) to summarise the nine features into two values, or principal components. Figure 10(a) shows these components (PC1 & PC2) over 200 microarchitectures for the benchmark fft.

5.3 Gathering Training Data

Before being able to build our model, a few microarchitectures need to be selected to train the model with. For each of these, an exploration of the optimisation space is required to obtain the performance of the best optimisation settings. For this reason it is important to select carefully the training configurations so as to avoid unnecessary explorations.

To achieve high efficiency, we select the training samples that best cover the projected space. We therefore use the K-Means algorithm [Lloyd 1982] to find clusters of microarchitectures based on the performance counters. Then one representative microarchitecture is selected for each cluster followed by an exploration of the optimisation space for that particular microarchitecture. Figure 10(b) shows the

configurations being selected for training for the benchmark fft with a number of clusters set to 12. The optimal number of clusters selected is discussed later in section 6.2 where the technique is evaluated. As can be seen the selected microarchitectures successfully cover the space.

Once this selection has taken place, we search the compiler optimisation space on each of the selected microarchitectures using iterative compilation with 1000 randomly-selected optimisations. Note that this search could be made more efficient by using more advanced search strategies [Triantafyllis et al. 2003; Almagor et al. 2004; Cooper et al. 2005; Haneda et al. 2005]. However, this is orthogonal to the focus of this paper. The result of this search corresponds to an estimation of the maximum performance achievable on each of the selected training microarchitectures. This is shown in figure 10(b) where darker points lead to better performance. With this data gathered, the model is ready to be trained.

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 [Smola and Schölkopf 2004]. 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 10(c). Here we have circled the training configurations. The model learns the areas of similar colour based on the best performance seen on the 12 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 10(d).

Having trained our model, we can predict the performance of the optimising compiler on any new microarchitectural configuration within our space. To do this we run **O1** on the architecture and gather performance counters. The model uses PCA to reduce the number of features to two 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 within our space. We first run **O1** on 200 architectures and gather performance counters. We use PCA to reduce these and then select a few 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 simply 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 also compare it 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 Methodology

Our model is evaluated using *cross-validation*. This technique ensures that the programs used to test the model are not used, or *seen*, during the training phase. This allows a fair evaluation and is standard practice among the machine-learning community.

To measure the performance of our model we use the relative mean absolute error and the coefficient of correlation. The relative mean absolute error is defined as:

$$\textbf{rmae} = \frac{1}{N}\sum_{i}^{N}\left|\frac{\textit{predicted value}_i - \textit{real value}_i}{\textit{real value}_i}\right| \tag{1}$$

and the coefficient of correlation as

$$\textbf{correlation} = \frac{\textbf{cov}(\textit{predicted value}, \textit{real value})}{\sigma_{\textit{predicted value}} \cdot \sigma_{\textit{real value}}} \tag{2}$$

where $\sigma_{\textit{predicted value}}$ and $\sigma_{\textit{real value}}$ are the standard deviations of the *predicted value* and *real value* respectively and $\textbf{cov}(\textit{predicted value}, \textit{real value})$ is the covariance between the *predicted value* and the *real value*. These two functions are finally defined as:

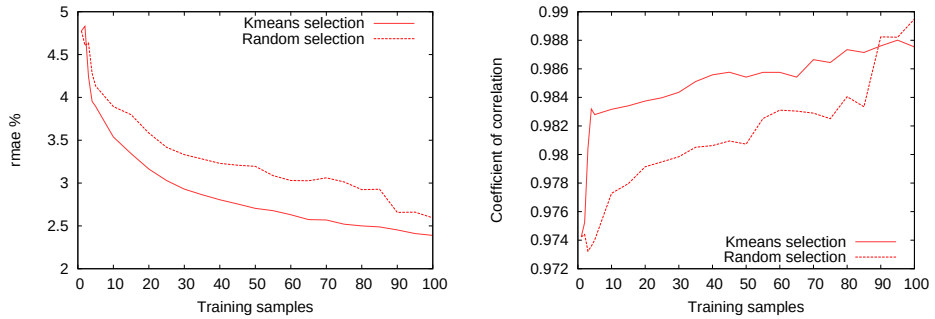
$$\begin{aligned} \sigma_X &= \sqrt{\tfrac{1}{m}\textstyle\sum_i (X_i - \overline{X})^2} \\ \textbf{cov}(X,Y) &= \tfrac{1}{m}\textstyle\sum_i^m (X_i - \overline{X}) \cdot (Y_i - \overline{Y}). \end{aligned} \tag{3}$$

The correlation coefficient only produces values between -1 and 1. The larger this value is, the stronger the relationship between the two variables (ignoring the sign). At the extreme, a correlation of 1 means that both variables are perfectly positively correlated; one variable can be expressed as the product of the other (linear relation). A correlation of 0 means that there is no linear relationship between these two variables.

