Reducing the Cost of Heuristic Generation with Machine Learning

William F. Ogilvie

Doctor of Philosophy
Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh
2017

Abstract

The space of compile-time transformations and or run-time options which can improve the performance of a given code is usually so large as to be virtually impossible to search in any practical time-frame. Thus, heuristics are leveraged which can suggest good but not necessarily best configurations. Unfortunately, since such heuristics are tightly coupled to processor architecture performance is not portable; heuristics must be tuned, traditionally manually, for each device in turn. This is extremely laborious and the result is often outdated heuristics and less effective optimisation.

Ideally, to keep up with changes in hardware and run-time environments a fast and automated method to generate heuristics is needed. Recent works have shown that machine learning can be used to produce mathematical models or rules in their place, which *is* automated but not necessarily fast. This thesis proposes the use of active machine learning, sequential analysis, and active feature acquisition to accelerate the training process in an automatic way, thereby tackling this timely and substantive issue.

First, a demonstration of the efficiency of active learning over the previously standard supervised machine learning technique is presented in the form of an ensemble algorithm. This algorithm learns a model capable of predicting the best processing device in a heterogeneous system to use per workload size, per kernel. Active machine learning is a methodology which is sensitive to the cost of training; specifically, it is able to reduce the time taken to construct a model by predicting how much is expected to be learnt from each new training instance and then only choosing to learn from those most profitable examples. The exemplar heuristic is constructed on average 4x faster than a baseline approach, whilst maintaining comparable quality.

Next, a combination of active learning and sequential analysis is presented which reduces both the number of samples per training example as well as the number of training examples overall. This allows for the creation of models based on noisy information, sacrificing accuracy per training instance for speed, without having a significant affect on the quality of the final product. In particular, the runtime of high-performance compute kernels is predicted from code transformations one may want to apply using a heuristic which was generated up to 26x faster than with active learning alone.

Finally, preliminary work demonstrates that an automated system can be created which optimises both the number of training examples as well as which features to select during training to further substantially accelerate learning, in cases where each feature value that is revealed comes at some cost.

Lay Summary

In order to optimise software for performance, heuristics are used to estimate good settings for compiling and running code on a system. Traditionally these heuristics are created by people with knowledge and experience of software–hardware interaction, who can guess how to get the most from a machine under varying conditions. The problem with this technique is that these heuristics are inherently tied to the underlying hardware architecture, and owing to the arduous nature of this work, combined with the speed at which technology moves, heuristics are often left outdated.

Machine learning techniques have been shown to be able to automatically create heuristics, based on mathematical models or rules, which can be more effective than those built by human experts; but, unfortunately, the time needed to create them in this way is prohibitive. This thesis proposes three techniques which can be used to accelerate this automated heuristic generation process using a mixture of active learning, sequential analysis, and active feature acquisition.

First, it is shown that active learning is on average 4x faster than current approaches at producing an exemplar heuristic. Where previous machine learning implementations learn at random how to map the characteristics of a program to performant settings active learning attempts to select carefully what would be most beneficial to learn next in order to save time.

Second, since measuring performance in computer experiments produces noisy results it is often necessary to evaluate an optimisation strategy some number of times before its effect can be understood. The number of samples in previous works have always been fixed, which is potentially wasteful. By dynamically determining how many samples are needed for each training example in turn, based on the information already collected, learning can be accelerated. In particular, a model which can predict runtime from code transformations is created up to 26x faster with this sequential analysis approach included than with active learning alone.

Finally, in machine learning applications data scientists have to work out which features to measure in order to best map these to target values, however, good features are usually not known *a priori*. It is demonstrated in this thesis that as well as optimising the number of samples per training example, and the number of training examples, it is also possible to optimise the number and selection of features simultaneously and automatically at learning-time, and where these features come at a cost training is further accelerated.

Acknowledgements

I would like to take this opportunity to thank my first teachers—my parents, Billy and Norma Ogilvie—for all their love and support, and for always believing in me and encouraging me to follow my heart.

I would also like to thank my primary supervisor, Hugh Leather, particularly for his patience over the years. He's one of the most gifted people I have ever had the pleasure of meeting, has taught me more than I could possibly enumerate, and is not just a mentor but a friend. Likewise, the same can be said for my 'academic uncle' Pavlos Petoumenos, and my co-supervisor Zheng Wang. I feel I have been incredibly fortunate in being able to collaborate with these individuals, and I hope we remain in touch throughout the remaining chapters of my life.

Lastly, I must thank my wife Gillian. She is my soulmate, a fantastic mother to our beautiful children, and I simply couldn't have completed this without her help.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material in this thesis has been published in the following papers:

- William F. Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather **'Active Learning Accelerated Automatic Heuristic Construction for Parallel Program Mapping'**. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014

- William F. Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather **'Fast Automatic Heuristic Construction Using Active Learning'**. In *Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2014

- William F. Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather **'Minimizing the Cost of Iterative Compilation with Active Learning'**. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2017

(*William F. Ogilvie*)

For my family

Table of Contents

Chapter 1

Introduction

It was predicted at the turn of the century that by 2016 processors would be available with clock rates in excess of 28 GHz [ITRS, 2001]; however, five years after this prognostication manufacturers such as Intel and AMD abandoned pursuing chips with ever faster frequencies in favour of multi-core platform designs [Hennessy and Patterson, 2011]. The primary reasons for this paradigm shift were three-fold, the limitation to the amount of Instruction-Level Parallelism (ILP) that can be exploited in any given instruction stream, the growing disparity between the speed of memory and CPU, and, in particular, the end of Dennard scaling [Patterson, 2006; ITRS, 2015]—the ILP, memory, and power walls, respectively.

Previously, from the perspective of software developers, performance of codes could be relied upon to improve over time for 'free' as CPU frequencies increased thanks to the shrinking of electrical components, as observed by Moore's Law, and the constancy of power required per unit area, given by Dennard scaling. Unfortunately, it was seen in the 1990s that clock frequencies in excess of 3.5 GHz have prohibitively expensive heat dissipation problems [Denning and Lewis, 2017], and more recently researchers have found that there are at least seven reasons to believe Moore's Law (as it relates to complementary metal-oxide-semiconductor technology) will not be able to continue for much longer [Wu et al., 2013]. This means that whilst the move to multi-core platforms has allowed chip manufactures to deliver increased performance from hardware in the short-term, going forward the onus for finding speed-ups will be increasingly the responsibility of the software engineer, not the platform architect.

In order to take full advantage of parallel hardware the optimal compile-time code transformations and run-time settings for a given program need to be found. Unfortunately, modern processors are extremely complex. They often have multiple cores and are becoming progressively heterogeneous [Power et al., 2013], where each core often specialises in different classes of workload [Shan, 2006] by offering either different capabilities [ARM, 2016] or distinct instruction set architectures entirely [Kahle et al., 2005]. Within each core a large number of components run in parallel and are sensitive to the behaviour of the others. Since it is often infeasible (in any reasonable time-frame) to obtain the best configuration for a code through exhaustive techniques [Massalin, 1987; Joshi et al., 2002] heuristics must be leveraged instead.

Traditionally, optimisation heuristics are fine-tuned by experts with a deep understanding of the underlying platform, and are intrinsically non-portable. Which is to say, each distinct processor requires a unique heuristic even if it is based on a previous chip design from within the same product family. This is problematic since creating heuristics in this way is laborious and extremely expensive, with compiler back-ends often incurring an investment of man-decades of work to reach maturity [Fisher et al., 2005]. Moreover, because there is a relatively quick turnaround on new hardware from processor manufacturers, and because it requires so much effort to recreate heuristics each time, these heuristics are often left outdated [Kulkarni and Cavazos, 2012]. This is a significant problem since without properly tuned heuristics good program performance cannot be expected.

1.1 Machine Learning Based Heuristic Generation

For the reasons outlined above, manually fine-tuning optimisation heuristics is no longer scalable, and the failure to always have easily to hand quality heuristics for a given architecture has ultimately resulted in poorly optimised code. In order to ameliorate this situation researchers have shown that the process of heuristic construction can be successfully automated instead. For example, iterative compilation [Aarts et al., 1997; Bodin et al., 1998; Cooper et al., 1999; Kisuki et al., 2000; Knijnenburg et al., 2002] was proposed as a means by which compiler heuristics can be computed with little human involvement. The basic idea is to apply different optimisation strategies in some systematic fashion at compile-time to examine their affect on speed, code size, power, or energy. Augmented with machine learning [Agakov et al., 2006] these data are used to form a model from which good optimisation strategies can be extracted for any specified set of features, where these features can identify a program enough to predict how to optimise it. Forming models on which to base heuristics in this way has been shown to outperform those manually crafted by human experts [Dubach et al., 2009; Kulkarni and Cavazos, 2012; Grauer-Gray et al., 2012; Wang et al., 2014]. Machine learning has also been applied to many different problem domains besides compilation, such as parallelism mapping [Grewe et al., 2013b], porting across architectural spaces [Cavazos et al., 2007], and run-time tuning [Cummins et al., 2016]. Unfortunately, despite the potential gains of this powerful technique it appears to remain relatively unappealing, evidenced by the lack of machine learning based heuristics in production systems. The intuition on which the works in this thesis are based is that it is the cost of automatically producing these models that is its key flaw. Which is to say, it can take months to gather enough data for a model to be built which is of sufficient quality to be useful: hardly better than the manually fine-tuned alternative. In an ideal world, an automated *and fast* method to produce heuristics is required, so that they can be created quickly as and when they are needed.

This thesis presents three ways in which construction of models on which to base heuristics can be substantially accelerated. Firstly, active learning [Settles, 2013] can significantly reduce the number of training instances required to form a high-quality model by concentrating on informative training examples only. Secondly, sequential analysis [Wald, 1944] can be used to reduce the number of samples *per training example*, further reducing the learning overhead. Finally, it is demonstrated that active feature acquisition [Veeramachaneni and Avesani, 2003] can be used to optimise the selection of feature values that need to be recorded during the learning process, in order to reduce training expense in circumstances where more features requires more time.

1.2 Contributions to Knowledge

The main contributions made in this thesis are as follows:

1. demonstrating that the training overhead of machine learning based heuristics can be significantly reduced without sacrificing prediction quality;

2. presenting empirical measurements which prove active learning can be used to automatically derive a heuristic to map OPENMP and OPENCL programs on a CPU–GPU based heterogeneous platform;

3. hypothesising that the training overhead of machine learning based heuristics can be further lessened through the employment of sequential analysis;

4. developing a technique which combines sequential analysis and active learning to reduce both the number of training examples and the number of samples per training example used in heuristic construction;

5. presenting evidence which proves this combination can substantially reduce the training overhead of runtime prediction models, which can be used to determine good compilation options, as compared to the state of the art;

6. and demonstrating that feature selection can also be accomplished automatically at learning-time, making data collection much more efficient and straightforward.

1.3 Thesis Structure

The rest of this thesis is composed as follows:

Chapter 2 will outline some background material necessary to fully understand the technical aspects of the presented works;

Chapter 3 will provide context as to where these works sit in relation to the broader academic literature relevant to this topic;

Chapter 4 will detail the work carried out with regards to mapping inputs to the most appropriate device in a heterogeneous system using active learning, and in particular the efficiency of that technique over previous approaches;

Chapter 5 will discuss research on the topic of combining sequential analysis and active learning, and demonstrate the potential of this idea with regards to selecting good compile-time optimisations for a fraction of the cost;

Chapter 6 will demonstrate how the selection of missing *but expensive* features values can be adaptively revealed during data collection, in such a way as to minimise learning time whilst attaining overall good quality models;

Chapter 7 will look at potential directions for future work, and give a conclusion to this thesis.

1.4 Summary

This chapter has provided a brief explanation as to why processors are increasingly multi-core and heterogeneous, and what that means for the necessity and complexity of heuristic generation; has mentioned that due to the arduous nature of manually fine-tuning such heuristics there already exist outdated heuristics that provide poor code performance; and concluded that researchers have shown machine learning to be capable of revolutionising this process, but has argued that current implementations are unnecessarily inefficient.

In the rest of this thesis it will be shown how the training overhead required to create these crucial heuristics can be substantially reduced through the application of active learning, sequential analysis, and active feature acquisition; thus, providing fast, automated processes for future heuristic generation, ensuring good optimisation decisions can be more easily and quickly determined for modern codes.

Chapter 2

Background

The sections which follow in this chapter will briefly cover any background material that is not explicitly explained in the contributory works presented in this thesis—Chapters 4–6. Although the following text is not exhaustive it will be sufficient for a reader to appreciate the technicalities of the research, and, where appropriate, references will be provided if more extensive literature is desired.

This chapter is organised as follows: Section 2.1 will define terms used in this thesis; Section 2.2 will discuss the difference between OPENMP and OPENCL, which are topics relevant to Chapter 4; Section 2.3 will similarly elaborate on techniques discussed in Chapter 5, i.e. common loop optimisations and iterative compilation; Section 2.4 will begin with the basics of supervised machine learning, move onto discussing active learning, briefly outline the Random Forest model, which is the only model used in these works not explained in the technical chapters themselves, and end by describing how quality has been measured for all classification and regression problems; Section 2.5 gives explanations of the statistical methods used in this research; and Section 2.6 summarises the chapter.

2.1 Terminology

The term *runtime* is used to refer to an actual measurement of the time it takes for a program to begin executing and to reach completion. The hyphenated term *run-time* is not a measurement but is used to indicate that an event occurs during program execution, just as learning-time refers to the period during which data is collected to generate a heuristic, or compile-time refers to the time period during which a program is compiled.

The phrase *machine learning*, when used in this thesis, relates exclusively to *supervised* machine learning, as opposed to the unsupervised or semi-supervised variants which are not discussed further here. The form of *active learning* used in these works is of the *pool-based* variety, as opposed to *stream-based active learning*: the difference being multiple candidates are ranked based on their estimated utility versus making a decision to label (or not) each potential candidate in a sequential stream.

The aim of supervised machine learning is to determine a mapping between a set of explanatory variables X (called the *feature vector*) and a dependent variable Y. A unique permutation of feature vector values $x \in X$ is called a *configuration*, or a *candidate training example* in the context of a configuration which is being considered for *labelling*; the process of labelling involves the measurement of the value $y \in Y$ which is associated with the feature values x. A tuple (x,y) is interchangeably referred to as a either a *training instance* or a *training example*. When Y is in a discrete space the problem of determining this mapping is called *classification*, whereas *regression* refers to the case where $Y \in \mathbb{R}$.

2.2 Parallel Programming Models

Chapter 4 uses a run-time optimisation decision to illustrate the merits of active learning over a passive approach. In particular, the exemplar heuristic is that when given a set of input values to a program, and two implementations of that program written in OPENMP and OPENCL, would it be faster to run that program's kernel on the CPU using the OPENMP programming model or on the GPU with OPENCL. Relevant to that work is a discussion of the different characteristics of OPENMP versus OPENCL, which are given in the following subsections.

2.2.1 Open Multi-Processing

Open Multi-Processing, or OPENMP for short, is a high-level, shared-memory parallel programming model for C, C++ and Fortran, which comprises compiler directives, library routines and environment variables [OpenMP, 2011].

In OPENMP a programmer identifies a *structured block* of code they wish to execute using either thread or data level parallelism with a compiler directive or comment for C/C++ or Fortran, respectively. The run-time library then distributes work or *tasks* from the master thread to some number of slave threads spread across multiple cores.

The run-time library is configured through compiler directives/comments, function calls, or environment variables on the system. Options include specifying the maximum number of threads; which variables are shared between threads and which are private; and how work is allocated: for example, should each thread have an equal number of tasks or should they be served in a FIFO (first-in first-out) queue.

An example of using OpenMP to parallelise a simple loop is given in Listing 2.1.

Listing 2.1: this is a simple OpenMP code example where the pragma directs a compatible compiler to generate parallel code that distributes the loop iterations across a multi-core processor, where variables `a` and `b` are shared but the loop counter `i` is private.

```
void simple(int n, float *a, float *b) {

    int i;

    #pragma omp parallel for
    for (i=1; i<n; i++) { // i is private by default
        b[i] = (a[i] + a[i-1]) / 2.0;
    }

}
```

2.2.2 Open Compute Language

The Open Compute Language (OpenCL) is a formalised standard designed to allow developers to write general-purpose, parallel code that can be executed on CPUs, GPUs, DSPs (Digital Signal Processors), FPGAs (Field-Programmable Gate Arrays), and other processors and hardware accelerator devices without alteration: although, as ever, performance is not portable. Despite its name, OpenCL is actually an entire framework for parallel computation which includes a language, based on C99; an API (Application Programmable Interface); libraries; and a run-time system [OpenCL, 2012]. In comparison to OpenMP, OpenCL is a much lower-level specification but its advantage is its functional portability between device types.

In order to run code on a parallel device supported by OpenCL a number of steps must to taken. First, an OpenCL *host* application connects to a *platform*, which is a vendor-specific OpenCL implementation: at the time of writing, there are numerous implementations provided by Intel, AMD, ARM, Qualcomm, and NVIDIA amongst others. This platform is a run-time system which allocates work to one or more compute *devices*, such as a CPU or GPU. After device objects have been created, via calls to the platform, a *context* is formed to keep track of the run-time objects. Next, an OPENCL program is generated from source code written in the OPENCL language. Specifically, this code is compiled at run-time and a *kernel* created by specifying which function in the source code gives the entry point for the desired computation, since numerous kernels can be present in the same source code. Lastly, buffers may need to be instantiated before the kernel is instructed to begin executing using a *command queue*. An example of a relatively simple OPENCL program is provided in Listing A in Appendix A.

In terms of the OPENCL execution model, one instance of a kernel runs on one *compute unit* to tackle one *work-item*, where one or more work-items are collected into one or more *work-groups*. To make this a little clearer, the OPENCL platform model is shown conceptually in Figure 2.1, where a host machine can connect to multiple devices simultaneously; each device contains one or more compute units; and each of those contains one or more processing elements. To give a real-world example, a multi-core CPU is a compute device where each of the cores are a compute unit and each contain a single processing element; in this scenario each core would execute all work-items in the work-groups that it is assigned.

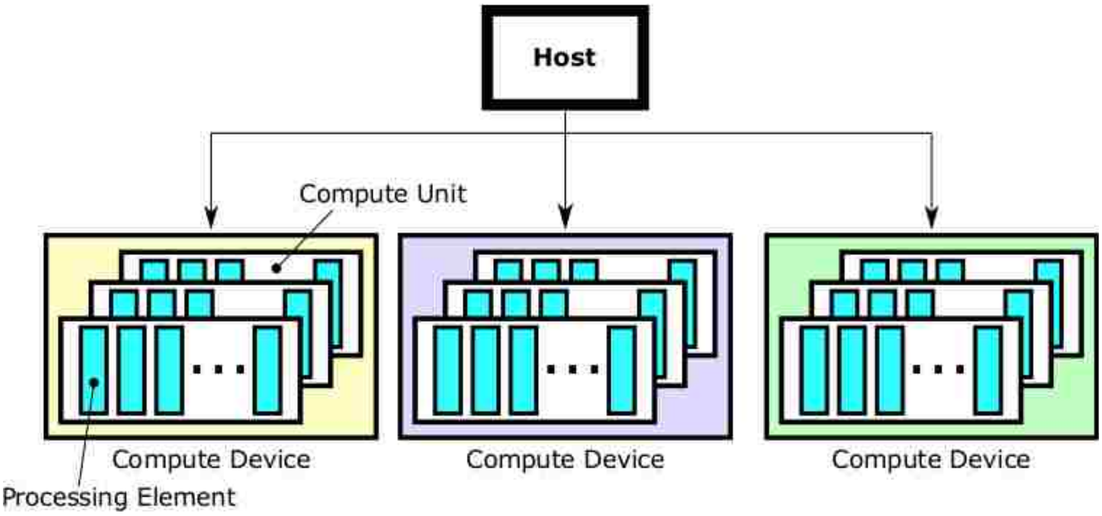


Figure 2.1: an illustration of the OPENCL platform model adapted from OpenCL [2012]: one host can have multiple compute devices, each of which contains one or more compute units that each hold one or more processing elements.

OpenCL also has its own conceptualised memory model illustrated in Figure 2.2. Each processing element in a compute unit has access to a *private memory* that only it can see. Across processing elements there is a *local memory* for shared data, where between compute units there lies a cache supplying information from the *global memory* and a memory specifically designated for constant values. As might be expected, it is often the case that as one traverses the memory hierarchy from private to global memories the latency for memory operations increases substantially.

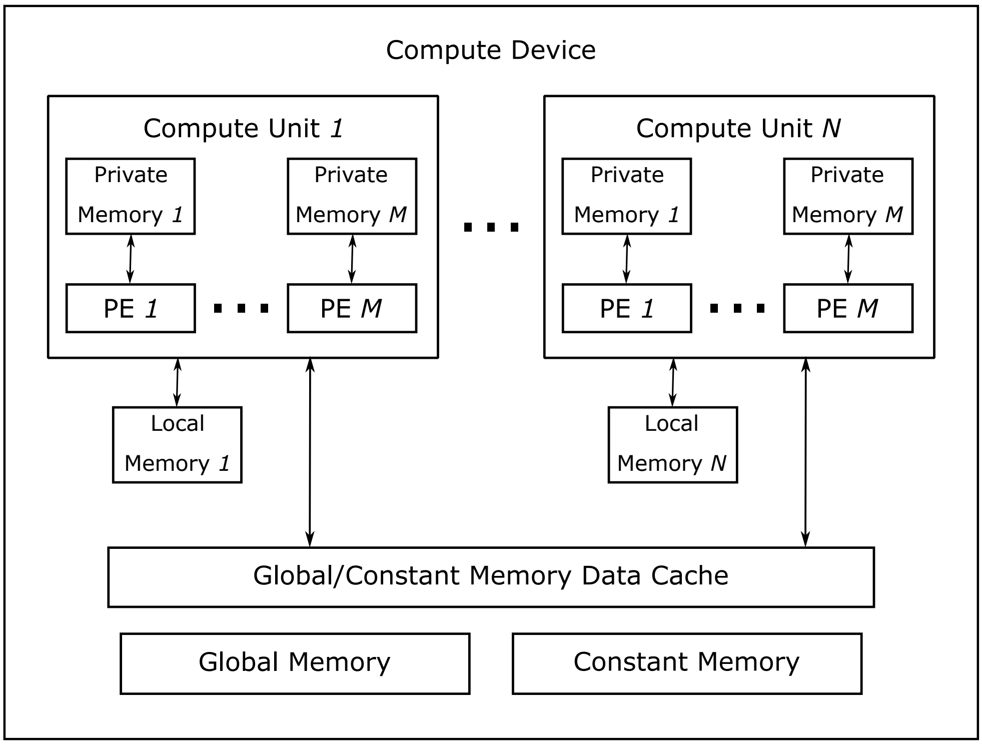


Figure 2.2: a conceptualisation of the OpenCL memory model, adapted from OpenCL [2012].

As with OpenMp, there is much more to the OpenCL standard than it would be sensible to include here so an interested reader should consult the relevant specification, where OpenCL v1.2 was used for the work in this thesis [OpenCL, 2012].

2.3 Compile-time Optimisation

This section briefly describes the loop optimisations that are referenced in Chapter 5 and then illustrates the concept of iterative compilation discussed in the same.

2.3.1 Loop Transformations

Loop Unrolling One of the most basic optimisation transformations one can apply to code is loop unrolling: sometimes called loop unwinding. The goal of loop unrolling is to reduce the overhead of index arithmetic and boolean checks whilst simultaneously increasing the potential to exploit ILP [Ullman and Aho, 1977]. Some disadvantages of the technique are an increase in code size as well as, potentially, an increase in the instruction cache miss rate and or register use. An example is provided in Listings 2.2–2.3, where the number of increments and boolean checks on `i` is cut by four-fifths.

Listing 2.2: before loop unrolling

```c
int i;
for (i=0; i<100; i++) {
    doSomething(i);
}
```

Listing 2.3: after loop unrolling

```c
int i;
for (i=0; i<100; i+=5) {
    doSomething(i);
    doSomething(i+1);
    doSomething(i+2);
    doSomething(i+3);
    doSomething(i+4);
}
```

Cache Tiling Cache tiling or strip mining [Loveman, 1976; Wolfe, 1987] is an optimisation whereby the iteration space of a loop is separated into blocks in order to increase the *locality of reference* [Denning, 2005] and potentially allow parallel execution of the blocks if the blocks hold memory references which are independent of each other [Wolfe, 1989]. Take Listing 2.4 as an example, showing a simple matrix-vector multiplication in C; if the value of `n` is relatively large and the cache size relatively small then the instruction given on Line 7 may induce cache misses. In contrast, the code in Listing 2.5, which uses 5×5 tiles, will necessarily fit more easily into a cache.

Listing 2.4: before cache tiling

```
int i, j;
int n = 50;
int x[50][50], y[50], z[50];
for (i=0; i<n; i++) {
    z[i] = 0;
    for (j=0; j<n; j++) {
      z[i] += x[i][j] * y[j];
    }
}
```

Listing 2.5: after cache tiling

```
int i, j, k, l;
int n = 50;
int x[50][50], y[50], z[50];
for (i=0; i<n; i+=5) {
    z[i] = 0;
    z[i+1] = 0;
    z[i+2] = 0;
    z[i+3] = 0;
    z[i+4] = 0;
    for (j=0; j<n; j+=5) {
        for (k=i; k<min(i+5, n); k++) {
            for (l=j; l<min(j+5, n); l++) {
                z[k] += x[k][l] * y[l];
            }
        }
    }
}
```

Register Tiling Register tiling, as the name suggests, is very similar to cache tiling in that a loop is divided into blocks or tiles in such a way as to fit all the data being accessed into registers or a level in the cache hierarchy, respectively. These optimisations reduce the latency for stores and loads and so speed-up code execution, where

1. original code

```
do I = 1,N
  do J = 1,N
    do K = 1,N
      loop body
enddo
```

2. permuted

```
do J = 1,N
  do K = 1,N
    do I = 1,N
      loop body
enddo
```

3. tiled J

```
do JJ = 1,N,B_JJ
  do J = JJ,JJ+B_JJ-1
    do K = 1,N
      do I = 1,N
        loop body
enddo
```

4. permuted

```
do JJ = 1,N,B_JJ
  do K = 1,N
    do J = JJ,JJ+B_JJ-1
      do I = 1,N
        loop body
enddo
```

5. tiled K

```
do JJ = 1,N,B_JJ
  do KK = 1,N,B_KK
    do K = KK,KK+B_KK-1
      do J = JJ,JJ+B_JJ-1
        do I = 1,N
          loop body
enddo
```

6. permuted

```
do JJ = 1,N,B_JJ
  do KK = 1,N,B_KK
    do I = 1,N
      do J = JJ,JJ+B_JJ-1
        do K = KK,KK+B_KK-1
          loop body
enddo
```

7. fully unrolled

```
do JJ = 1,N,B_JJ
  do KK = 1,N,B_KK
    do I = 1,N
      C(I,JJ) = C(I,JJ) + A(I,KK) * B(KK,JJ)
      C(I,JJ) = C(I,JJ) + A(I,KK+1) * B(KK+1,JJ)
      C(I,JJ+1) = C(I,JJ+1) + A(I,KK) * B(KK,JJ+1)
      C(I,JJ+1) = C(I,JJ+1) + A(I,KK+1) * B(KK+1,JJ+1)
enddo
```

8. scalar replacement

```
do JJ = 1,N,B_JJ
  do KK = 1,N,B_KK
    RR1 = B(KK,JJ)
    RR2 = B(KK+1,JJ)
    RR3 = B(KK,JJ+1)
    RR4 = B(KK+1,JJ+1)
    do I = 1,N
      C(I,JJ) = C(I,JJ) + A(I,KK) * RR1
      C(I,JJ) = C(I,JJ) + A(I,KK+1) * RR2
      C(I,JJ+1) = C(I,JJ+1) + A(I,KK) * RR3
      C(I,JJ+1) = C(I,JJ+1) + A(I,KK+1) * RR4
enddo
```

Figure 2.3: adapted from [Jiménez et al., 2002], this figure illustrates the steps in conventional register tiling for a matrix multiplication code, where the block size is 2×2.

register tiling adds a few steps beyond those performed by cache tiling. Firstly, after cache tiling has been applied the inner-most loop must be fully unrolled since registers cannot be addressed using offset addresses. Secondly, *scalar replacement* can be used to further save on load and store operations between iterations [Callahan et al., 1988, 1990; Duesterwald et al., 1993; Carr and Kennedy, 1994]. A concise example adapted from Jiménez et al. [2002] is given in Figure 2.3, where loop permutations or interchange [Wolf, 1992] are used to manipulate the order of the loops to tile each in turn.

2.3.2 Iterative Compilation

Iterative or profile-driven compilation is a technique that can be used to find a good transformation or a good set of transformations to optimise a given code [Aarts et al., 1997; Bodin et al., 1998; Cooper et al., 1999; Kisuki et al., 2000; Knijnenburg et al., 2002]. The process is simple, a driver program is fed the source code and a list of

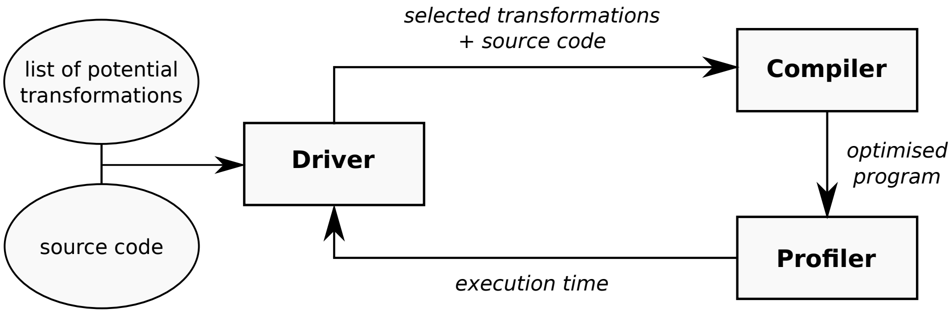


Figure 2.4: the three main stages in iterative compilation

potential optimising transformations that could be used to optimise it. Some search is performed and one or more transformations are selected to be applied. After the code has been optimised, compiled and linked the final executable is profiled to determine the performance of those transformations. This information is then fed back into the driver program and this loop continues until some completion criterion is satisfied, with the best transformation(s) out of those tested being used to optimise production code. This process is illustrated in Figure 2.4.

2.4 Supervised Machine Learning

Supervised machine learning is the process of attempting to infer a relationship when given some labelled training data. Which is to say, when given a set of training examples $(x_1,y_1),(x_2,y_2),(x_3,y_3),\ldots,(x_n,y_n)$, where x_i is a vector of explanatory variables or features which identifies example i of n, and y_i is the associated label, the task of supervised learning is to work out how to map the input-space x to the output-space y. In this way, when presented with a previously unseen feature vector x_{n+1} the technique is able to predict its label y_{n+1}. When the label is some continuous value this is called regression, as opposed to classification where the potential values of y are from some discrete set.

2.4.1 Active Learning

> Not hearing is not as good as hearing, hearing is not as good as seeing, seeing is not as good as mentally knowing, mentally knowing is not as good as acting; true learning continues up to the point that action comes forth
>
> —Xun Zi, *The Teachings of the Ru*, 3rd Century BC

The term active learning first appeared in the context of pedagogy, where it has been shown that students who *actively* participate in the learning process fair better than those who are *passive* observers [Harke, 1998; Prince, 2004; Michael, 2006; Hoellwarth and Moelter, 2011; PCAST, 2012]. This concept is unsurprisingly ancient, evidenced by the quote from Xun Zi—written sometime in the third century BC—but has recently been rediscovered, popularised, and applied to the modern classroom. In a similar fashion, an active machine learning algorithm asks queries of an *oracle* or a teacher, choosing what it wishes to learn next based on what it already knows about the space, thereby actively engaging in the learning process as opposed to merely receiving information. In this way, the chance of redundant examples being learnt is reduced and, moreover, so is the cumulative learning time [Settles, 2013].

Figure 2.5 provides an overview of the key steps involved in all active machine learning algorithms:

1. some number of random but distinct configurations are chosen to be labelled by the oracle, and when this is done these constitute an initial training set;

2. an intermediate model is built using the training set thus far accumulated;

3. completion criterion are checked, which can involve, for example, an estimate of the intermediate model's quality and or the total learning cost which has accumulated,

   (a) if completion is reached then the final model is taken as the last intermediate model,

   (b) otherwise new training data is sought;

4. from amongst one or more candidate training examples that could be chosen to be learnt from next one or more of these are selected to be labelled by the oracle based on an estimate of how much new information they might provide;

5. and the loop starts again at step 2.

2.4.2 Random Forest

Random Forest [Breiman, 2001] is a type of ensemble learning algorithm which leverages the concept of bootstrap aggregating, more commonly referred to as *bagging* in

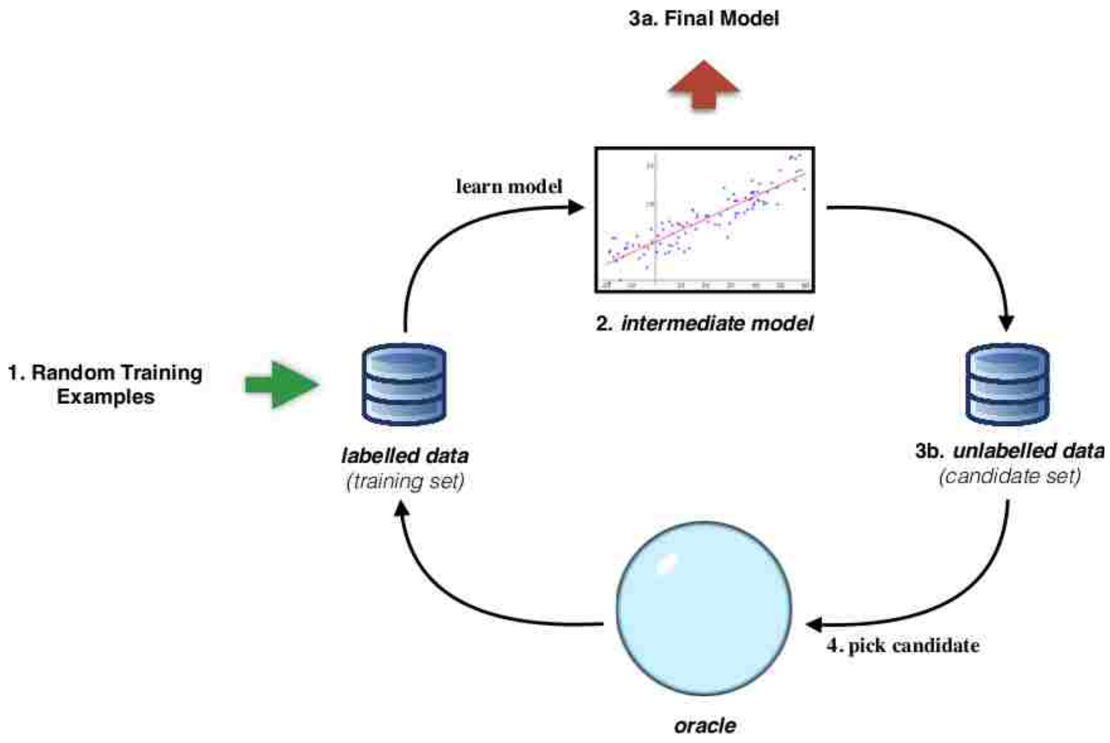


Figure 2.5: a generic view of active learning

the literature, to ameliorate the tendency of tree learners to *overfit* [Hastie et al., 2009] to their training data, see Breiman et al. [1984] for details.

In bagging, m randomly generated training sets are created from an original training set of n instances by uniformly sampling from that original set with replacement, such that each new set also contains n instances. In regression the output value of a test instance is then given as the average predicted output value of the m models, whereas a classification problem uses majority vote instead—see Figure 2.6.

In the canonical Random Forest algorithm the m models are tree-based, but each model differs slightly from the typical regression or decision tree since at each potential splitting point in the tree a random subset of features are considered instead of all of them. This is done in an attempt to avoid correlation among the m trees, particularly when some small subset of features have relatively high predictive power.

2.4.3 Evaluating Quality

The following paragraphs will explain how the various metrics used for evaluation throughout this thesis are calculated, firstly for classification and then for regression.

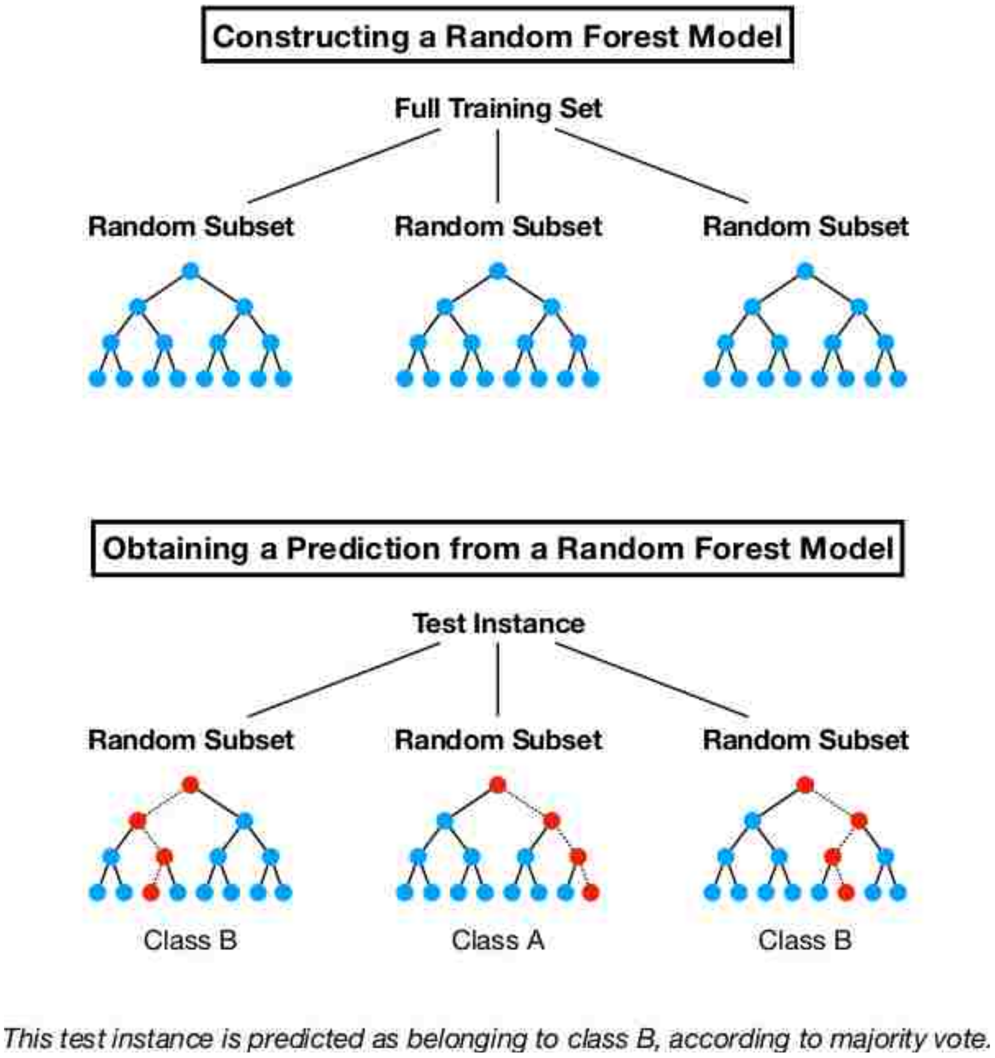
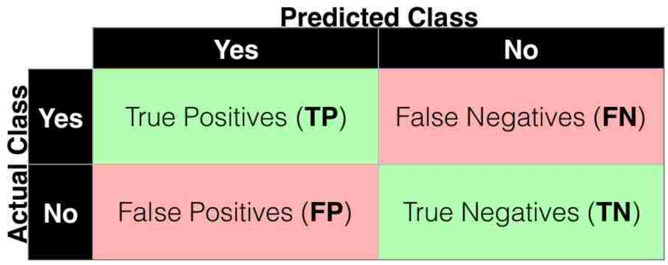

Constructing a Random Forest Model
Full Training Set
Random Subset
Random Subset
Random Subset
Obtaining a Prediction from a Random Forest Model
Test Instance
Random Subset
Random Subset
Random Subset
Class B
Class A
Class B
This test instance is predicted as belonging to class B, according to majority vote.

Figure 2.6: illustrates the construction and use of a random forest model



		Predicted Class	
		Yes	No
Actual Class	Yes	True Positives (**TP**)	False Negatives (**FN**)
	No	False Positives (**FP**)	True Negatives (**TN**)

Figure 2.7: a confusion matrix for a binary classification problem

Accuracy in Classification Probably the easiest metric to understand with regards to classification models is accuracy, it is simply the number of correctly classified test instances relative to the total number of test instances, reported as a percentage. In relation to the *Confusion Matrix* [Stehman, 1997] in Figure 2.7, the calculation is given explicitly in Equation 2.1.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{2.1}$$

Cohen's Kappa versus Accuracy Accuracy is a trivial metric to understand and to calculate, but it does have one significant flaw. If there is a relatively extreme imbalance in the number of instances in the space belonging to one class over another then a high level of accuracy does not necessarily give a good indication of a high-quality model. For example, for the majority of different workload sizes evaluated for the SRAD kernel, discussed further in Chapter 4, the GPU was deemed the best device—see Figure 2.8. Since test instances are fetched at random and the majority of the space belongs to the GPU class any model which only ever predicts GPU will likely have a high level of accuracy, incorrectly giving the impression that the heuristic is able to accurately determine the optimal class for any instance in the space. To get a fairer picture of a model's quality Cohen's Kappa is sometimes used to calculate how good a model is at predictions, taking into consideration the probability of guessing right by chance [Cohen, 1960]. Cohen's Kappa is defined in Equation 2.2, where p_o is equivalent to accuracy as previously defined, and p_e is the overall probability of guessing a test instance's classification correctly based solely on the prevalence of that class in the space, and is defined in Equation 2.3 with reference to Figure 2.7. The correlation of kappa value to accuracy is demonstrated visually in Figure 2.9, where this graph was generated through simulation.