6.2 Training Samples Selection: K-Means vs Random

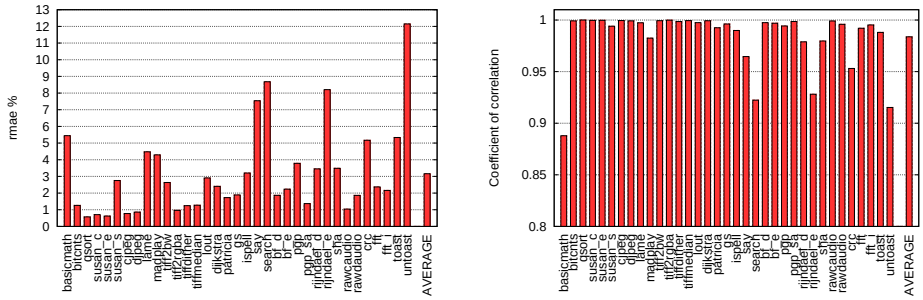
As seen in the previous section which described our model, a few microarchitectures are selected in order to train the predictor. This selection process is performed by using the K-Means clustering technique to pick a representative microarchitecture for each cluster found. This procedure is now evaluated for different training sizes and compared with a purely random selection process.

Figure 11 shows the mean error and the coefficient of correlation of the model for various training sizes when predicting for EDD. These results are averaged across all the benchmarks and obtained using cross-validation where the microarchitectures used for training are left out of the testing set. This figure clearly shows that selecting the training points with the K-Means algorithm is better than using a purely random selection. For instance with 20 training samples, our K-Means approach achieves an error rate of just 3.2% and a correlation of 0.984 whereas the random selection achieves only 3.7% and 0.979 respectively. This also shows that a low error can be achieved using only a fraction of the design space.



(a) Error (b) Correlation

Fig. 11. Mean error and coefficient of correlation for out model using both K-Means and a random selection process to choose our training microarchitectures. This is averaged across all programs for varying training sample sizes.



(a) Error (b) Correlation

Fig. 12. The prediction errors of the model broken down by program when predicting the EDD value achievable by the optimising compiler using only 20 training microarchitectures. The average error is just 3.2% and the average correlation 0.98.

It might at first seem surprising to see that the coefficient of correlation is high even when using very few training samples. Looking back at figure 6 it can be seen that the performance of the best flag settings is strongly correlated with the performance of the baseline optimisation **O1**. This is due to the fact that the microarchitectural space has a much higher variance than the compiler optimisation space. Furthermore, it is important to keep in mind that these numbers are averaged across all programs. Hence, programs that show very little variation in their optimisation space will tend to be easier to predict, independently of the number of training points.

In the following sections, the training budget is fixed to 20 training samples because this represents a good trade-off between accuracy and number of samples. As the next section shows, this leads to a very good correlation and relatively low error for most of the programs.

6.3 Prediction Accuracy Per Program

The prediction error and coefficient of correlation for each benchmark in MiBench is shown in figure 12 for 20 training samples. As stated earlier, these 20 samples

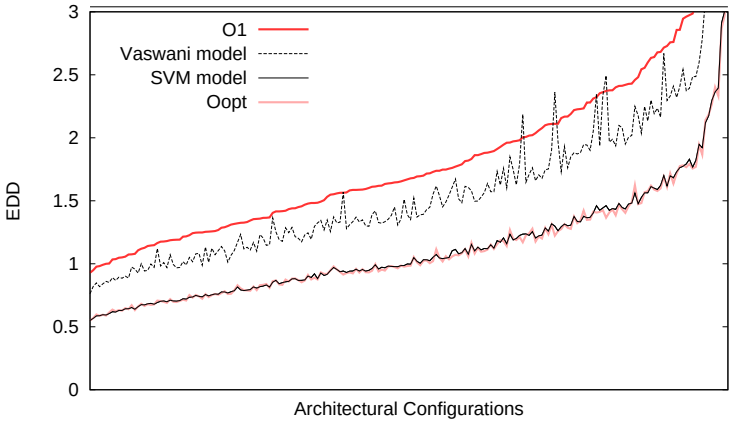


Fig. 13. Predicting the performance of the optimising compiler across the microarchitectural space for the whole of MiBench. Also shown are the predictions made by Vaswani's model. Note that our model is highly accurate and overlaps significantly with **Oopt**.

correspond to 20 microarchitectures on which the best performance achievable was estimated using 1000 random optimisations. The average error and correlation of the whole suite is also shown.

As can be seen, our model achieves a very low error rate of 5% or under for the majority of the benchmarks. In fact, for some benchmarks (such as susan_e), the error is as low as 0.6%. The coefficient of correlation is also very good for all the benchmarks, the lowest achieving a correlation of 0.88.