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{2.2}$$

$$\begin{aligned} p_{\text{yes}} &= \frac{\text{TP} + \text{FN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \cdot \frac{\text{TP} + \text{FP}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \\ p_{\text{no}} &= \frac{\text{FP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \cdot \frac{\text{FN} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \\ p_e &= p_{\text{yes}} + p_{\text{no}} \end{aligned} \tag{2.3}$$

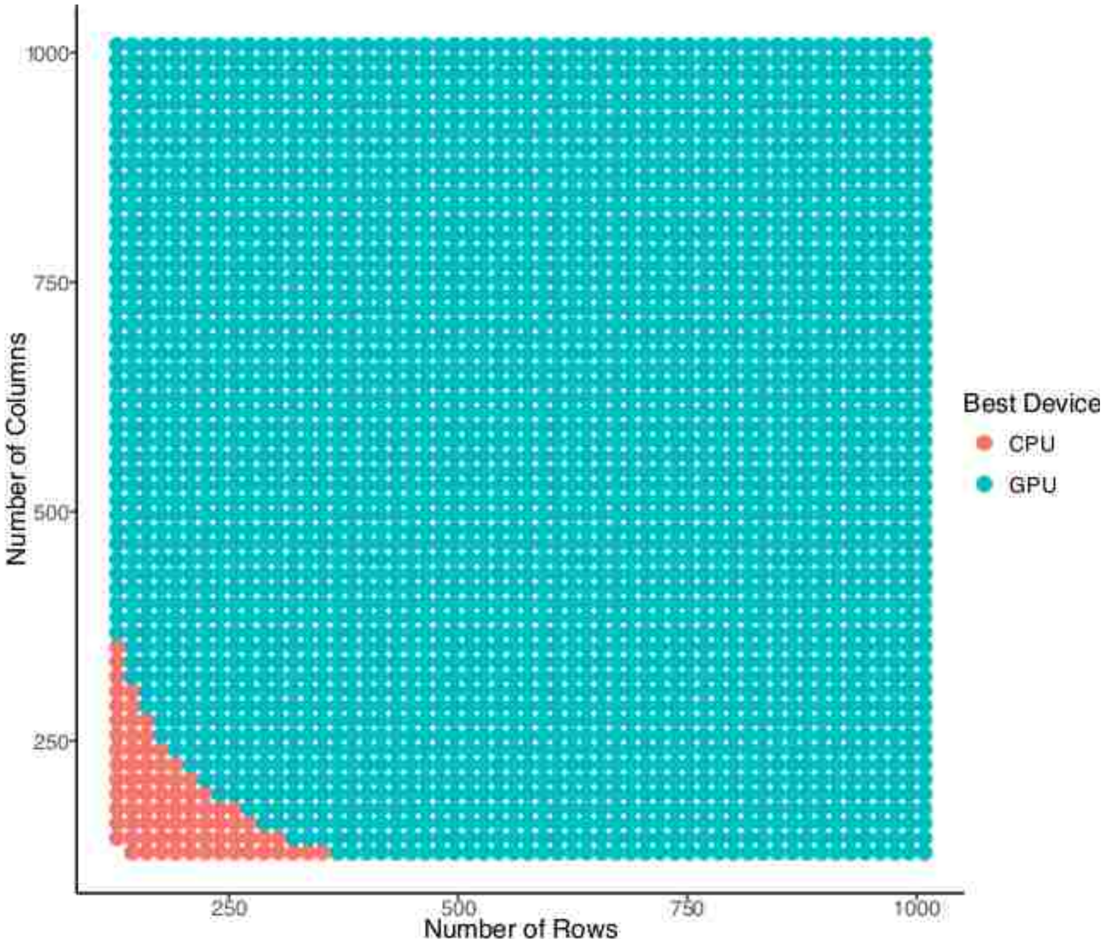


Figure 2.8: the SRAD feature-space, discussed more in Chapter 4, is a good example of one in which the distribution of classes is heavily skewed; this presents a problem for classification accuracy since a simplistic model, which only ever predicts one class, can score a high accuracy rate even though it does not fully reflect the underlying data.

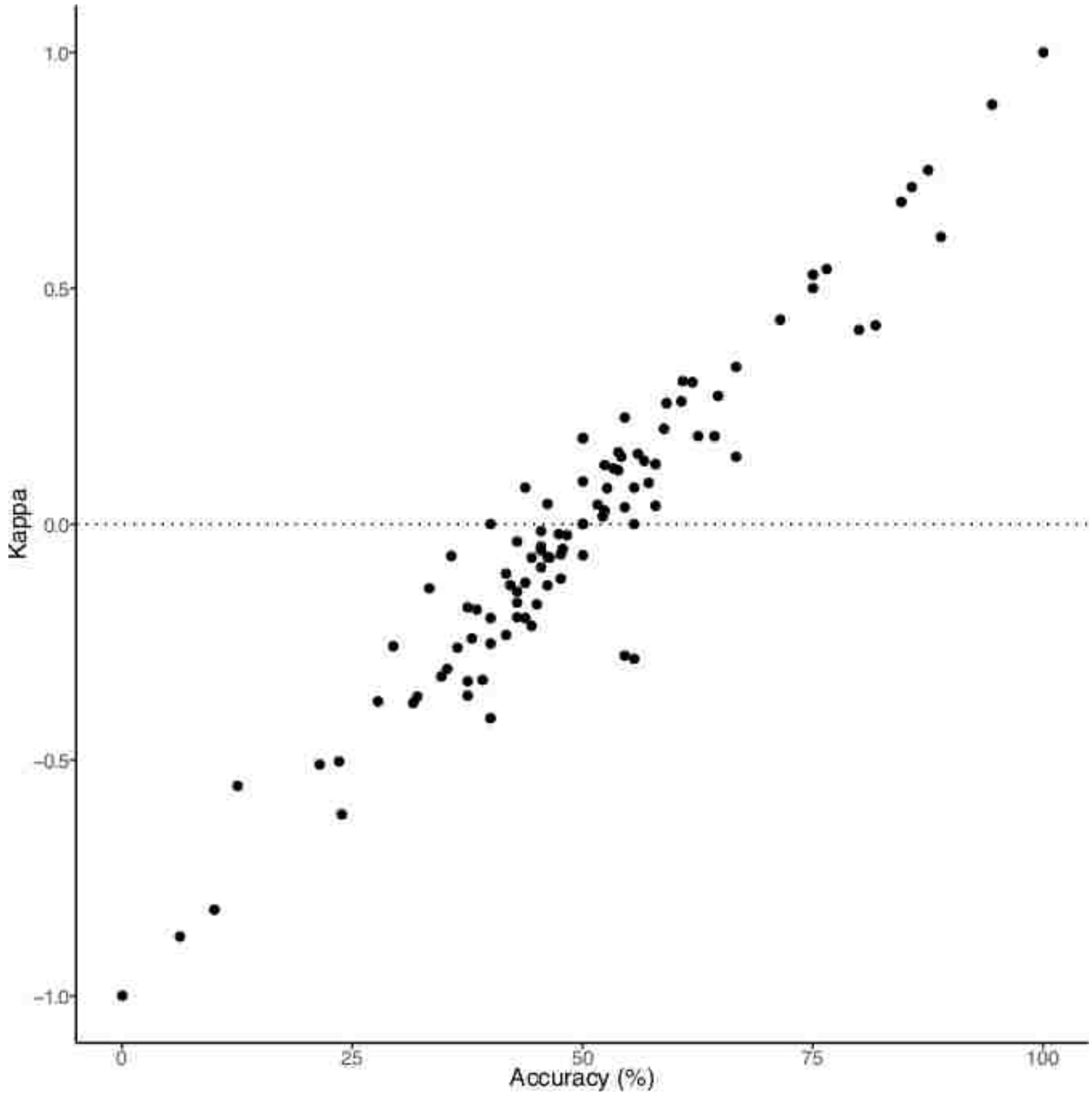


Figure 2.9: these data were generated through simulation, where the values for TP, TN, FP and FN were randomly generated, and illustrates a correlation between kappa and accuracy values.

Mean Absolute Error In regression, since output values $y \in \mathbb{R}$, accuracy and kappa cannot be used to evaluate performance; instead, Mean Absolute Error (MAE) is often used and is given as the absolute difference between the predicted value $\hat{y}_i$ and the actual value y_i of each test instance averaged across the entire test set—see Equation 2.4.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |\hat{y}_i - y_i| \tag{2.4}$$

Root-Mean-Squared Error The Root-Mean-Squared Error (RMSE) of a prediction model is defined as the value predicted by that model $\hat{y}_i$ compared to the observed mean value y_i for n test instances, as follows:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{n}} \tag{2.5}$$

2.5 Statistical Techniques

There are three statistical techniques used in these works which are not explicitly defined in the technical chapters, instead these will be explained in the following subsections.

2.5.1 Confidence Intervals

Confidence Intervals (CI) are used in statistics to give an indication of the uncertainty associated with an estimate of a population metric based on a number of samples [Neyman, 1937]. CI are used in these works to give a sense of the confidence one should have in the estimate of the population or true runtime mean, aggregated from sample program runtimes.

Where the population standard deviation σ is unknown and the number of samples n is less than 30, which is the case for all experiments in this thesis where CI has been calculated, the CI is defined as the range given by Equation 2.6, where $\overline{X}$ is the sample mean and s is the sample standard deviation. The *critical value* t^* is taken from Student's t-distribution, a table of pre-computed critical values parametrised by degrees of freedom $r = n - 1$ and $\alpha = \frac{1}{2}(1 - C)$, where C is the confidence level.

$$\left[\overline{X} - t^* \frac{s}{\sqrt{n}}, \overline{X} + t^* \frac{s}{\sqrt{n}}\right] \tag{2.6}$$

It is important to note that a given percentage confidence interval does not signify that the population or true mean lies within the given range but rather that upon repeated experimentation the population mean will lie between the calculated confidence intervals some percentage of the time. For example, a 95% confidence level provides the ability to say that one is 95% *confident* that the population value lies within the stated interval.

2.5.2 Outlier Removal

It is often common practice in machine learning to remove samples from data which appear to have extreme feature values relative to the bulk of the samples collected. These are called outliers, and this is necessary because some machine learning algorithms (such as Linear Regression) are very sensitive to their influence. Which is to say, a single outlier can cause the whole model to skew wildly in one direction, and if the instance that this outlier represents is atypical for the data then the resulting model will have poor overall performance.

Unfortunately, there is no theoretical definition of what constitutes an outlier when given a set of samples so a heuristic is typically used instead to distinguish and eliminate them from consideration. In this thesis, and in particular the work presented in Chapter 4, the method of outlier removal used is based on *Tukey's Fences* [Tukey, 1977] and his definition of *far out* observations. This rule of thumb relies upon the calculation of the InterQuartile-Range (IQR), and states that any samples with values out-with the interval $[Q_1 - 1.5\,(Q_3 - Q_1)\,, Q_3 + 1.5\,(Q_3 - Q_1)]$ can be considered an outlier, where Q_1 and Q_3 are the first and third quartiles of the distribution, respectively, and $Q_3 - Q_1$ is the IQR.

2.5.3 Welch's T-test

In the context of this thesis, a t-test has been used to determine if two sample means can be said to be statistically different from one another. This is useful since it can allow one to say, with some level of confidence, that there is actually a significant difference in the runtimes of an executable when one optimisation strategy is employed over another, where this is not necessarily obvious from looking at the raw data.

Since the runtime means being compared come from independent distributions there is no natural way in which to pair observations, and since each set of measurements may have different sample counts the unpaired two-sample t-test variant is used.

Furthermore, since the variance is not assumed to be constant for both distributions Welch's t-test is used [Welch, 1947], as opposed to Student's original test [Student, 1908]. It is assumed, and a requirement of these tests, that runtime samples are *independently and identically distributed* and, more specifically, that these are taken from Gaussian distributions.

To determine if the two sets of runtimes are significantly different the t-statistic is first calculated using Equation 2.7, where $\overline{X_i}$ is the sample mean, s_i the sample standard deviation, and n_i the number of samples from distribution i of two.

$$t = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{2.7}$$

Once the t-statistic has been calculated the degrees of freedom r is worked out using the Welch–Satterthwaite Equation (2.8) [Welch, 1947; Satterthwaite, 1946]. By consulting the probability density function of the t-distribution with degrees of freedom r the probability of obtaining the calculated t value can be obtained, and if this is lower than the alpha value, where C is 95% for example, then one can say that there is a 95% probability that these these two means come from different distributions.

$$r = \frac{\left(s_1^2/n_1 + s_2^2/n_2\right)^2}{\frac{s_1^2/n_1}{n_1-1} + \frac{s_1^2/n_1}{n_2-1}} \tag{2.8}$$

2.6 Summary

This chapter has provided brief details of the alternative optimisation strategies chosen during run-time or compile-time in the heuristics generated for the works in Chapters 4–5, respectively. It has also discussed both supervised machine learning and active learning, the essential constituents of iterative compilation, how Random Forest models are generated, and how models in this thesis are evaluated using various metrics and statistical methods. The chapter which follows will provide a literature review relevant to this research topic.

Chapter 3

Literature Review

This chapter provides a brief but comprehensive review of all academic literature relevant to the topic on which the works in this thesis are based; that is to say, this chapter summarises all publications in which the authors attempt a solution which automatically selects a good optimisation strategy from a complex space *and* where they have tried to address the long search times associated with such problems. In broadly chronological order, this chapter looks at the various techniques that have been attempted in the past, including, but not limited to, active learning, and where appropriate makes comparisons between these related works and the contributions presented in this document.

Wolf et al. [1996] appear to have been the first to construct a compiler which both searches for a good optimisation strategy automatically for a code and performs a pruning of the optimisation-space in order to accelerate the search. In particular, the authors leverage the independence of the transformations being evaluated to greatly reduce the potential number of combinations that need to be attempted while looking for good configurations. Unfortunately, this early technique is heavily tied to the hardware for which it targets its optimisations, both in terms of the details needed by the heuristics to limit the search space (e.g. the number of hardware registers) and the relatively detailed and bespoke processor model used for estimating the number of cycles per loop iteration that would result from a given set of transformations. Therefore, this approach is brittle and porting across architectures is relatively laborious when compared to later machine learning based works.

Kisuki et al. [1999] provided a more portable solution to auto-tuning than the previous work in that they used actual runtimes of program executions, as opposed to estimates from a complex simulator, to base their predictions of performance upon.

More specifically, in their paper Kisuki et al. [1999] attempted to ameliorate the increase in compile time that often results from auto-tuning by suggesting three search plans that could be used to cover the optimisation-space—random, gradient, and grid-based methods. Ultimately, concentrating on the latter of the three as the most efficient, their algorithm begins by constructing an evenly-spaced, but coarse, grid over the entire optimisation-space; the corresponding optimisation strategies at each point on this grid are used during compilation and profiling for a given application; and from this information the point with the lowest runtime, together with any points within some threshold of this minimum, are added to a priority queue. For each point in the queue, if the performance of that point is still within the threshold of the minimum program runtime seen thus far, the grid around that point is refined, such that the spacing in each dimension around the point is half its previous value. Again, these configurations (points) are used during compilation and the resulting binaries profiled, and again any new points found to be within the threshold of the current minimum runtime are added to the priority queue also. This process continues until no more points in the queue are within the threshold and the best configuration found in this search returned as a good optimisation strategy. This work is simple and easy to understand, but is flawed in that the search pattern is rigid and could easily skip over global minima, even more so than is typical for non-exhaustive search, or result in an excessive number of evaluations if the spacing between the grid points is not optimal. In comparison, active learning is an adaptive approach which does not need to search equally along all dimensions since it is capable of estimating uncertainty in its knowledge about the space to a finer granularity, and in this way it should usually be more efficient.

Active Harmony is an automated run-time tuning system which defines an API that allows library writers to permit profiling of CPU and memory usage of their code at run-time, together with a server application; it is also the first occurrence in the systems optimisation literature where an active learning technique has been applied to a search problem, although that term never actually appears in the relevant texts [Ţăpuş et al., 2002; Chung and Hollingsworth, 2004]. The intuition behind the system is that since different libraries and or algorithms are more performant when used with different applications it may be beneficial if one could create libraries which held multiple versions of a procedure; the idea being that at run-time one variant of the required function could then be adaptively chosen over another based on previously recorded performance data. In a later work, Tiwari et al. [2009] combined Active Harmony with the CHiLL framework [Chen et al., 2008], a polyhedral transformation and code generation engine, in order to power iterative compilation. In particular, Active Harmony requests multiple codes be generated by CHiLL using specific transformations; these variants are executed and profiled in parallel; and the performance data fed back into Active Harmony to move a simplex within the search space. This iterative process will then ultimately converge to a solution, but as with as with the earlier incarnation this will not necessarily be a good solution since the method makes an assumption of monotonicity in the space as the simplex method requires it, whereas it has been shown that optimisation problems are often non-convex—e.g. Cooper et al. [2002] and Fursin et al. [2002].

In contrast to the techniques discussed so far, Triantafyllis et al. [2003] attempted to reduce the time taken to explore an optimisation-space by limiting the transformations being considered to those that were presumed to be the most interesting and beneficial, in other words by manually and artificially reducing the size of the search space. This was acomplished by first eliminating those transformations from contention that the researchers believed to be generally well-tuned already, those that consistently degraded program performance, and optimisations that seemed to be too similar to those already selected to be included in their limited subspace. In order to search over the given optimisation-space they built a tree by selecting a set of m most relevant transformations O for all code segments C in a particular program and use these as the children of the root node. For each child node $o_i \in O$ a set of transformed code segments C_i are chosen whose performance is estimated to be maximised by those particular transformations, relative to the other segments in the program. Using this information, the successors of the o_i node are those most relevant or important optimisations for the C_i segments. This process continues until only leaf nodes remain. The final optimisation strategy that is selected to be used is the best performing path through the constructed tree, where a breadth-first search is used so that sub-trees whose estimated performance is already too high with respect to other nodes can be eliminated from further consideration. The estimates of the importance of optimisations to code segments are provided by a model based on profiled data, as in Li et al. [1999]. In order to prune the tree the initial code segments present in C are those which contain hot code only—i.e. code that is frequently executed. Despite initially appearing an elegant proposition to shrinking optimisation search spaces there are a number of weaknesses to this approach. Firstly, construction and search through this optimisation tree occurs at compile-time which greatly increases the effort required to produce any code, whereas active learning can be learnt off-line; secondly, the authors assume that the space of useful transformations can be easily reduced to a small subset *a priori* which cannot be trivially achieved; moreover, the authors acknowledge that this process would need to be repeated per processing device (and thus necessarily involves much manual effort for each processor being targeted) where in this thesis we wish to have a portable and fast approach to auto-tuning. Lastly, it has been shown that 'dangerous strategies' can actually substantially increase performance under certain conditions so eliminating them so casually would seem to be unnecessarily short-sighted—e.g. `-funroll-all-loops` [Ashouri et al., 2016].

Pan and Eigenmann [2004] were the first researchers to suggest that a program's source code could be chopped up into smaller *Tunable Sections* (TS) in order to speed-up auto-tuning. Moreover, they proposed that the performance of an optimisation strategy when used on a TS can be compared even when the workloads or *contexts* differ by using one of three systems—context, model, or re-execution based ratings. More specifically, in context-based ratings performance results can be directly compared when different invocations of a TS use the same workload, however, in order to determine if this is the case when transforming code on-line the TS needs to contain variables which are scalar only: either plain scalar variables, array references with constant subscripts, or memory references with pointers that are fixed throughout the execution of the TS. Naturally, with such constraints this comparison cannot be applied often, limiting its utility. In re-execution based rating the TS is forced to execute again with the same workload which allows for performance measurements to be compared fairly and directly under differing optimisation strategies, but again this method is limited in its applicability in that the TS cannot be executed in this loop-like way, and the measurements remain comparable, unless side effects are prohibited. Which is to say, any library calls with side effects, such as `malloc`, `rand` or I/O functions, prohibit the use of re-execution based rating. Finally, model-based rating can be used to formulate a linear regression between input and performance achieved by a given strategy, but this only applies to relatively small benchmarks since each TS requires a large number of invocations for the model to have good accuracy, and the more TSs in a program the more substantial the tuning time will be.

Fursin et al. [2004] offer a novel idea to combat the long search times in iterative compilation, by providing a means to estimate when one should stop searching for another optimisation to try. In particular, the researchers proposed a fast method to estimate the lower bound on the execution time of a scientific code by timing a program when nearly all cache misses are removed. This can then be used to stop the search when an optimised binary is within some percentage of this lower bound. To calculate an approximation of the lower-bound runtime they record the memory addresses of all load or store operations in subsections of code during a normal program run. Each subsection was then modified such that the memory operations in each loop within that subsection point to pre-set scalar values, whilst checking and maintaining data dependencies. In this way, the time for memory referencing is reduced and so is the cache miss rate. Unfortunately, this lower bound estimate is only accurate for memory bound applications and so cannot be universally applied. Furthermore, it is often difficult to determine data dependencies as this can often be influenced by run-time behaviour. Nevertheless, this is an interesting piece of research but is somewhat orthogonal to the works in this thesis since it deals with stopping criterion not optimising the body of the search algorithm itself.

Zhao et al. [2005] tackle the high cost of applying optimisations and experimentally observing the results by constructing analytical models and using these to estimate the profitability of such optimisations being succesful without requiring further profiling. This is quite similar to the work by Wolf et al. [1996], except that in that case the authors used simulation as opposed to analytical models to predict performance. That said, the disadvantages are the same. Zhao et al. [2005] use code models generated by the optimiser together with models of the optimisations being applied and models of the resources of a particular target machine in order to establish a performance estimate. These latter two models need to be constructed by an engineer manually which makes the approach initially very expensive and means these models are not easily ported between architectures. By contrast, machine learning derived models can be constructed in such a way as to avoid processor peculiarities and permit performance prediction to be portable.

The ACME compiler was presented by Cooper et al. [2005] as the first user-friendly iterative compilation enabled system, in that it allowed the user to easily select between four distinct algorithms that could automatically search the space of optimising transformations for a good strategy: these were a greedy constructive, a genetic, a hill climbing, and a random-probing search algorithm. Arguably more interesting, these search strategies were accelerated by the use of *Estimated Virtual Execution*, a technique which predicts the performance effect of optimisations based on counts of instructions executed by a single program run. In particular, in cases where the Control Flow Graph (CFG) is not changed by an optimisation it is possible to record each basic block's execution frequency in an unoptimised code profile run and predict the performance of the optimisation strategy by summing over all blocks the frequency of their executions multiplied by the number of instructions within those blocks after optimisation. To deal with cases where the CFG will change the ACME compiler runs the unoptimised code version and associates a *block-identifying tag* with each block whose frequency it counts. For every potential CFG changing optimisation a pass is made which detects changes in that CFG and updates the frequency counts as needed. The authors acknowledged that this technique is expensive, only works for a subset of optimisations, and often fails for applications with complex control flow, making it of limited use in practice despite being an elegant technique.

Epshteyn et al. [2005] also suggested an active learning like strategy in their work, although they refer to it as *active sampling*, and it is more of an analytical model than machine learning proper. In their algorithm, they use information contained in analytical models derived by Yotov et al. [2003, 2005] to seed their model with good information, and from that data use a potential field to find likely informative candidates to explore next, that are also not too expensive. They achieve this by placing a positive (attractive) charge on the origin, close to where the faster executing binaries will be located, as well as any points they estimate have good performance; negative (repulsive) charges are placed on already explored points, and they find the minimum potential located in this field to sample from next. This compares favourably to the alternative of using the near-exhaustive search that was already built into the ATLAS BLAS library generator [Whaley et al., 2005]. In terms of comparing this research to the works in this thesis it remains to be seen whether this technique would beat a passive, random search strategy in the small optimisation-space in which it has been applied. Moreover, since it is not machine learning based but analytical in nature it is tied to an architecture and not easily portable. Furthermore, no attempt is made to minimise the number of samples that should be taken to avoid noise for each configuration tested in the optimisation-space like in Chapter 5.

In Franke et al. [2005] the authors recognised that there are distinct benefits to selecting either an iteratively-improving, focussed search versus a broadly random exploration of the space. Indeed, this is the same exploitation versus exploration problem discussed in Chapter 5–6. Namely, that exploitation or focussed search can fall into local minima, whereas random exploration will eventually find the global minima but at a potentially greater cost. In order to take advantage of both techniques Franke et al. [2005] used a genetic algorithm called *Population-based Incremental Learning* (PBIL) [Beluja, 1994] in competition with a random search strategy. The PBIL algorithm selects transformations stochastically, but with the probability distribution biased towards those optimisations which have generally improved performance in previous evaluated configurations. Compared to the the active learning strategies discussed in this thesis the approach outlined in this paper necessarily involves much more evaluations of candidate points, since not only are they performing full random search alongside PBIL but they also rely on stochastic processes to drive the PBIL genetic algorithm itself—hence, there is a higher chance of redundant data.

Unlike Triantafyllis et al. [2003], Haneda et al. [2005] attempted to reduce the search space of possible optimisations which need to be evaluated not by eliminating dangerous optimisations but by concentrating on those transformations which interact positively with each other in aggregate across a range of supposedly representative programs. To achieve this they take advantage of *Orthogonal Arrays* [Rao, 1947] (OA), where an OA is an $m \times n$ array which can be used to enumerate all possible ways that m-optimisations can interact with each other in pairs of two. Using this method one can work out how well each pair of optimisations complement one another whilst only considering a smaller region of the overall space. This is certainly an interesting solution to the large optimisation-space problem, but is limited in that it only looks at pair-wise interactions, whereas optimising transformations are known to interact in very complex and not necessarily intuitive ways that may be k-tuple, where $k > 2$. Active learning approaches, on the other hand, can deal with the whole optimisation-space in an efficient way without artificially limiting the number of possible interactions.

Similar to Pan and Eigenmann [2004], Fursin et al. [2005] also suggested that one could accelerate iterative compilation by simply chopping up a program into smaller, quicker-running constituent pieces, however, in their implementation they look for *phases* of consistent performance in loop nests or function executions, in terms of instructions per cycle counts found using the Performance API (PAPI) library [Mucci et al., 1999]. Once these regions have been identified and entered into either the original unaltered code is run to check that the phase is still stable or a new version which has been optimised differently is executed. In this way their technique can try different strategies during run-time whilst ensuring that the performance measurements are always comparable between the unoptimised and optimised code variants. Disadvantages of this approach are that a good optimisation for one program cannot be applied to another one, and thus this is a non-portable, application-specific technique. Furthermore, not only is this non-portable but it also only works at run-time. Active learning strategies, on the other hand, can be trained off-line and use portable-friendly features.

In an interesting work, Cavazos et al. [2006] developed a method called *reaction-based modelling*, where an artificial neural network is provided with training examples which have inputs of 4 carefully selected *canonical transformations* together with randomly picked optimisations, with the output being the speed-up achieved by the optimised code relative to a baseline. These canonical transformations are selected based on a calculation of highest entropy compared to their peers and, therefore, the transformations deemed canonical changes as data is incrementally collected. The intuition behind this approach is that codes which react similarly to a small set of canonical transformations must behave in a similar way and, thus, good optimisation strategies can be found by looking for similarly behaving programs—since not all transformations are tried the search is accelerated. Unlike the implementations of active learning in this thesis, however, the optimisation strategy chosen to be applied in each training instance is still heavily dominated by a stochastic process, meaning code executions may be performed which do not contribute useful information used to increase the quality of the resultant final model.

In Agakov et al. [2006] the authors proposed using machine learning to reduce the size of an optimisation-space which needs to be searched over by trying to find those regions within it which are particularly profitable for some characterisation of a given program. In order to accomplish this objective the researchers first construct a probability distribution for each of their training benchmarks, where the distribution gives the probability of each of the evaluated transformations appearing in good transformation sequences of some fixed length; there are 14 transformations attempted for each benchmark, and the length of the sequences are limited to five, thus, the authors perform 14^5 exhaustive evaluations (profile runs) per benchmark. When a new benchmark is encountered static code features are put through *Principle Components Analysis* (PCA) [Jolliffe, 2002] which reduces their number from 36 to 5. The closest learnt probability distribution is determined from this 5D feature-space using nearest neighbour—euclidean distance in this case. From this the learnt distribution can be used to bias either random search or seed an initial population in a genetic algorithm. Although this technique does speed-up iterative compilation as compared to non-biased random or genetic algorithms it still requires a substantial amount of work, in the form of the model built off-line. This paper is one of the more influential on this topic, but it is hypothesised by this author that the technique could have been accelerated further if active learning had been applied in this case, since it is unlikely that an exhaustive search would have been needed.

Machine learning approaches only work well if the training data on which a model is built is as representative as possible. Naïvely, one could try to achieve this by training on as many benchmarks as possible, however, just as Chapter 4 will discuss the inefficiencies of randomly collecting training examples which provide redundant data, Joshi et al. [2006] point out that there are redundant benchmarks which should be avoided too. In particular, the article proposes reducing the number of benchmarks worth training on to create a high-quality model by finding a subset of all benchmarks that are the most representative of the whole set. To demonstrate the potential of this technique they use a combination of *microarchitecture-independent features* together with PCA and clustering [Everitt, 2011] to find such representative benchmarks from well known benchmark suites. This is a good approach, so long as the features being used to characterise the benchmarks represent their behaviour in an accurate way. Unfortunately, since the features in this work are based on static analysis it is of limited use in practice. One might imagine two benchmarks with similar code features which do not behave in the same way at run-time due to dynamic behaviour, and, moreover, that different optimisations might be better applied because of these differences. Furthermore, in active learning there is an explicit, continuous learning process which evolves over time, where this methodology requires a vast number of benchmarks to be available at learning-time.

Pan and Eigenmann [2006] designed an algorithm they called *combined elimination* which iteratively discards transformations from a pre-defined set S that are found to cause an increase in runtime for a given code. This is done by first compiling and profiling a benchmark with an initial baseline B_t strategy, where all optimisations in S are applied; each optimisation is then individually turned off in isolation to see what affect this has on runtime; and the optimisations which are shown to degrade performance relative to the baseline are collected into an ordered set $X = x_1, x_2, \ldots, x_k$, such that x_1 is the most damaging. This most dangerous transformation x_1 is then immediately removed from S and a new baseline measurement taken B_{t+1}. The performance of the application when each optimisation in X is switched off in turn is again recorded, but this time measured with respect to the new baseline B_{t+1}. Those optimisations that still degrade performance, despite the absence of the X_1 transformation, are also removed from S. This process repeats until no transformations are found to harm performance. The main difficulty with this approach is that it is specific to each code and is expensive. As an alternative, machine learning can be used to generalise good optimisation strategies across programs and even architectures.

Coons et al. [2008] attempted to speed-up automatic construction of compiler heuristics, and in particular instruction placement, through careful feature selection. Their approach relied upon using LASSO (Least Absolute Shrinkage and Selection Operator) regression [Tibshirani, 1996] to find features which most closely affected performance; correlation coefficients to eliminate redundancy; and then a variant of Neuro-Evolution of Augmenting Topologies (NEAT) [Stanley and Miikkulainen, 2002] called FS-NEAT [Whiteson et al., 2005] to further constrain the search space, where NEAT is a type of genetic algorithm which drives the creation of a neural network. Unfortunately, the authors themselves state that for LASSO regression to work well it requires a large sample of performance results, something that this thesis explicitly attempts to avoid.

Similar to the work performed by Joshi et al. [2006] discussed previously, Thomson et al. [2009] used feature selection techniques combined with clustering in order to reduce the number of benchmarks that would be learnt from to only those that were generally well representative of a complete set. Specifically, their implementation counted the proportion of times a type of machine instruction was used in an unoptimised program, normalised to the total number of instructions in that program; from these counts 9 features were found to be particularly relevant by PCA; when projected into the reduced 9D feature-space, clustering was used to find a subset of benchmarks which were representative of the others; random optimisation strategies were then applied to each of these and the performance exhibited by the resultant binaries recorded; finally, a nearest neighbour model was used to find a good optimisation strategies given an unseen program's proximity in the reduced feature-space to existing training data. Although this technique can be used to speed-up learning by reducing the number of benchmarks being used for training purposes it suffers from the same drawbacks as the other related work by Joshi et al.; which is to say, by relying on static code features alone it potentially does not take into consideration run-time behaviour which may be useful in differentiating the good optimisation decisions from the bad.

Staying on the theme of feature selection, in Leather et al. [2009a] and Ting et al. [2017] both groups of researchers noted the importance of choosing good, representative features on which to train machine learning models in order to obtain quality outcomes, however, these works were not aimed at accelerating iterative compilation *per se* but rather at obtaining more accurate end results. In particular, Leather et al. [2009a] used a combination of *Grammatical Evoluation* [Ryan et al., 1998] and *Genetic Programming* [Koza, 1990] to search over possible features to map static code attributes to performance. In comparison, Ting et al. [2017] developed an automated framework which can similarly find good features on which to base a model using either LASSO regression, *sequential forward* or *backward selection* methods. Both these works suffer from the same basic drawback, they perform feature selection in an *a posteriori* way, which is inefficient since it requires collection of a large number of data before feature selection can take place—Chapter 6 attempts to remedy this problem.

Leather et al. [2009b] produced an interesting contribution and were the first researchers to optimise the number of samples required per program variant when determining which optimisation strategy may be the most profitable in a collection. In particular, they presented an algorithm which compares some number of candidate strategies by first taking an initial, fixed number of runtime samples for each and then using hypothesis testing—Welch's T-test [Welch, 1947]—to eliminate *losing strategies* from contention in a statistical manner. That is, if there is only one non-loser left, or if all the non-losers are deemed equivalent, then the search completes, otherwise the sample size is increased by one for all remaining candidates and the process continues until some maximum number of attempts is exceeded. This is the same approach used to classify data for Chapter 4, and it is a technique which is particularly useful where *n* candidates are present and a 'winning' strategy needs to be found quickly. That said, the paper does not apply this technique to machine learning. Moreover, no discussion of how candidate strategies should be collected in the first place is presented, whereas Chapters 4–6 in this thesis all address the optimisation of search through active learning.

Fursin and Temam [2010] proposed a unique solution to speed-up iterative compilation based upon what they termed *collective optimisation*. Their idea was to construct a central database to hold the performance obtained from using a particular optimisation strategy on a given program, input and architecture. In practice, users of their GCC modified compiler upload performance results for an attempted optimisation strategy to the database in the background and, transparently, a server associated with that same database then informs the compiler which optimisations to try next. When a lot of data has already been accumulated for a given code a program-specific probability distribution can be used to stochastically select a likely good optimisation strategy; when little is known reaction-based modelling [Cavazos et al., 2006] is used instead to probe which program-specific distribution the current program behaves like the most, and uses that data as a proxy; finally, when nothing is known the data from all program-specific distributions are used in aggregate. The selection of which of these three distributions to choose from is also stochastic, so generally the best optimisation strategy found so far will be used but other configurations are also tried. The most obvious disadvantage to this approach is that it uses a checksum to characterise the programs, where two programs that are similar but not exactly the same must be found through reaction-based modelling rather than an arguably more simplistic and efficient approach such as nearest neighbour. Having said that, implementation problems aside, this is a compelling idea, bringing 'big data' to the problem of program optimisation.

Staying on the topic of large scale data processing, Tartara and Reghizzi [2012] demonstrated that iterative compilation driven by machine learning could be accelerated by fitting the process to a MapReduce workload, so long as the `map` nodes had identical hardware and software configurations. Alternatively, they showed that it was possible to use such a MapReduce strategy on a single machine if specific tools were utilised to isolate parallel `map` executions from interfering with each other over shared resources. Unfortunately, this proposal relied upon random search for selecting optimisation strategies where an active learning approach could have been used instead by performing MapReduce in a `while` loop until a completion criterion was satisfied—similar to Balaprakash et al. [2013a].

Chen et al. [2012a]; Fang et al. [2015] proposed an idea similar to that presented by Tartara and Reghizzi [2012], which is to say they suggested that iterative compilation could be performed at scale in a MapReduce type workflow in order to speed-up search. Contrary to Tartara and Reghizzi, however, Fang et al. attempted to analyse and combat the problem of performance noise incurred by co-running programs on the `map` nodes, and thus presented a more realistic environment similar to that found in actual data centres as opposed to private and artificially unloaded machines. In particular, the authors showed that good optimisation strategies could still be found for MapReduce or compute-intensive server applications even in the presence of performance noise from co-runners, and that the type of co-runner made no difference to what constituted a good optimisation strategy for a given program. That said, they did find the best optimisation strategy changed based on co-runners, particularly when two co-running programs vie for the same resource—L2 cache, for example. In order to mitigate this behaviour they maintain a table which measures the influence co-runners have over each other and dynamically schedule them to minimise interference. Unlike the works presented in this thesis, no attempt is made to optimise search where the authors simply state it to be orthogonal to their work; furthermore, the number of sample runtimes for a program running under some input/co-runner combination is not dynamically found.

Balaprakash et al. [Balaprakash et al., 2013a,b] proposed the use of a relatively new type of modelling algorithm called dynamic trees [Taddy et al., 2009; Gramacy et al., 2013] in their works, in which they actively train models to predict good optimisation strategies for CPU or GPU codes, respectively. The details regarding the learning approach used in these papers are discussed in detail in Chapter 5, but the main drawback of their methodology lies in the fact that they do not optimise the number of samples they need to collect for a high-quality heuristic. Chapter 5 explicitly tackles this inefficiency by improving on their work, incorporating sequential analysis which is shown to greatly accelerate the learning speed of their dynamic tree based approach.

The paper by Zuluaga et al. [2013] presents an algorithm called Pareto Active Learning (PAL) which is useful for finding data that is highly likely to be Pareto-optimal. These are then evaluated to find the Pareto-frontier in a multi-objective optimisation-space. The researchers provide a theoretical analysis of their technique together with empirical results obtained from using three distinct datasets, where one is taken from Siegmund et al. [2012] and relates to optimising both performance and memory footprint by changing LLVM compiler flags. The algorithm itself uses *Gaussian Processes* (GPs) [Rasmussen and Williams, 2006] to estimate the mean and standard deviation vectors for each configuration in the design space as information accumulates. Using this data an *uncertainty region* or hyper-rectangle is constructed which can be used to estimate whether a point is likely Pareto-optimal or not. In a subsequent article Zuluaga et al. [2016] improves the technique by allowing a user to control the granularity of Pareto-optimal points in the frontier, thereby potentially further accelerating training at the expense of accuracy. Although these are interesting works, since they tackle multi-objective problems in systems literature, they are somewhat hindered by their use of GPs in that these are known to be particularly expensive to train and, thus, arguably less suitable in active learning works compared with other techniques, such as dynamic trees [Balaprakash et al., 2013a].

Ashouri et al. [2014] approach the problem of selecting good compiler optimisations per program in a similar way to Agakov et al. [2006], in so much as they use machine learning to focus iterative compilation towards more promising areas of the optimisation-space. That said, Ashouri et al. [2014] use a Bayesian network to establish a probability distribution learnt from training data which is then biased using the features of a test program. Another difference is that the more recent work uses micro-architecturally independent dynamic values instead of static code features to base their model upon. Compared with the works in this thesis, and others based on active learning, this technique is likely more inefficient since it samples uniformly from the optimisation-space during training which can result in redundant examples being evaluated.

Lastly, Martins et al. [2014, 2016] employ a novel clustering methodology to find functions which are similar to the one that they wish to optimise, in order to reduce the search time. To avoid any peculiarities intrinsic in code features they begin by first encoding the source code into a DNA like symbolic representation [Sanches and Cardoso, 2010]; next, the pair-wise distance between each representation is measured using the *Normalised Compression Distance* (NCD) [Cilibrasi and Vitanyi, 2005]; the NCD values are used by a *neighbour joining* algorithm [Felsenstein, 2003] to construct a Phylogenetic tree and from this a clustering algorithm can be used to partition the tree into sub-trees, where the leaf of a sub-tree on which a test function is present represents previously optimised functions over which a search technique can be applied. Although this technique does slim down the optimisation-space to more representative examples it does not deal with the efficient training of which optimisations should be applied to which function characterisations.

Summary

This chapter has provided a brief summary of all works which not only attempt some form of auto-tuning but also tackle the associated problem of long search times inherent in the techniques applied. In all cases, the contributions of this document are shown by qualitative or quantitative comparison to provide a substantive and novel addition to the literature.

Chapter 4

Heuristic Generation with Active Learning

As stated in the Introduction, heuristics are necessary because the optimisation-space for any given code is often so immense that it is infeasible to find the *best* optimisation strategy for a particular platform, ergo, a good one must suffice. Unfortunately, the traditional method of creating such heuristics through manual, fine-grained adjustments does not scale well with the speed at which new hardware is released. This is because it takes time to perfect heuristics and each distinct processor requires its own [Fursin et al., 2011]. Indeed, the result of following this failed methodology to date has too often been outdated heuristics [Kulkarni and Cavazos, 2012] and, thus, poorly optimised code. Therefore, to ensure good performance from any modern computer, for any code, an automated process to create these optimisation heuristics is required.

Machine learning based modelling has rapidly emerged as a viable means to automate heuristic construction; by running example programs optimised in different ways and observing how these variations affect program runtime, automatic tools can predict good settings for new, as yet unseen, programs. This research area is promising, having the potential to fundamentally change the way in which heuristics are designed, however, before this potential can be realised there remain hurdles which must be tackled. In particular, the concern targeted in this thesis is excessive training time. Which is to say, while machine learning allows heuristics to be automatically constructed with little human involvement the cost of collecting training data (that allows a learning algorithm to accumulate knowledge) is often very high. This chapter presents a novel, low-cost and broadly applicable predictive modelling approach that can significantly reduce the overhead of collecting this training data without sacrificing model quality.

In predictive modelling it is often common for training data to be gathered in a random fashion, and there are good reasons for this since it avoids inadvertent bias, giving a more representative coverage of the space. Naturally, random collection can also encounter redundant information since multiple training instances can have both very similar output and feature values. In contexts where obtaining training data is fairly cheap this is not a problem, but where collection is relatively expensive it means some considerable cost is being paid with little or no resultant improvement to the quality of the final model. All machine learning based heuristics literature until recently has employed the former, random approach, but this ignores the expense of labelling each training example. In particular, when training some optimisation model it is necessary to run a given program under differing conditions multiple times to get relatively sound measurements of their effect, where repeated compilation is often also a part of this process. Clearly, if some of this effort is exhausted and nothing useful has been learnt in the process then time and money have both been wasted. In this chapter, this substantial problem is targeted by employing active learning.

In the sections which follow the effectiveness of leveraging active machine learning to build optimisation models versus random learning, which is called passive learning in the literature, is demonstrated. In particular, heuristics are constructed to determine which processor will give the better performance on a CPU–GPU based heterogeneous system at differing inputs for different programs, where across four benchmarks the average learning speed-up is 4x under specific test conditions.

The rest of this chapter is organised as follows: in Section 4.1 a motivating example is given for this work; in Section 4.2 an overview of the approach and the implementation details of the system are discussed; in Section 4.3 the experimental setup used to validate the technique is outlined; Section 4.4 provides the results and accompanying analysis; Section 4.5 critiques the work in a constructive manner; and a summary is given in Section 4.6.

4.1 Motivation

To motivate this research it is first demonstrated how much unnecessary effort can be involved in the traditional, randomisation-based passive learning techniques used extensively in prior literature, and to point out in what way the proposed active learning strategy can improve upon efficiency.