This shows that our model is accurate and correctly predicts the performance of the optimising compiler. In the next section we conduct a comparison with a state-of-the-art technique for co-design space exploration.

6.4 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 optimisation space. We have built the model proposed by Vaswani et al. [2007] 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 and EDD values 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 13 shows the EDD value achieved by the baseline compiler on each microarchitectural configuration averaged over the whole of MiBench (labelled O1). It also shows the EDD 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 (SVM model).

Table IV. Parameters and EDD value of the best configuration found using the SVM model.

(a) Microarchitectural parameters

Configuration	Icache			DCache			BTB	
	size	assoc.	block	size	assoc.	block	size	assoc.
Best EDD	32K	64	16	64K	32	32	256	4

(b) Best value predicted and achieved

	Predicted EDD		Real EDD
	SVM model	*Vaswani* model	
Best EDD	0.550	0.762	**0.549**

It is immediately obvious that our predictions follow the curve of the optimising compiler with great fidelity. More specifically, our model accurately predicts the peaks and troughs in EDD 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 EDD where there are none and follows the **O1** line closely. This predictor, therefore, is inappropriate for finding the performance of the optimising compiler.

6.5 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 EDD 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 EDD value in the co-design space of 0.550 on the configuration shown in table IV.

To verify the prediction accuracy, we used iterative compilation on this architecture with 1000 randomly-selected optimisation settings. We found that the best EDD value achievable is 0.549. This is just 0.2% away from our prediction, showing that our model is very accurate. Had we used the *Vaswani* model, it would have predicted an EDD value of 0.762 for this configuration, which is an error of 39%.

In addition to this, we wanted to verify that this prediction is actually the best EDD 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 EDD value in the sample space.

This best microarchitectural configuration found by our model achieves a performance increase of 19% and energy savings of 16% compared to the baseline. It produces the smallest EDD value because it is well balanced. The instruction and data caches have high associativity to avoid conflicts. The instruction cache has the same size as the baseline configuration (table I) and the data cache is double

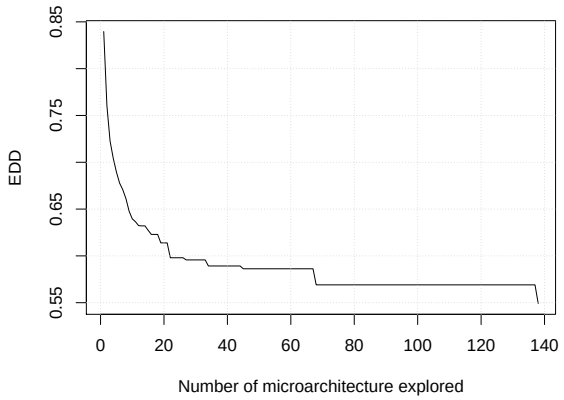


Fig. 14. Best EDD achieved as a function of the number of architectures explored when using an iterative search of the microarchitecture space. On average, 140 microarchitectures need to be explored to match the performance of our model.

the size of the baseline to improve performance without significant increase in the energy consumed.

6.6 Search Cost

We are now interested in evaluating the benefits of our approach over a simple approach which searches iteratively through the space. If we were to pick randomly a microarchitecture from our space, search its optimisation space for the minimum EDD and repeat this process iteratively until we found an EDD value as good as the one found by our model, we would need to explore nearly 140 microarchitectures as shown in figure 14. With our model, this EDD value is in fact achieved by exploring only 20 microarchitectures as seen previously. This means that our technique needs to explore seven times fewer architectures than an iterative approach. As the space becomes larger and more complex, our technique will even show greater speedup over iterative approaches since only a fraction of the space needs to be explored.

7. RELATED WORK

7.1 Design Space Exploration Using Predictive Modelling

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 [Joseph et al. 2006a], artificial neural networks [İpek et al. 2005;2006], radial basis functions [Joseph et al. 2006b] and spline functions [Lee and Brooks 2006; 2007]. All have similar accuracy [Lee et al. 2007]. The second type of model makes use of prior-knowledge, learning across programs. Linear regression [Dubach et al. 2007b], artificial neural networks [Khan et al. 2007] and program-features based predictors [Hoste et al. 2006] have been proposed. However, since none of these models consider the compiler optimisation space, sub-optimal microarchitectural designs could be chosen.