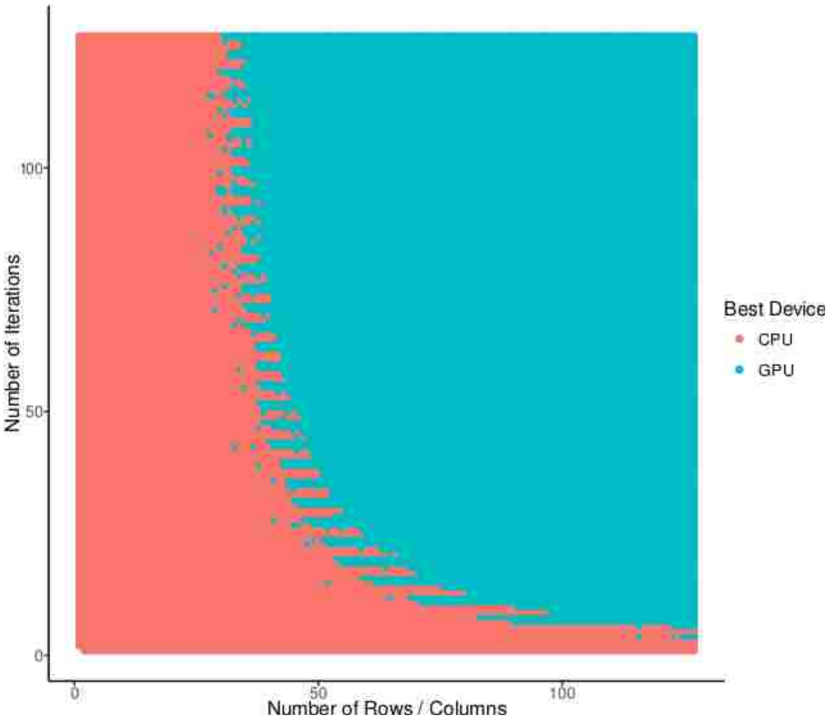


Figure 4.1: the problem-space

In Figure 4.1 it is shown for the HotSpot benchmark [Huang et al., 2006] (from the Rodinia suite [Che et al., 2009, 2010]) when it is better to run on the CPU using OPENMP [OpenMP, 2011] or the GPU with OPENCL [OpenCL, 2012] for maximum performance. This benchmark accepts two independent program inputs and their values form the axes of the graph. The data itself was generated by selecting all input permutations in the space and running them on both the CPU and GPU enough times to make a statistically sound decision about which device is better for each individual case.

Automated learning has been shown to be a viable option for creating heuristics for this type of problem [Cooper et al., 1999; Wang and O'Boyle, 2009]. To build such a heuristic, a learning algorithm requires a set of training examples to learn from, where, in this case, each example requires profiling a program with a permutation of input values in order to label it. In Figure 4.2 a random selection of 200 input permutations for HotSpot has been chosen. This was achieved by putting all possible permutations for the space shown into a 'bag' and selecting them with equal likelihood (i.e. with a uniform distribution) over a single iteration. These distinct examples are then put through the labelling process and from this a training set emerges. A heuristic was created from this data using the RandomForest machine learning algorithm from the

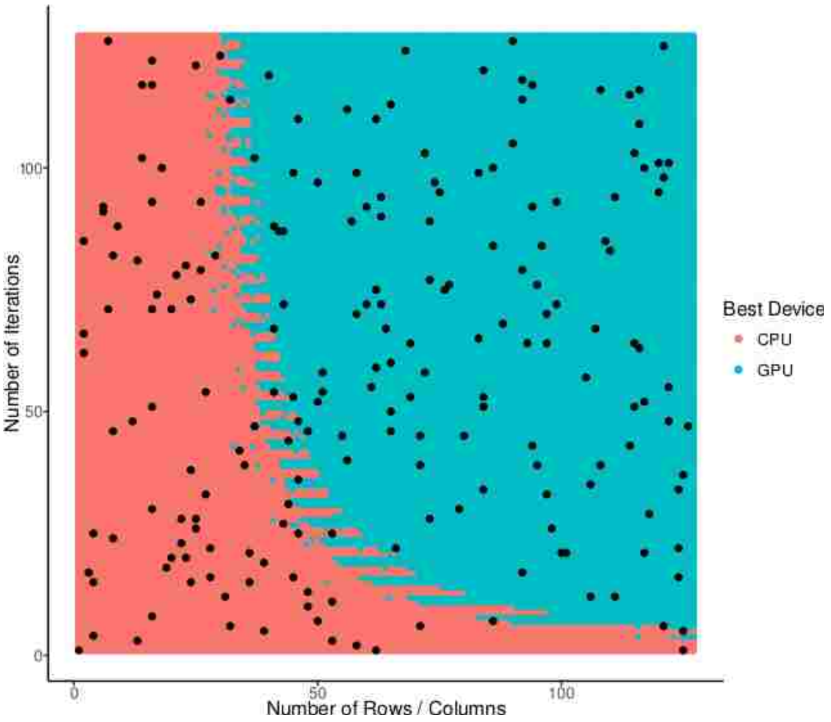


Figure 4.2: random training instances

Weka Toolkit v3.6.9 [Hall et al., 2009; Frank et al., 2016]. This heuristic was able to achieve a respectable 95% accuracy rate when a randomly chosen test set of 500 examples was used for evaluation, where the training and test sets were kept disjoint.

From this quick example it is clear that machine learning can create good heuristics in this case, however, intuition insists that a proportion of the randomly-selected training examples may offer little useful information. In fact, given the appearance of the problem-space in Figure 4.1, it was hypothesised that those training examples chosen nearest to the reciprocal-shaped boundary would have the most impact: since they best define the trajectory of that separation. The intuition of redundant examples is proved correct in a follow-up experiment, where just 29 training examples have been selected to form the training set instead. Evaluating with the same test set as before gave an accuracy of 97% despite 15% as many examples being used—see Figure 4.3. This indicates that there is significant potential to reduce the training cost for machine learned heuristics if only the better examples to train over can be found. Unfortunately, without already knowing the shape of the space it is impossible to tell what these examples should be, but nevertheless it will be shown that with active learning techniques it is possible to approximate their location.

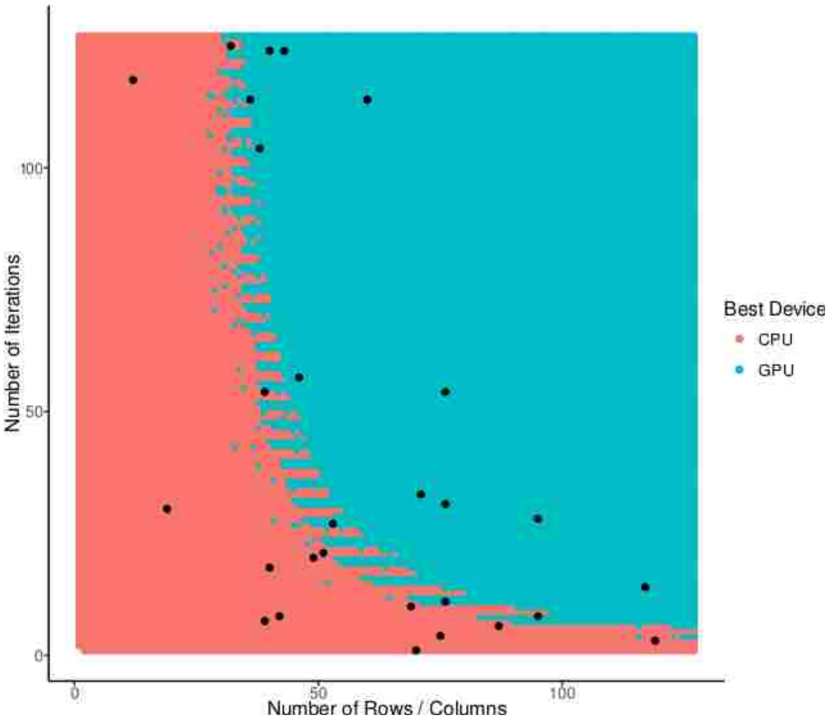


Figure 4.3: intelligent training instances

In this chapter a simple active learning technique is provided which accumulates a set of training examples gradually, only adding to the set those instances the algorithm predicts will most improve the quality of the heuristic. The following section describes the methodology in detail.

4.2 Methodology

As a case study, this work aims to train a predictor to determine the best processor to use for given program inputs while avoiding profiling those that provide little or no information to the learning algorithm. This is achieved by using an active learning approach which carefully chooses each example to be examined in turn, where these are selected based on their predicted usefulness.

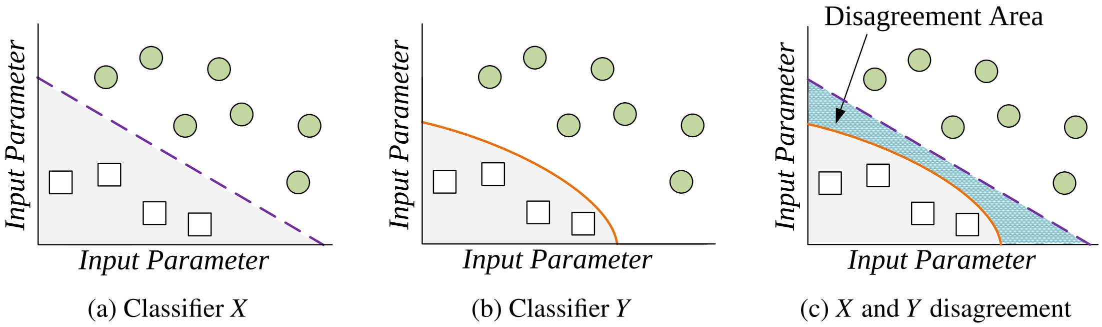
4.2.1 The Query-by-Committee Technique

The form of active learning used to create the inputs-to-device mappings in this work is called *Query-by-Committee* (QBC) [Seung et al., 1992]. The QBC algorithm follows the same basic structure as that outlined in Subsection 2.4.1, and shown graphically in Figure 2.5, but differs in that it uses an ensemble of intermediate models in combination to provide both predictions and to select new, informative training examples to learn from.

There are a number of ways to implement QBC but the 'committee' used in this work consists of numerous randomisation-based learners. Each member algorithm is constructed using a distinct seed value which initialises an internal psuedo-random number generator; each are also given the exact same small training set, where the examples constituting this set were selected at random; however, since each learner has a different seed they tend to form different intermediate models despite possessing the same data. In order to label a new, previously unseen instance its class is defined as the majority class predicted by the models making up the committee. To determine which candidate should be chosen to be labelled next the QBC algorithm asks each constitute member to predict to which class each candidate in a pool of potential training examples should belong. The candidate with which the members disagree the most is then selected, as this suggests that the models have something to gain from its labelling.

The insight behind the QBC technique is that it is thought less profitable to learn from parts of the problem-space which are already collectively (and relatively) well understood, and rather it is more beneficial to learn from regions which are least well defined. Gradually over time these regions of disagreement shrink as data is fetched from them, and the model converges upon the true boundary over which the most efficient processing device should be changed. This corresponds to the idea discussed with respect to Figure 4.3, that the most informative points are likely closest to the reciprocal-shaped boundary in that figure since they will most accuracy define its trajectory.

An example Figure 4.4 provides a hypothetical example to demonstrate how new training examples are selected by QBC. In Figure 4.4(a) an input-space is presented which is fully described by two parameters and has the location of some training examples already shown, where the shape of the points indicate different classes. In this case, the committee consists of two classifiers X and Y with different seeds giving different models, as illustrated in Figure 4.4(a) and (b). If the classification boundaries of the two models are overlapped, as in Figure 4.4(c), it becomes clear that there are parts of the space that both classifiers are in agreement about as well as an area where they disagree. Knowing the location of any disagreement region the algorithm can then select new training examples from within that, thereby maximising the likelihood that something new will be collectively learnt.



(a) Classifier X (b) Classifier Y (c) X and Y disagreement

Figure 4.4: a simplified two-parameter feature-space with the locations of profiled training examples marked, where two distinct learning algorithms build two different models—(a) and (b). These models are then combined, as in (c), to find the region of disagreement between them and this information used to better select where future training examples should be drawn from.

4.2.2 Quantifying Committee Disagreement

Shannon information entropy H [Shannon, 1948] is used to evaluate the level of disagreement between committee members for each candidate training example, where in Equation 4.1 $p(x_i)$ is the proportion of committee members that predict that the candidate instance x is fastest on device i of n. If out of all candidate instances, per learning loop iteration, a multiple number have the same maximum entropy value then one is randomly chosen from amongst those to be profiled next. In this case, profiling or labelling involves using the input values associated with that training example to run the CPU and GPU kernels enough times to determine which processor is faster in that case. Once this information has been added to the training set the learning loop begins another iteration forming fresh intermediate models, only this time new data has been included.

$$H(x) = -\sum_{i=1}^{n} p(x_i) \log p(x_i) \tag{4.1}$$

4.2.3 Statistically Sound Performance Profiling

Since computer experiments, particularly those that require accurate time measurements, are known to give noisy results statistical techniques are used to increase the reliability of the models being produced in this work. In particular, a minimum number of performance samples per device are recorded, as specified by the user. Furthermore, IQR outlier removal [Moore and McCabe, 2005] is applied before Welch's t-test [Welch, 1947] to discover if one hardware device is indeed statistically faster than the other. If it cannot be concluded from the t-test that this is the case then an equivalence test is performed. Both devices are said to be 'equivalent' if the difference between the higher mean runtime plus its confidence interval minus the lower mean minus its confidence is within some pre-defined threshold of indifference. This threshold is set to be within some percentage of the minimum of the two means. If the fastest device cannot be determined and yet they are not deemed equivalent then a single extra sample per device is obtained and the tests applied again, up until some pre-set number of tries. For classification purposes, if the devices are determined to be equivalent or the number of maximum tries has been exceeded then the CPU is chosen as the preferred device since it is at least more energy-efficient than the GPU: see Figure 4.5 for a graphical example of the equivalence test and Algorithm 1 for some pseudo-code of the labelling procedure.

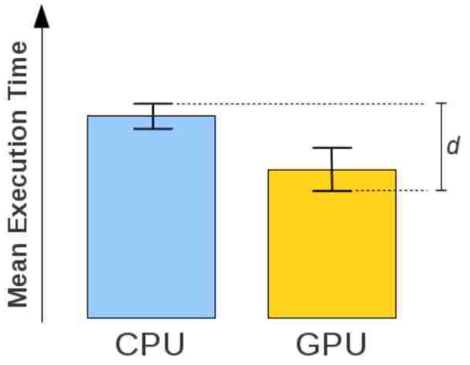


Figure 4.5: there is no mathematical formula for equivalence but a heuristic can be used which relies upon the confidence intervals of the two classes. For example, if the maximum difference between the 95% confidence intervals d is within 1% of the minimum mean runtime (of the GPU in the case above) then the devices are said to provide equivalent performance. In such cases the CPU is designated as the more performant hardware since at least this increases energy-efficiency.

4.3 Experimental Setup

This section gives the exact details of the experiments carried out in this case study, starting with the platform and benchmarks used, moving onto the particular QBC settings, and finally discussing the evaluation methodology.

Algorithm 1 to label an instance, with features X, as being either faster on the CPU or the GPU the minimum n_{min} and maximum n_{max} number of sample runtimes to record per device needs to be set. A confidence level C is also required for Welch's t-test and the equivalence test, as well as a threshold d—see Subsection 4.2.3.

```
procedure LABELINSTANCE($X, n_{min}, n_{max}, C, d$)
    for $i \in \{cpu, gpu\}$ do
        $Y_i \leftarrow \emptyset$
        for $j = 0, n_{min}$ do
            $Y_i \leftarrow Y_i \cup$timeExectionOnDevice($X, i$)
        end for
        $Y_i \leftarrow$ removeOutliers($Y_i$)
    end for
    repeat
        if isStatisticallyDifferent($Y_{cpu}, Y_{gpu}, C$) then
            if mean($Y_{cpu}$) $<=$ mean($Y_{gpu}$) then
                return $cpu$
            end if
            return $gpu$
        end if
        if isEquivalent($Y_{cpu}, Y_{gpu}, C, d$) then
            return $cpu$
        end if
        for $i \in \{cpu, gpu\}$ do
            $Y_i \leftarrow Y_i \cup$ timeExectionOnDevice($X, i$)
            $Y_i \leftarrow$ removeOutliers($Y_i$)
        end for
    until $|Y_{cpu}| >= n_{max}$
    return $cpu$
end procedure
```

4.3.1 Platform and Benchmarks

Platform The approach was evaluated on a CPU–GPU based heterogeneous platform with an Intel Core i7-4770k CPU, containing 4-cores (8 hardware threads) which run at 3.4 GHz. The machine has 16 GB of RAM and the GPU is an NVIDIA GeForce GTX Titan with 6 GB of memory. The operating system is OpenSuse Linux v12.3 and both GCC v4.7.2 and the NVIDIA CUDA Toolkit v5.5 are used for compilation.

Benchmarks In terms of benchmarks, three were selected from the Rodinia suite based on their ability to allow fine-grained changes to their kernel inputs and since they had implementations in both OpenMP and OpenCL—namely `HotSpot`, `PathFinder`, and `SRAD`. A simple matrix multiplication kernel was also written in both OpenMP and OpenCL to allow for increased testing. Table 4.1 gives all relevant details regarding the input-space explored for each of these benchmarks, where the input ranges were chosen to give a substantial, but fairly realistic, region to learn over based on default testing values. It is vitally important to mention here though that the size of the input-space will have a significant impact upon the value of any resultant speed-ups calculated, and so the relevant information during experimentation is more that there is a significant effect rather than the absolute value itself.

Table 4.1: the *Size* of the input-space for each of the benchmarks are given in the table below, as well as the number of *Dimensions*; the *Min* and *Max* values for each dimension, inclusively; the number of *Candidates* considered at each loop iteration; and the step value of the *Stride*.

Benchmark	**Dimentions**	**Min**	**Max**	**Stride**	**Size**	**Candidates**
`HotSpot`	2	1	128	1	16,384	10,000
`Matrix Mult.`	3	1	256	1	1.6×10^7	10,000
`PathFinder`	2	2	1,024	1	1.0×10^6	10,000
`SRAD`	2	128	1,024	16	3,136	2,636

4.3.2 Active Learning Settings

Learning Models The active learning committee was formed using 5 `RandomForest` classifier objects from the Weka Toolkit v3.6.9, where each was given a unique seed. This algorithm was selected since it was found to give the most accurate models for all benchmarks during experimentation, it can produce distinct models with different initial seed values, and it is used commonly in machine learning literature.

Initial Training Set and Candidate Set Sizes For all experiments the training set was initialised with a single randomly chosen instance, i.e. the minimum possible. The effect of changing this parameter is discussed in Subsection 4.4.3. The candidate set size was either 10,000 examples not already present in the training and test sets or the maximum number of points not in those sets, whichever was smaller—see Table 4.1.

Termination Criterion The learning loop was halted when the number of training examples learnt by the algorithm reached 200, this was found experimentally to be sufficient for the improvement in heuristic quality to have reached a fairly consistent plateau.

4.3.3 Evaluation Methodology

Runtime Measurement and Device Comparison To determine if a benchmark with a given input permutation was better suited to the CPU or GPU it was run on each device using OPENMP or OPENCL, respectively, at least 10 times and at most 200 times: N.B. different programming frameworks were used since OPENCL was not thought mature on the CPU and OPENMP could not execute on a GPU at that time. As mentioned in Section 4.2, IQR outlier removal was employed, with Welch's t-test and equivalence testing to ensure the statistical soundness of the gathered program execution times.

Statistical Difference and Equivalence Testing In order to determine if one compute device is indeed better suited for a particular workload size over the other Welch's t-test is used. The t-test is parametrised by a confidence value C of 0.95, i.e. a 95% confidence level, which is generally considered to be the minimum standard for scientific experiments. If it cannot be determined that the devices have mean runtimes which are statistically different enough to judge that one *is* better than the other then an equivalence test is attempted. This equivalence test (see Figure 4.5 for the details) uses the same confidence level (95%) with a threshold d of 1%.

Testing For testing purposes, a set of 500 inputs were excluded from any training and candidate sets. Both the active and passive learning experiments were run 10 times for each benchmark and the arithmetic mean of Cohen's Kappa was recorded. For active learning a committee of 5 models were used in combination to form a single implicit model to predict classifications of test instances, or to work out entropy for selecting the next training instance, based on 'votes' cast; in comparison, for passive learning a single classification model was used as this was the *de facto* state of the art.

4.4 Experimental Results

This section begins by presenting the overall results of these novel experiments, showing that an active learning approach can significantly reduce the training time by a factor of four on average, when compared to the passive learning technique used in prior works. Then, the performance exhibited by the learning system for each benchmark in turn is examined. Finally, how three user-supplied parameters affect the performance of the methodology is inspected and discussed.

4.4.1 Overall Efficiency Savings

Figure 4.6 shows the average learning speed-up of an active approach over the passive, random technique traditionally used in automated heuristic construction.

The speed-up values are based on the average relative costs, in terms of training examples required, for the maximum common kappa value to be obtained through active versus passive learning. As can be seen from Figure 4.6, active learning consistently outperforms the classical technique for all benchmarks tested, which in real terms means saving weeks of collecting data.

That said, as mentioned previously in Subsection 4.3.1, these absolute speed-up values are heavily dependant upon the size of the input-space so the important point here is that the active learning methodology does significantly accelerate learning, not that it accelerates learning by some calculated amount.

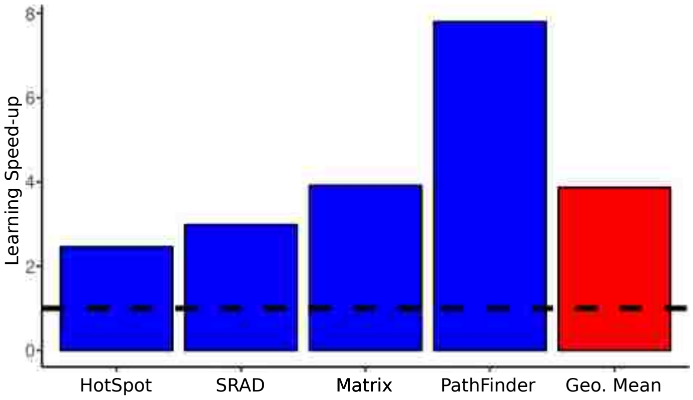


Figure 4.6: on average this active learning implementation requires 4x fewer training examples to create a high-quality heuristic on the given input-spaces, as compared to the traditional, random learning technique.

4.4.2 Training Examples Selection

Figures 4.7–4.10 clearly illustrate where the cost savings associated with this QBC implementation are coming from, where each graph shows the training points selected by the QBC algorithm in a single run. That is to say, in all cases the algorithm quickly chooses points surrounding the boundary between the CPU and the GPU optimum regions, giving it the ability to more accurately model its shape in less time.

4.4.3 Sensitivity to Parameters

As well as confirming the validity of this active learning approach three further experiments were conducted to determine the impact that some user defined parameters might have on its effectiveness. The first experiment (Figure 4.11) involved altering how many randomly-selected training examples were initially supplied to the QBC algorithm to get it started. The second experiment, whose results are presented in Figure 4.12, investigated the extent to which changing the size of the candidate set would have an effect on the speed of heuristic construction. Finally, Figure 4.13 gives an indication of the impact of increasing the number of committee members. For all three experiments, the benchmark used was `HotSpot` and all variations per experiment were run 10 times to calculate the respective mean performances.

From Figure 4.11 it is clear that increasing the number of random training instances used to seed the QBC algorithm for `HotSpot` has no significant effect on its long-term performance, but is somewhat detrimental in the short-term. This makes sense since it simply reflects the behaviour of passive learning. Which is to say, passive learning is slower than QBC at reaching high kappa values but once it has accumulated enough random knowledge it is competitive with an active training set. That said, one can imagine a case where a complex space with many localised features may be better explored through an initially random approach followed by active learning.

Figure 4.12 illustrates how changing the size of the candidate set for the `HotSpot` benchmark affects the performance of the system. In particular, the data indicates that although a smaller candidate set size *may* be more beneficial up to a point, the absolute difference between the best and worst kappa values for any variation is small enough that this could actually be the result of experimental noise, despite repeated runs.

In Figure 4.13 the committee size was altered to see what affect this might have on the learning curve for the `HotSpot` benchmark. The results show that there is little difference made when a higher number of committee members is chosen—5 is fine.

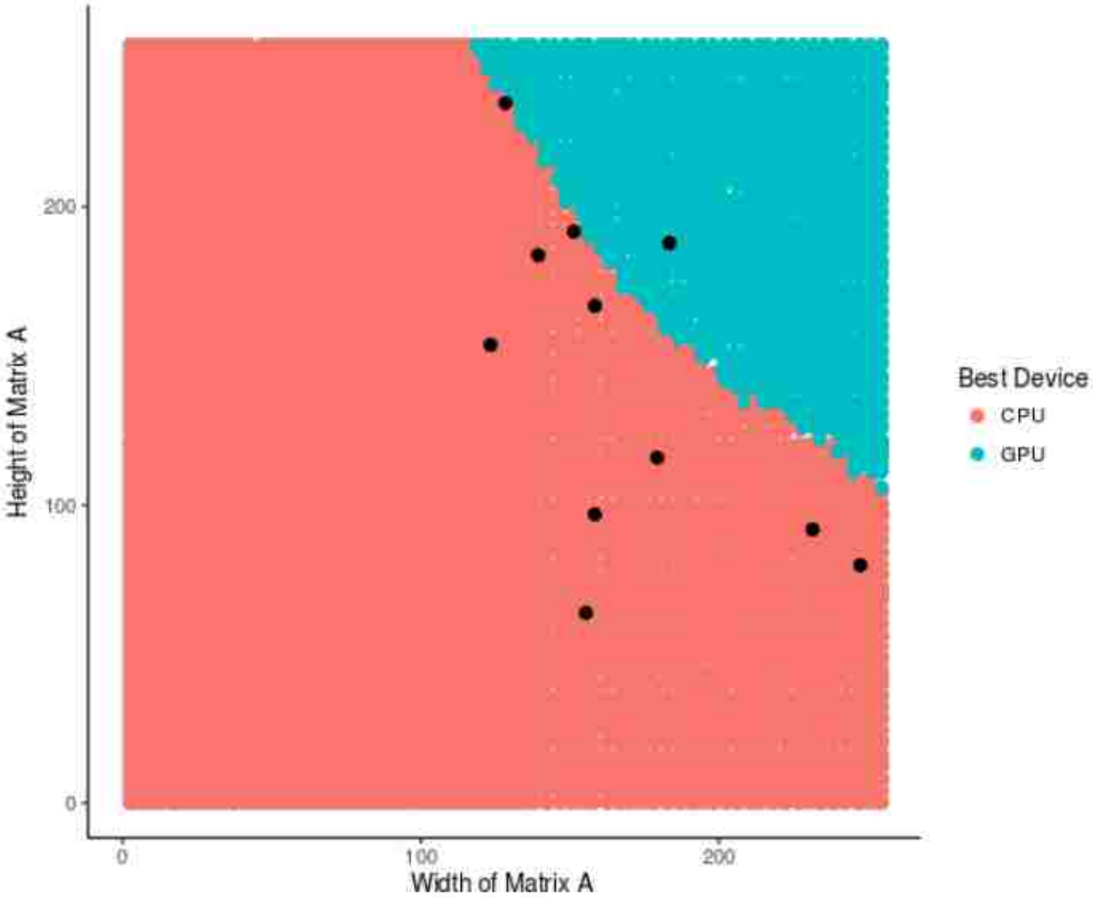


Figure 4.7: since the Matrix Multiplication space is three-dimensional it is slightly more difficult for a human to visualise the separation between CPU and GPU regions; to make things a little easier the graph above was flattened such that the z-axis has values $122 \leq z \leq 130$. Active was approximately four-times faster than passive learning at producing a high-quality model for this code, one of the quickest tested, presumably because the additional dimension reduces the effectiveness of random selection further.

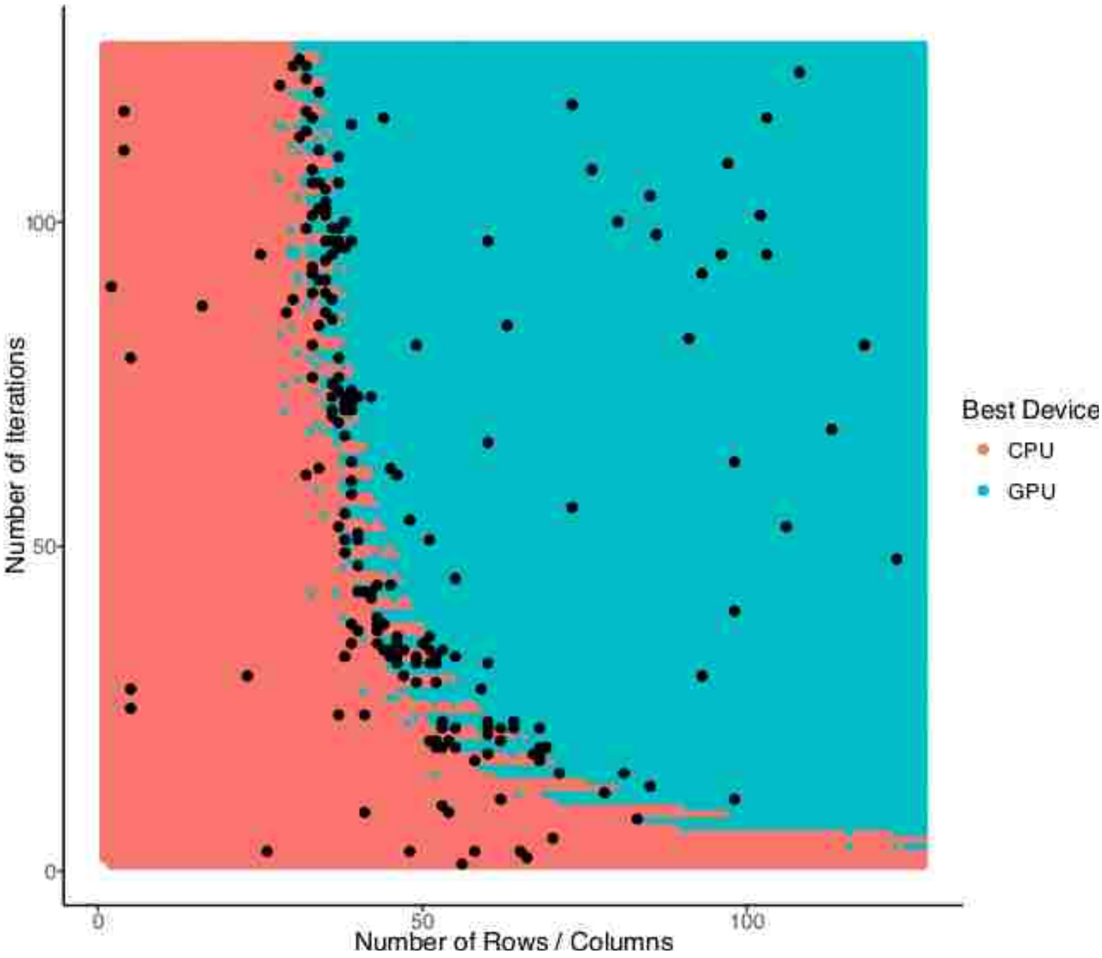


Figure 4.8: the efficiency of QBC at finding the boundary between the best class of device is dramatic for the `HotSpot` code. When compared to a random approach it is easy to understand how QBC would be over twice as fast at achieving a high kappa value, indicating a quality heuristic.

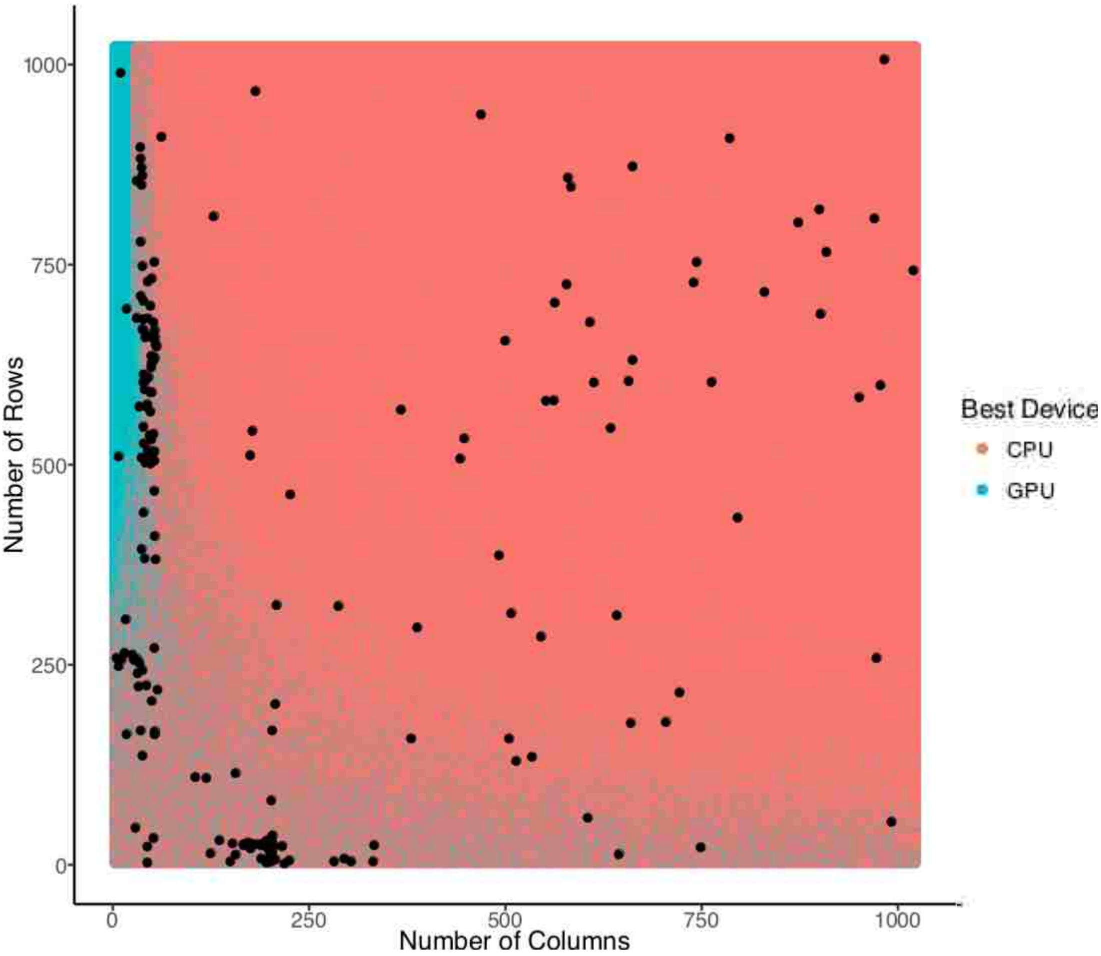


Figure 4.9: active learning was nearly eight times faster at producing a quality model than passive learning for the `PathFinder` benchmark, and the reasons are obvious. The input-space for this benchmark was one of the largest and the location of the region at which the GPU is fastest is relatively small. Thus, a random approach will naturally find it difficult to locate the boundary versus an 'intelligent' approach.

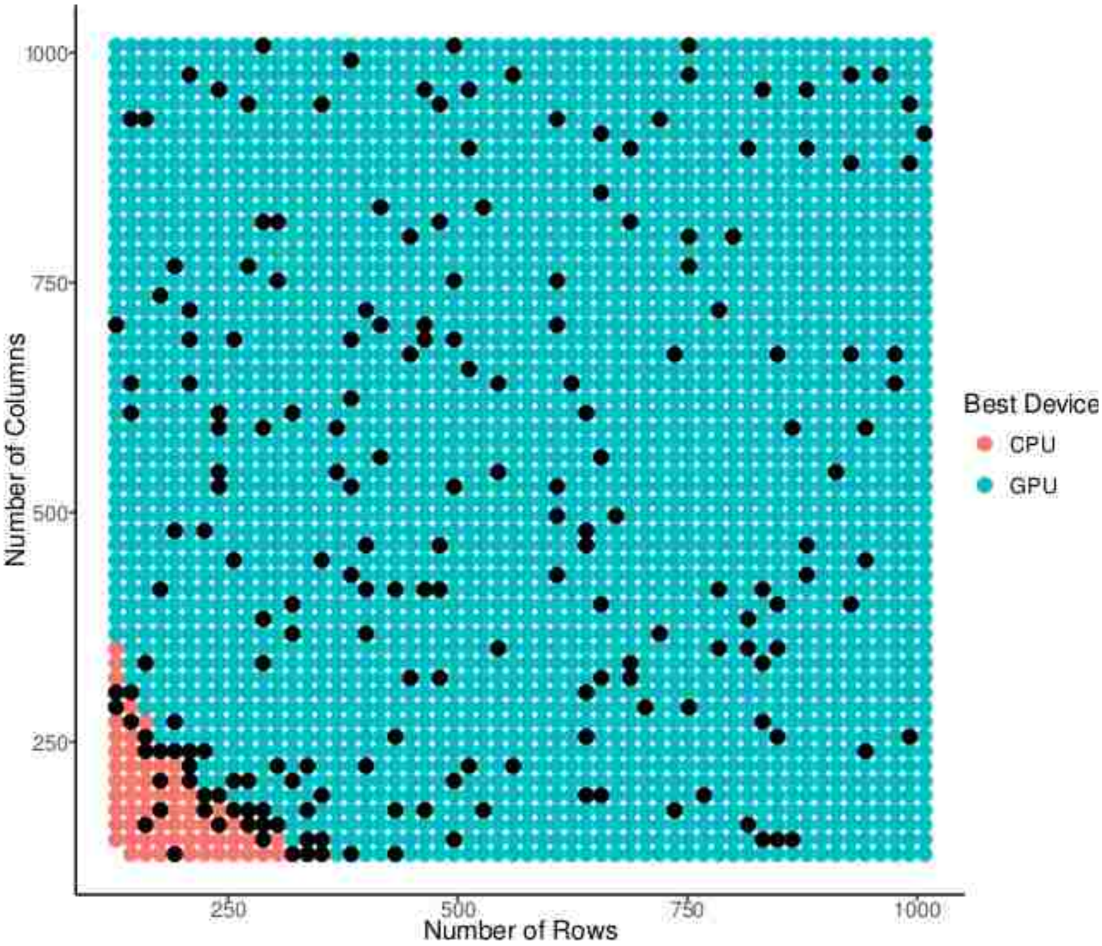


Figure 4.10: somewhat similar to Figure 4.9 in that one device dominates the input-space, the points selected to train on SRAD appear to be initially random but then can concentrate on the relevant boundary once discovered. This gives the QBC approach an advantage over random learning, and is the reason why it is ultimately three times faster than that classical technique at producing a high-quality heuristic.

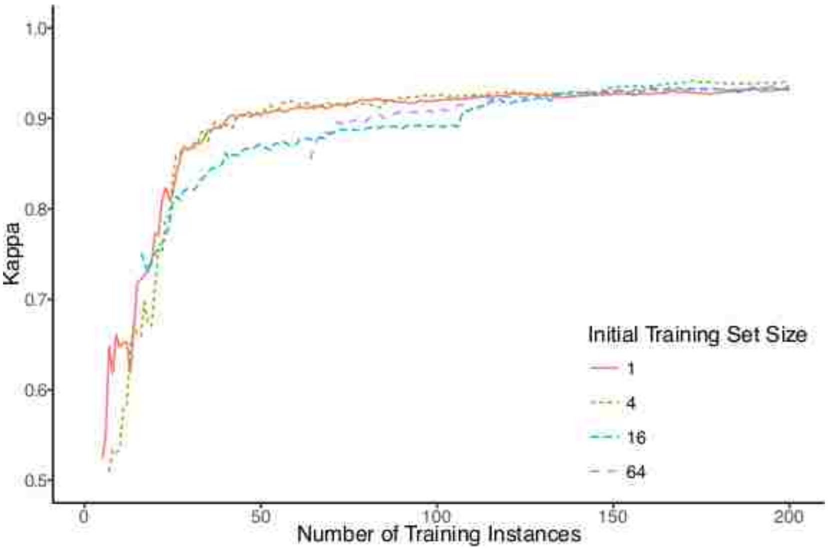


Figure 4.11: this figure shows that increasing the number of random examples given initially to the QBC algorithm for `HotSpot` is on average at first detrimental to its performance but in the long-term has no significant effect on the model quality; however, in a more complex feature-space increased randomness may help uncover small localised features.

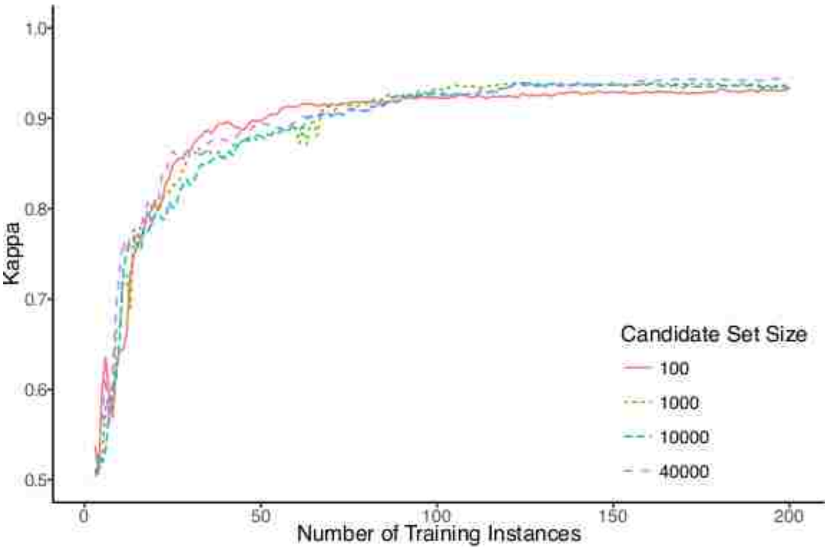


Figure 4.12: this figure shows that choosing a lower candidate set size *may* be more beneficial than a larger one up to a point; however, as the absolute difference of the best and worst kappa values between any two variations is rather small this could be down to experimental noise.

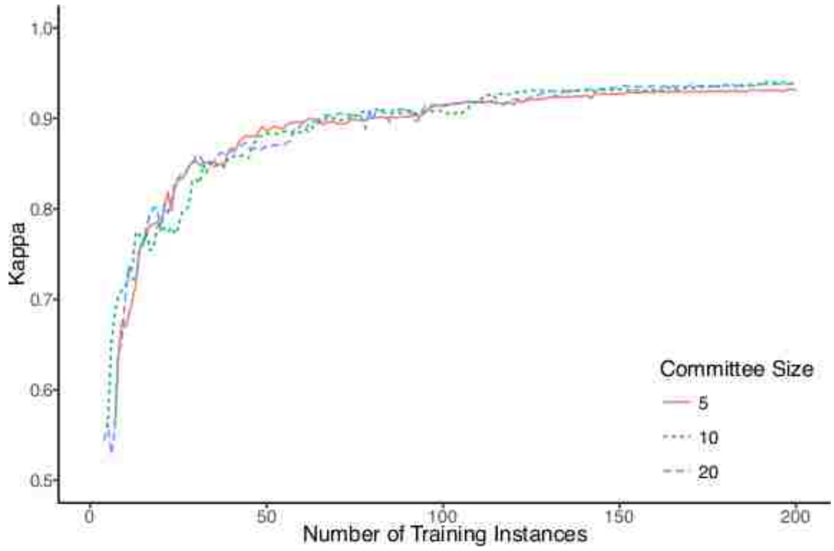


Figure 4.13: choosing a higher number of committee members seems to make little difference to the rate of change in the learning curve.