7.2 Compilation Optimisation Space Exploration

Searching the compiler optimisation space has been extensively explored in the literature. Feedback-directed optimisation [Triantafyllis et al. 2003; Almagor et al. 2004; Kulkarni et al. 2004; Cooper et al. 2005; Haneda et al. 2005; Agakov et al. 2006; Pan and Eigenmann 2006] uses different algorithms to search the optimisation space. Agakov et al. [2006] built a model offline that is used to guide search. Haneda et al. [2005] make use of statistical inference to select good optimisations. Cooper et al. [2004; 2005] explore the optimisation space using hill climbing and genetic algorithms. Other researchers have used analytical [Zhao et al. 2005] or empirical [Vuduc et al. 2004;Cavazos et al. 2006;2007;Dubach et al. 2007a] models to explore the optimisation space. In fact, these techniques are orthogonal to our approach and can be used to reduce training costs.

7.3 Architecture/Compiler Co-Design Framework

The integration of compiler and architecture development is not new and has been the focus of prior research over the last 10 years. Frameworks such as Buildabong [Fischer et al. 2001], Trimaran [Trimaran 2000] or Pico [Abraham and Rau 2000] allow automatic exploration of both compiler and architecture spaces. The compiler and the simulator live side by side and are often tightly coupled within these frameworks, allowing great flexibility in terms of space exploration. The LISATek [Leupers et al. 2005] is a commerical tool based on the LISA language. It can generate a compiler and a simulator automatically, in addition to generating the HDL code ready for synthesis. These tools can be used and have been used to explore the co-design space [Fischer et al. 2002]. However, this work relies on search algorithms and has not considered the inclusion of predictive models.

7.4 Co-Design Space Exploration

Co-design space predictors [Vaswani et al. 2007] 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 6.4, this model fails to capture interactions between compiler optimisations and microarchitecture and cannot be used to accurately predict the performance of an optimising compiler.

Analytic models [Silvano1 et al. 2007] have also been proposed as an efficient way to explore the architecture/compiler co-design space. Unfortunately this approach requires a large amount of knowledge about the microarchitecture it attempts to explore and, furthermore, these models need to be built by hand. In contrast, our technique focuses on building such a predictor automatically.

Desmet et al. [Desmet et al. 2009] conducted an exploration of the microarchitectural and compiler optimization co-design space but did not propose any kind of prediction mechanism. Finally, this present paper is an extension to the work published in Cases [Dubach et al. 2008] in 2008. It presents additional results for the EDD metric, extends the analysis section by showing which optimisation are good, evaluates the training process and its impact on accuracy, shows the accuracy of our model in terms of correlation and compares the cost of using iterative search to find the best configuration in our sampled space with our technique.

8. CONCLUSION

This paper has addressed the co-design space problem by automatically predicting the performance of an optimising compiler on any microarchitectural configuration from our space, without needing to tune 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 3.2% when predicting EDD. We used this predictor to find the best optimizing compiler/architectural configuration for EDD in our design space and shown that our model could predict its EDD value with just a 0.2% error. This best configuration found achieves a 19% performance increase and 16% energy savings, leading to an EDD value of 0.55.

ACKNOWLEDGMENT

This work has been supported by Milepost [mil], the Royal Academy of Engineering and EPSRC. Additionally, it has made use of the resources provided by the Edinburgh Compute and Data Facility (ECDF) [ecd]. The ECDF is partially supported by the eDIKT initiative [edi].

REFERENCES

The eDIKT initiative. http://www.edikt.org.

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

Milepost. http://www.milepost.eu.

Abraham, S. G. and Rau, B. R. 2000. Efficient design space exploration in pico. In *CASES*.

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

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

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

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

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

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

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

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

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

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

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

Fischer, D., Teich, J., Thies, M., and Weper, R. 2002. Efficient architecture/compiler co-exploration for asips. In *CASES*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Leupers, R., Hohenauer, M., Ceng, J., Scharwaechter, H., Meyr, H., Ascheid, G., and Braun, G. 2005. Retargetable compilers and architecture exploration for embedded processors. In *Computers and Digital Techniques, IEE Proceedings*. Vol. 152. 209 – 223.

Lloyd, S. 1982. Least squares quantization in pcm. *IEEE Transactions on Information Theory 28,* 2 (Mar), 129–137.

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

Silvano1, C., Agosta1, G., and Palermo, G. 2007. Efficient architecture/compiler co-exploration using analytical models. *Design Automation for Embedded Systems 11,* 1, 1–23.

Smola, A. J. and Schölkopf, B. 2004. A tutorial on support vector regression. *Statistics and Computing 14,* 3.

Tarjan, D., Thoziyoor, S., and Jouppi, N. P. 2006. Cacti 4.0. Tech. Rep. HPL-2006-86, HP Laboratories Palo Alto.

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

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

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

Vuduc, R., Demmel, J. W., and Bilmes, J. A. 2004. Statistical models for empirical search-based performance tuning. *Int. J. High Perform. Comput. Appl. 18,* 1.

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