4.5 Discussion

This section provides some self-reflective criticism of the work presented thus far, and how it might be improved upon. In particular, a way to combat the greedy nature of the QBC algorithm to ensure localised spatial features might be better reflected in the final model is presented. A novel idea for an entropy-related completion criterion is also investigated and discussed.

4.5.1 Localised Classification Changes

It was speculated, prior to the commencement of this research, that there would be some global transition within each program input-space between configurations for which the CPU is better than the GPU and vice versa. In fact, it was assumed that the CPU would naturally be quicker at small workload sizes since there is an overhead associated with transferring data over the PCIe bus—Peripheral Component Interconnect Express—and that a GPU, with its relatively high number of processing units would then overtake as the better device. In practice, although the transition has not always been from CPU to GPU with increasing work, the input-spaces were all found to be nicely separable into two distinct regions as predicted. The QBC algorithm was chosen based on this prediction, but, if one imagines a more complex input-space, such as that imagined in Figure 4.14, it is clear that QBC could run into a problem.

More specifically, if a QBC algorithm detects points which surrounds one boundary but misses another then it will assume incorrectly that the space is separable into just two distinct regions.

An obvious solution to this problem would be to increase the number of random training examples at the beginning of the learning process, but this would not be sufficient in itself since it is still possible that a region might still be missed. Rather, it is suggested that at every n iterations of the learning loop a training example is selected which is deliberately not near the known boundary but actually some distance away from already explored parts of the space. Thus, increasing the likelihood in such a situation as displayed that all localised changes will eventually be found. Alternatively, an ε-greedy approach [Sutton and Barto, 1998] could be used to combine *exploitation* and *exploration*, such that the probability of selecting a random instance to learn is ε and $1-\varepsilon$ is the probability of selecting a high-entropy example instead. Unfortunately, since we have not encountered such a complex space this method has not been evaluated, but as this is a well known strategy to combat this problem it seems sensible to include it here for completeness.

4.5.2 Entropy Completion Criterion

In Subsection 4.3.2 it was described how the committee of models was comprised of 5 Random Forest classifiers, each given the same training data but also a distinct seed. Such a committee works because the psuedo-random number generator within each object will produce different outputs which then, in turn, has an effect on the aggregate model shape. Having said that, it is likely that as time goes on, and more data is accumulated, each model will begin to converge on one another. It would therefore be interesting to examine whether or not it would be possible for a completion criteria to be based on total entropy within the candidate set. Which is to say, if the total entropy of n consecutive learning loop iterations is below a certain threshold then the process is deemed complete, since all committee members agree in the main on the class distribution for the space. This would seem to be a far more natural completion state than an arbitrary time limit or size of the training set. Unlike Subsection 4.5.1, which cannot be properly evaluated since it would require the creation of an artificial problem, the success of this technique can be measured.

In a brief experiment, the `HotSpot` benchmark is evaluated again using the same active learning strategy and parameters given in Section 4.3 but this time metrics sum-

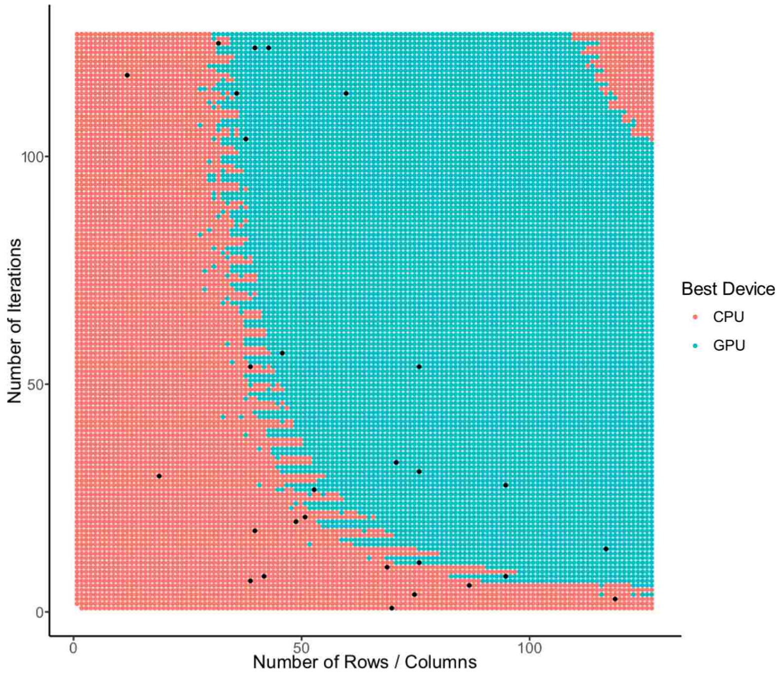


Figure 4.14: QBC copes well with a singular classification boundary but for more complex spaces a more elaborate approach may be required. For example, this figure shows the same data as that presented in Figure 4.3, except that the classification of points in the upper extreme of the space have been artificially altered. If the points selected to be sampled are the same (shown as block dots) the final model will incorrectly assume the space can be separated into two distinct regions. To combat this problem at every n iterations of the learning loop a random training instance biased away from current known points could be selected for labelling, or an ε-greedy strategy could be used instead.

marising the distribution of entropy values within the candidate set are recorded. These metrics are the minimum and maximum entropy values; the 25th, 50th, and 75th percentiles of that distribution; and the mean. Figure 4.15 gives the results aggregated over ten runs. For ease of comprehension only the average maximum and median entropy values at each iteration of the learning loop are plotted alongside average kappa. What the data shows is that although the entropy values for most of the candidate set in any given distribution is zero there is a relationship between a decreasing kappa and the maximum entropy observed. Unfortunately, the relationship is so volatile, however, that it is doubtful that such a technique could be used in practice and so more conventional completion criterion must be relied upon.

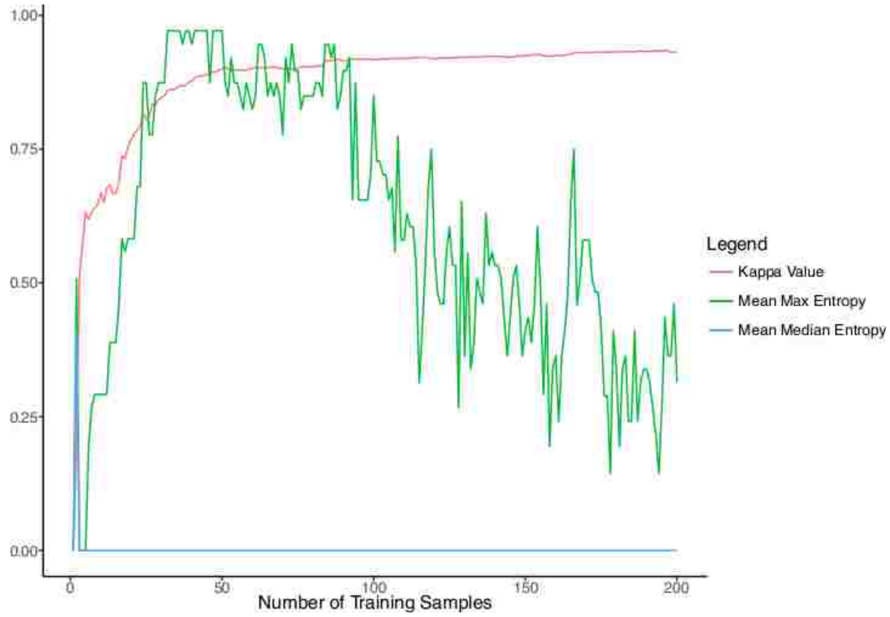


Figure 4.15: it appears from this experiment that the average entropy value of a candidate in the candidate set is zero, particularly after an initial phase at the beginning. There is a relationship between average maximum entropy and average kappa but the data is so volatile that it would appear this is not a reliable stopping criterion after all.

4.6 Summary

This chapter has presented a novel, low-cost, and broadly applicable predictive modelling approach for machine learning based heuristic construction. Instead of building heuristics based on randomly chosen training examples, as was the previous standard process, active learning is used to focus on those instances that improve the quality of the resultant models the most. Using QBC to construct a heuristic to predict which processor to use for given program input values the approach is able to speed-up training by a factor of 4x for the given input-spaces, resulting in a saving of weeks of compute time.

This chapter has also attempted to suggest two techniques which might improve the overall approach through introducing some randomness back into the technique to help the discovery of any localised optimal classification changes, and a novel completion criterion dependent on total entropy in the candidate set over a number of consecutive iterations. Unfortunately, this latter suggestion was found to be unreliable after some experimentation.

Chapter 5

Active Learning with Sequential Analysis

In Chapter 4 it was pointed out that the majority of prior work has used a random, passive training technique to automatically construct optimisation heuristics. It was then demonstrated that the learning overhead of generating such models could in principle be reduced by using active machine learning instead. Indeed, other works have drawn the same conclusion [Zuluaga et al., 2013; Balaprakash et al., 2013a,b] and, when combined, these represent a substantial leap forward towards making heuristic generation quick and easy. Unfortunately, despite this, sizeable inefficiency does still exist in the training process. More specifically, all previous literature on machine learning based auto-tuning has used a fixed sampling plan to collect training data. Which is to say, each unique training instance is repeatedly profiled a set number of times, chosen *a priori*, to obtain a reasonable estimate of the runtime so that the affect of the configuration being tested can be accurately known. This is necessary since runtime measurements are inherently noisy, but it is also a potential source of wasted effort since a fixed sample size *n* does not allow for the case where less than *n* samples are all that are necessary, given a broader context.

The subsections which follow will give a brief discussion as to the origins of noise in computer experiments, how that noise can be reduced to ameliorate the need for repeated profiling, and argue that such noise reduction techniques may not actually be desirable. Instead, a summary of an adaptive system which can dynamically determine at learning-time how many samples to record per training example will be presented, followed by a brief outline of the layout for the remainder of this chapter.

The Origins of Runtime Noise

There are many sources of noise which perturb runtime measurements, the most egregious of which are caused by system processes or those belonging to other users. Such processes compete for resources with the tested application, especially for cores, caches [Petoumenos et al., 2006], and memory [Zhuravlev et al., 2010]; moreover, they do so in non-deterministic ways. In recent systems the power and thermal walls have lead to even more complex interference patterns. Intel's Turbo Boost technology, for example, might lower the frequency and the power consumption of a process running on a core when other cores wake up [Charles et al., 2009].

Even ignoring interference from other applications, there are still more sources of noise in computer experiments. Memory management mechanisms, such as dynamic memory allocations [Herter et al., 2011] and garbage collectors [Siebert, 2001], can introduce additional unpredictable overheads. On top of this, Address Space Layout Randomisation (ASLR) and the physical page allocation mechanism change the logical and physical memory layout of the application every time it is executed, potentially affecting the number of conflict misses in the CPU caches and branch misprediction rates [de Oliveira et al., 2014]. Multi-threaded applications can even force non-deterministic behaviour back on themselves, if the scheduler is not set to be perfectly repeatable, or if small timing changes alter communication patterns [Pusukuri et al., 2012]. Any I/O can have non-repeatable timings, and even changes to the environment variables between runs can shift memory and alter runtimes [Mytkowicz et al., 2009].

Reducing Experimental Noise

Past research has investigated ways to reduce measurement noise in performance experiments. Typical approaches include avoiding I/O, overriding the default scheduling policy [Pusukuri et al., 2012; Pouchet, 2012], using deterministic memory management [Pusukuri et al., 2012; Pouchet, 2012; Herter et al., 2011], or just minimising the number of active processes, including services and daemons. However, going to these lengths is not always enough or even desirable for multiple reasons. First and foremost, while these techniques do reduce noise they do not eliminate it. Therefore, multiple profile runs are still needed to determine whether noise affects the measurements significantly, which brings up the question as to how many samples are enough. Even when applied, the amount of variation might still be too high for optimisation heuristics dependent on accurate measurements [Leather et al., 2009b]. Secondly, modifications to reduce noise may do so at the expense of altering run-time behaviour, ultimately risking the possibility that the wrong heuristic is learnt. Heuristics targeting very specific, low-noise run-time environments may not match well when used in practice. For example, Curtsinger and Berger [2013] showed that the runtime variation caused by memory layout changes, such as ASLR, can dwarf the differences between optimisations. Therefore, if ASLR is disabled during training, or only a single runtime sample is taken, then an optimisation could be selected which is not helpful, in aggregate, when deployed; instead, multiple runs must be used to smooth out the effects of random layout changes. Finally, even if a low-noise environment did not actually alter the heuristic, it may prove difficult to convince companies that tuning heuristics in an environment different than their production one is acceptable.

An Adaptive Sampling Plan

Since reducing noise can be problematic for the reasons outlined in the previous subsection repeated measurements are inevitably required. This brings up the question again as to what sample size to choose; if it is too low then inaccurate information will be learnt and the model performance will suffer, whereas too many samples will mean that time is wasted with excessive, repeated profiling. The work in this chapter aims exactly at handling noise without having to reduce it, and without wasting time on unnecessary evaluations. The insight behind this research is that each additional sample, that is, each additional performance measurement for some optimisation strategy, would provide diminishing amounts of information. Indeed, that extra information will quickly reach zero if there is little experimental noise or if the samples already fit well with what is already known about the space. In other words, extra profiling runs for a decision are useful only if the results are likely to contradict what has been predicted about that decision. The experiments in the proceeding sections confirm that iterative compilation can be slowed down by using a fixed sampling plan, spending much of its time getting additional samples which provide little or no useful information.

A seminal work was presented by Wald [1944], where some theoretical foundations for the field of sequential analysis were established. In particular, the paper presented a *sequential probability ratio test* for binomial and normal distributions, as well as double dichotomies, such that a sample can be iteratively obtained for a random variable to establish some mean that can be used to either accept or reject a null hypothesis. That is, some small number of samples can be recorded and a sample mean used to accept or reject a null hypothesis, if neither action is taken then another sample is added and the same procedure attempted again. For normally distributed values, this means that one could establish, with some pre-defined allowable thresholds for incorrect classification, whether a mean is below or above a given value. This is similar to the approach taken by Leather et al. [2009b], but a problem lies in the fact that this particular test requires the standard deviation to be known. Nevertheless, sequential analysis is the name given to an approach where the number of samples are allowed to change during experimentation, and in this chapter a novel active learning technique for iterative compilation is introduced which includes sequential analysis, albeit in a more *ad hoc* form than that of Wald [1944].

More specifically, the approach taken in this work is able to more quickly produce an optimisation model, as compared to a fixed sampling plan, without sacrificing heuristic quality by profiling an application under the same optimisation decision only as long as this improves knowledge . This is achieved by taking a single sample runtime measurement for optimisations that are deemed to be most profitable to learn from, as defined by an active learner. As knowledge is built up, the algorithm is able to revisit these examples later instead of getting new ones. This happens if the algorithm determines that they are of continued interest, that is, if it appears that measurement noise has affected the data previously collected on that configuration.

To evaluate this approach models are created for 11 programs from the SPAPT suite [Balaprakash et al., 2012]. These models can predict, with low error, the runtime of a particular code given a number of optimisation options that one may want to apply, and in this way can find an optimal combination which minimises runtime. This is a variation on the commonly applied speed-up prediction technique [Cavazos et al., 2006; Dubach et al., 2007; Park et al., 2011] and is necessary since it is not possible to directly foresee a good combination from the outset as the model needs to guess relative performance before it can conclude which configuration may be best. The results show that this methodology can create high-quality heuristics on average 4x, and up to 26x, faster than a baseline approach which uses 35 samples per training example and active learning alone: where 35 was chosen as it was used in the most closely related literature— Balaprakash et al. [2013a].

The rest of this chapter is organised as follows: Section 5.1 provides a motivation, Section 5.2 describes the methodology; Section 5.3 details the validation; Section 5.4 lists results; Section 5.5 critiques this approach; and a summary is given in Section 5.6.

5.1 Motivation

The research in this chapter is based on the realisation that current procedures for creating machine learning based heuristics do not consider sample size a parameter for optimisation, but rather assume it to be a constant value fixed ahead of time. Moreover, little or no justification is ever provided for one chosen sample size over another. With active, iterative learning this need not be the case and the knowledge built up by the algorithm over time can be leveraged to adaptively select a more appropriate sample size per example, significantly accelerating training overall.

To motivate this work an examination of a simple optimisation-space was carried out. The exemplar problem was finding the best unroll factor for two loops in the `Matrix Multiplication` kernel from the SPAPT benchmark suite—specifically loops `i1` and `i2`. Compiling each training example on the machine whose specifications are given in Subsection 5.3.1, and with the `-O2` optimisation level as a baseline, the kernel was iteratively compiled multiple times, each time with a different combination of unroll factor values for the two loops. Each binary was then executed 35 times and the runtime measurements recorded.

Figure 5.1 presents the Mean Absolute Error (MAE) that would have been incurred had only one sample per configuration been taken versus 35 samples. This gives an estimated baseline for the worst error that could result in this space, as high as 4 ms (5% of the mean) for some binaries but practically zero for many. For the latter case getting even a second sample is a waste of effort. To estimate the potential speed-up that could be obtained if the optimal number of samples were known for each optimisation setting the space was iterated through again, but at each configuration samples were removed randomly from the group of 35 collected initially; this reduction continued so long as the calculated MAE remained below 0.1 ms.

Figures 5.2–5.3 show the error of this 'optimal' approach across the entire space and the number of samples that were actually needed per configuration to maintain such a small error, respectively. These figures demonstrate that there is quite considerable stochastic noise in measurements from this feature-space, as one might expect based on the previous discussion regarding noise. Hence, the number of samples needed for a low MAE varies. If the naïve, fixed sampling plan of 35 is taken then $35 \times 30 \times 30 = 31,500$ individual executions are needed, whereas if it was already known ahead of time what sample size would be required then a maximum error of only 0.1 ms is incurred at a cost of just $15,131$ runs—approximately half.

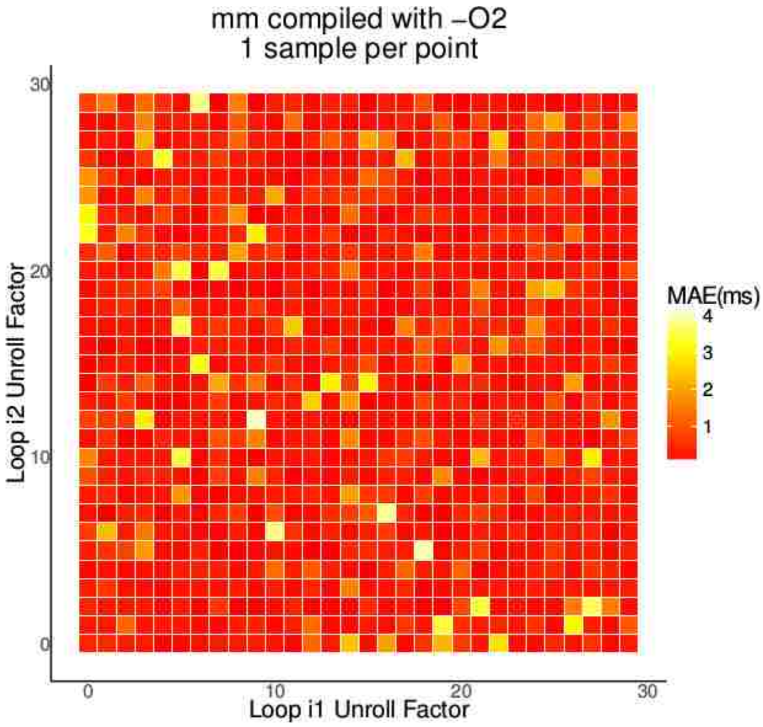


Figure 5.1: this decision-space represents different unroll factors for two loops of the `Matrix Multiplication` kernel of SPAPT, where the relative colour indicates the Mean Absolute Error (MAE) for a sample size of one, assuming 35 samples approximates the population mean.

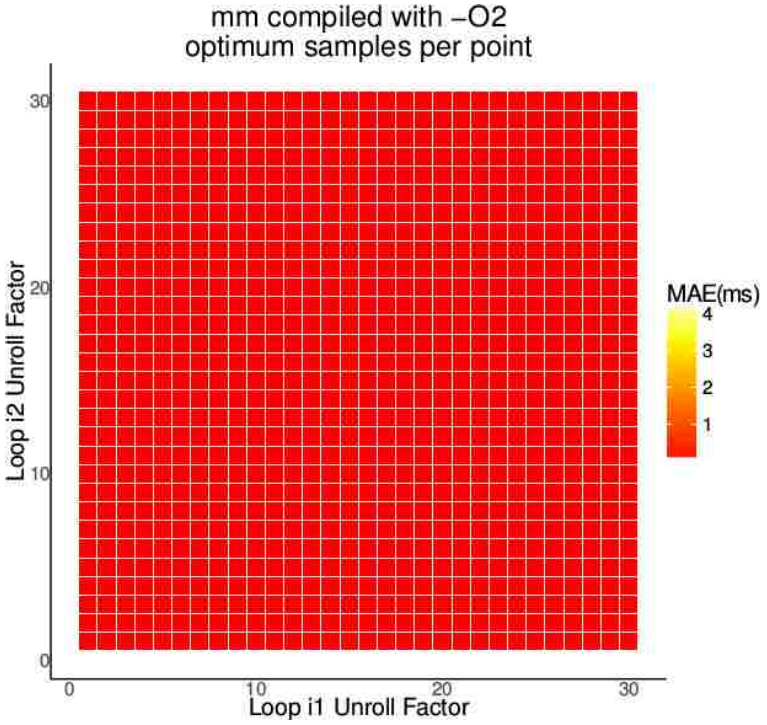


Figure 5.2: similar to Figure 5.1 except this graph shows the comparison between an ‘optimal’ number of samples versus 35.

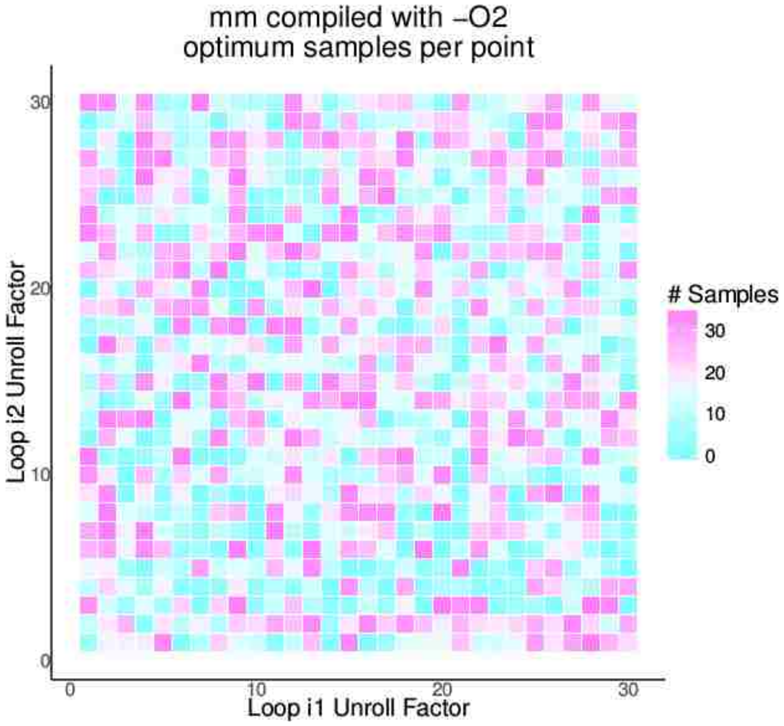


Figure 5.3: the optimal number of samples per configuration in the decision-space.

Although this example has not used machine learning directly, through simple deduction it can be established that if one had perfect knowledge, and knew how many samples per configuration were needed, then training time could be reduced. That is, since perfect knowledge reduces samples, a training example is composed of a number of samples, more samples cost more time, then perfect knowledge reduces time of training.

This motivating experiment has started out with 'perfect' information about each configuration in the decision-space and removed samples until the average runtime starts to deviate significantly from that initially calculated, but a real sequential analysis approach must work in the opposite direction. It must start from zero information about each configuration and add samples until the distance between the average runtime and the true population mean is within some threshold. Unfortunately, this distance cannot be known, but the intuition behind this research is that it may be approximated by looking at what information has been gathered from the rest of the space.

Consider Figure 5.4, where the `i1` loop in the `adi` benchmark is unrolled some random number of times and a single sample runtime measurement taken per executable. Despite the noise there appears to be a pattern identifiable to the human eye: a plateau starting at 2.1 s which climbs and levels off at 3.1 s around a loop unroll factor of 10. It is postulated, and it will be demonstrated that, points in areas where the pattern is clear and which fall nicely within that pattern are more likely to be nearer to their respective population means. The points where more samples are needed are the rest. In other words, from looking at the spatial locality of runtime sample means across an optimisation-space it is possible to guess which ones may be incorrect and need more samples, because they tend to stick out.

A sample runtime may not always fit to expectations if the mean runtime of the other training instances are badly estimated. The assumption in this work is that the runtime of training examples with less than optimal sample counts will be noisy, but that the model should be able to accommodate for this by taking neighbouring data into consideration to approximate the target value in question. This was inspired by Fermi estimation, where numerous roughly correct estimates combine in such a way as to cancel out any over or under estimation [Weinstein and Adam, 2008]; indeed, machine learning algorithms often explicitly generalise on purpose anyway to avoid overfitting [Hastie et al., 2009]. Hence, if a sample runtime is relatively far from where it is expected to be in the space there are four possibilities, either the model is poorly fit and the sample runtime is correct or the model is accurate and the sample runtime

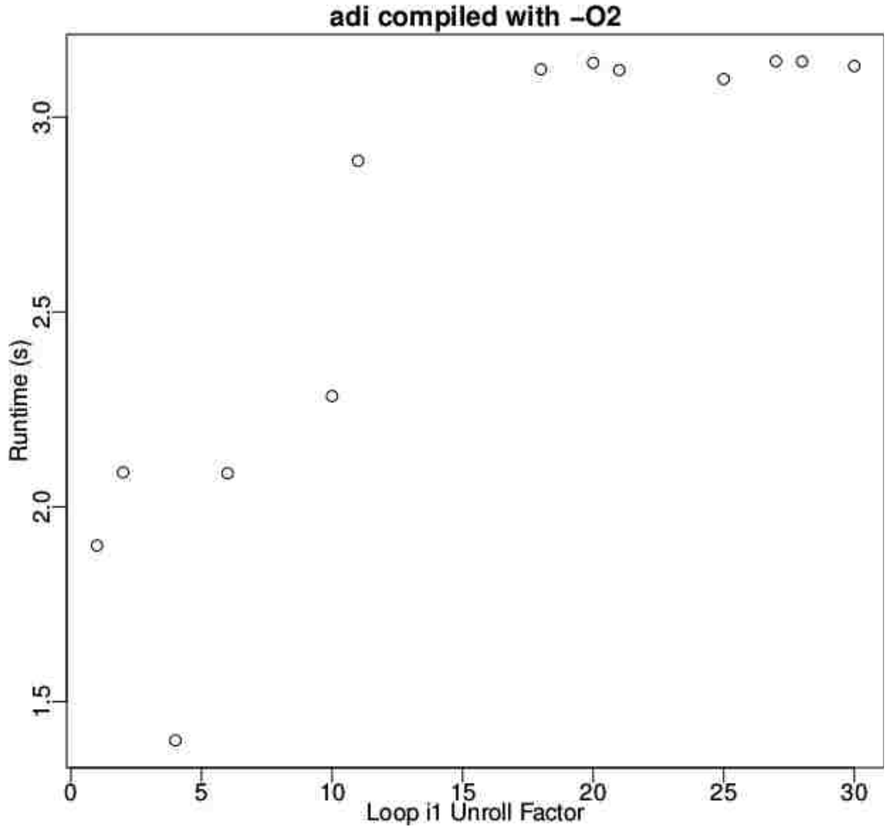


Figure 5.4: runtime versus unroll factor for a loop of `adi` when using a sample size of one. A relationship between unroll factor and runtime is relatively clear despite the noise: i.e. stable around $2.1\,\mathrm{s}$ until 10 where it climbs steadily and plateaus at $3.1\,\mathrm{s}$ for high factors of loop unrolling.

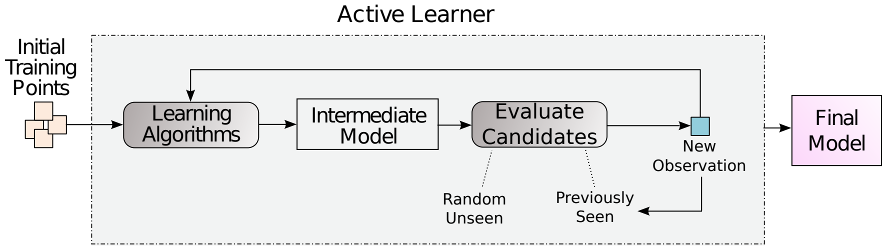


Figure 5.5: an overview of the active learning approach. An initial model is seeded with some high-quality data. A single training example is then selected from a set of candidate examples. Data is collected and fed back into the algorithm. The process repeats until some completion criterion has been satisfied. Contrary to existing active learning approaches, potentially noisy training data is collected one sample at a time. Visited training examples remain in the candidate set and can be revisited if getting more samples for that optimisation is more profitable than exploring other, new parts of the space.

is wrong, or they are both right or both wrong—in any case increasing the number of samples for such a point will increase knowledge. Note that this technique will not work as intended if the *bias* of the model fits the data poorly, but in such a scenario there is a larger problem since the heuristic will not fit the collected information irrespective of the sampling approach taken.

5.2 Methodology

The work in this chapter introduces a novel approach to active learning which is broadly applicable. While traditional active learning is used to reduce only the number of training examples, this methodology wishes to reduce the number of samples *per example*. Previous research in this area has ignored sequential analysis, presumably because many implementations of active learning are greedy so learning from noisy data on purpose would lead to poor conclusions being drawn from the intermediate models [Settles, 2013]. For example, as demonstrated in Subsection 4.5.1, using QBC with a noisy oracle can steer the search towards less informative areas of the decision-space, significantly reducing any heuristic's effectiveness; Subsection 5.2.3 explains how this problem has been overcome in this work.

5.2.1 Sequential Analysis

Traditionally in active learning the training set (the set of examples already seen) and the candidate set, a random subset of all examples that could be learnt from next, are kept disjoint. This makes sense because the information contained in the training set is assumed to be of good quality; each example will have been evaluated some fixed number of times to ameliorate the effects of noise, hence, there is little to be gained from revisiting those examples again. However, as has been demonstrated in Section 5.1, a fixed sample count can be overly conservative and wasteful.

In order to modify active learning to incorporate sequential analysis the algorithm is altered such that the sample size fetched during labelling is set to one. In cases of noisy data, the algorithm needs to be able to revisit previously compiled programs so training examples that have been explored remain temporarily in the candidate set—see Figure 5.5. That is to say, at each iteration of the learning loop the algorithm will consider not only getting a new example but also whether it is more profitable to try an old one again, similar to the multi-armed bandit problem in the field of mathematics [Robbins, 1952; Berry and Fristedt., 1985]. This is possible because the particular model used in this work provides a scoring function which quantifies the uncertainty the model has about each point in the space, based on the knowledge it has at that time. As knowledge is gained, given the shape of the intermediate model, noisy examples or examples in complex areas of the decision-space will begin to stick out, and will be more likely to be visited. In either case, with each iteration of the training loop the configuration that is estimated to provide the most amount of information will be sampled from.

The procedure for this work is summarised as pseudo-code in Algorithm 2. The algorithm begins by constructing a model M with n_{init} training examples which have been randomly chosen from all potential examples F as a seed. To generate this initial model a fixed number of observations n_{obs} are collected for each training example to give the active learner a quick and accurate look at the search space. The learning loop then proceeds whilst the completion criterion has not been met. In this implementation the criterion is set to a fixed number of training instances but could have been based on, for example, wall-clock time or some estimate of the error in the final model established through cross-validation [Hastie et al., 2009]. Within each iteration of the loop the candidate set C combines n_c random points which have never been observed before and those examples which have been seen previously but less than n_{obs} times. The next

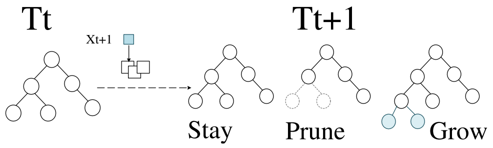


Figure 5.6: this diagram shows the three potential updates that are stochastically applied to the Dynamic Tree upon receiving a new training example (x_{t+1}, y_{t+1}). The tree either remains unchanged, a leaf node is pruned back so that the parent of the leaf becomes a leaf itself, or grown such that two new children divide the relevant subspace.

training example x is chosen based upon its predicted usefulness (see Subsection 5.2.3) and its runtime y measured once. The model is then updated as well as the required data structures. It should be noted that this algorithm is easily parallelised by selecting multiple training examples per loop iteration instead of just one [Balaprakash et al., 2013a] but as this is orthogonal to this work, and would have complicate analyses, the algorithm was made purposefully sequential.

5.2.2 Dynamic Trees Model

In regression problems where an estimate of uncertainty of a prediction is required the collective wisdom would arguably have been to use a Gaussian Process (GP) [Rasmussen and Williams, 2006]. GP inference, however, is relatively slow with $O(n^3)$ efficiency for n examples. This is problematic, particularly in active learning, since each time something new is learned a model needs to be constructed and evaluated. A more efficient algorithm, which is leveraged in this work, is the relatively new Dynamic Tree, which is based on a classical decision tree model [Breiman et al., 1984] with modifications to include Bayesian Inference [Chipman et al., 1998, 2002]. The advantages of the Dynamic Tree for this work are

- its ability to evolve over time as new data come in, without reconstructing the model from the ground up with each iteration;
- its estimation of uncertainty for any given point in the space, like a GP but without the overhead;
- and its avoidance of overfitting to the training data, which is vital since the algorithm is learning potentially noisy information.

Full details can be obtained from the article by Taddy *et al.* [Taddy et al., 2009], but a brief overview of how the Dynamic Tree model works is as follows. The static model used within the Dynamic Tree framework is a traditional decision tree for regression applications; a set of rules recursively partitions the search space into a set of hyper-rectangles such that training examples with the same or similar output value are contained within the same leaf node. The Dynamic Tree changes over time, when new information is introduced, through a stochastic process thereby avoiding the need to prune the tree once learning is finished to avoid over-fitting. At time t, a tree T_t is derived from the training data $(x,y)^t$ where x gives the training instance features and y the associated target value. When new data (x_{t+1}, y_{t+1}) arrives, an updated tree T_{t+1} is created, identical to T_t except that some mechanism has been randomly chosen from three possibilities—see Figure 5.6. The leaf node $\eta(x_{t+1})$ containing x_{t+1} either (1) remains completely unchanged; (2) is pruned, so that the parent of $\eta(x_{t+1})$ becomes a leaf node; or (3) is grown, such that $\eta(x_{t+1})$ becomes an internal node to two new children. The choice of transformation is influenced by y_{t+1} in a posterior distribution. This posterior distribution depends upon the probability of y_{t+1} given x_{t+1}, T_t, and $[x,y]^t$. Hence, the Dynamic Tree is more resilient to noisy data than other regression techniques.

5.2.3 Quantifying Usefulness

The most crucial part of the active learning loop is estimating which training example from within the pool of potential candidates C would be most profitable to learn from next. The `dynaTree` package for R [Gramacy and Taddy, 2017] that is used in this work offers two heuristics out of the box, both well cited in the literature for regression problems. The first was presented by MacKay [1992] and selects the candidate where the estimated variance of the output is maximised relative to the other candidates. The second heuristic by Cohn [1996] selects the candidate it calculates will most reduce the predicted average variance across the space. To put this in a more accessible way, it selects the example it believes will enable the model to best fit what it is already seeing, in an attempt to reveal key information that it may be missing. Both are competitive with each other, and both solve the greedy search problem discussed previously. Although the latter is more computationally intensive than the former—$O(|C|^2)$ versus $O(|C|)$—it is used here to provide a scoring function since it handles heteroskedasticity which is assumed for increased robustness.

5.3 Experimental Setup

The novel approach presented in this chapter was evaluated by examining how efficiently models are constructed to solve a classical but complex compilation problem. In particular, the problem considered in this work involves finding the optimal set of compilation parameters for a program. The set of parameters includes loop unrolling, cache tiling, and register tiling factors, where each parameter has a range of possible values unique to each loop in the source code. The combination of these parameters results in a large model which predicts program-specific runtimes. By using machine learning it is possible to find a good performing configuration for minimal runtime without having to compile and profile each possible combination in the optimisation-space.

5.3.1 Platform and Benchmarks

Platform The server used during experimentation was the same as in Chapter 4, but for convenience the relevant specifications are a machine running OpenSuse v12.3, with an Intel Core i7-4770k CPU running at 3.4 GHz, which contains 16 GB of RAM, and compiles using GCC v4.7.2. In terms of specific environment, the time of each application run was measured using the C library function `clock_gettime()`. As in previous iterative compilation literature, the machine was restricted to a single user and did not have any processes running other than those enabled by default under a standard OS installation. No further steps to reduce experimental noise, such as pinning threads to processor cores or using a non-standard memory allocator were used. This was so as to avoid potentially creating an artificial environment that might alter findings, as discussed previously in the introductory text to this chapter.

Benchmarks Eleven benchmark applications were taken from the SPAPT suite to be used for evaluation, where these codes are based on high-performance computing problems such as stencils and dense linear algebra. These particular 11 were selected because they were the only applications included in a dataset used for initial prototyping, kindly provided by Dr. Balaprakash of Argonne National Laboratory which he collected for his closely related research [Balaprakash et al., 2013a]. Each problem in the SPAPT suite is defined by three primary variables—kernel, input size, and tunable configuration. The tunable parameters are further broken down into a number of integer and binary values, with these values representing which optimising code transformations are applied. In this evaluation binary flags and input size were not considered so that a fair comparison could be made with Balaprakash et al. [2013a]. The precise size of each search space is given in Table 5.1.

5.3.2 Active Learning Settings

For each kernel the goal was to produce a model capable of estimating mean serial code runtime when a given set of compiler optimisation settings are used. To this end, the following parameters were used in the learning algorithm.

With respect to Algorithm 2, the learning process is started by seeding the training set with five random examples n_{init}, where for each example 35 samples n_{obs} were obtained to calculate a mean runtime. The Dynamic Tree model is created using the R `dynaTree` package [Gramacy and Taddy, 2017] with an entirely default configuration except that the number of particles N is set to $5,000$, where this value was found to be more effective at producing high-quality heuristics through 10-fold *cross-validation* [Hastie et al., 2009]. During each iteration of the learning loop 500 random and new candidate training instances are considered n_c.

The completion criterion for all experiments was set such that the maximum size of the training set n_{max} did not exceed $2,500$. All experiments were repeated 10 times with new random seeds. The results reported in Section 5.4 are all averaged over those 10 experimental runs.

5.3.3 Evaluation Methodology

Baseline Approach Most (if not all) machine learning in compilers literature uses a simple constant sampling plan [Moss et al., 1997; Monsifrot et al., 2002; Stephenson et al., 2003], where the number of samples for each training example is fixed ahead of time. Different sizes have been chosen in the past, for example, Grewe et al. [2013a]; Emani et al. [2013] uses 10, Grewe et al. [2011] uses 20, Petoumenos et al. [2015] uses 80, and Balaprakash et al. [2013a] uses 35. Based upon classical methodologies, two techniques are considered to be in competition with the one presented here. For both, a fixed number of samples are used to calculate the mean runtime for each training instance, and the candidate set is kept disjoint from past training examples. The first technique uses the average runtimes calculated over 35 samples, as in Balaprakash et al. [2013a]. The second records a single timing per example to estimate the mean runtime. In this way, this novel approach can be compared to both very low and relatively high-confidence estimates of the runtime for each configuration. Furthermore, to provide the best evaluation possible the methodology of the active learning approach in Balaprakash et al. [2013a] is also followed, in that the same benchmark suite, model, and error metric are used.

Description of the Datasets To collect the data for the experiments each program was profiled with $10,000$ distinct, randomly-selected configurations. For each configuration the mean runtime, as determined by averaging 35 separate execution times, as well as its compilation time were recorded. Per experiment, exactly $7,500$ examples were randomly marked for possible training whilst the remaining $2,500$ constituted a test set. The feature values of each data point, which is to say the values which make each example distinct from one another, were all normalised through scaling and centring to transform them into something similar to the Standard Normal Distribution: a common practice in machine learning work where features are not all on comparative scales.

Evaluation Metric The efficiency of model construction and, more specifically, the evolution of model error over training time for each one of the 11 benchmarks, using the three different sampling methods, is examined to validate this research. In particular, the error of the models produced by each approach is calculated using the Root-Mean-Squared Error (RMSE) of the predicted runtimes of the test set instances.

Note that the measurement of training time in each experiment is defined as the cumulative compilation and runtime of any executables used in training. The overhead of updating the Dynamic Tree was not measured as it is only a small part of the overall training overhead, is near constant for all evaluated approaches, and would change in proportion to the average benchmark runtime.

5.4 Experimental Results

This section begins by proving that a combination of active learning and sequential analysis can successfully and substantially accelerate heuristic construction; specifically, that the cost of profiling can be reduced by as much as 26x, as compared to a baseline approach that uses 35 observations per training example for active learning alone. This section then discusses the results for a few representative benchmarks in detail, and how these relate to the behaviour of the others.

5.4.1 Overall Efficiency Savings

To evaluate the overall efficiency of the proposed methodology versus a baseline active learning approach [Balaprakash et al., 2013a] the time needed for both techniques to first reach a common lowest average RMSE was measured. The evaluation was performed in this way to get an idea of the speed-up one could achieve whilst producing the best heuristic possible under the circumstances. To ensure a fair evaluation a fixed point at which speed-ups would be calculated was required to be set *a priori* to prevent biasing the results in the favour of the presented methodology, since speed-up can change dramatically over the course of a single training run.

Table 5.1 shows for each benchmark what this lowest error was and how many seconds it took to collect the profiling data needed to reach it for the competing methods on average. Whereas, Figure 5.7 presents graphically the acceleration achieved by this novel approach.

In all but one benchmark the proposed algorithm is technically faster at reaching the lowest average error as defined by the pre-set evaluation technique. Specifically, this new methodology was able to reduce the overhead for 10 benchmarks by 4x on average, and up to 26x. The only benchmark in which this approach failed to reduce the overhead was `adi`. However, the difference in RMSE between the two techniques is comparable for that benchmark: within a few thousandths of a second on average. That said, from Figure 5.11 it is clear that the definition of speed-up in this case is insufficient. Which is to say, although technically the novel approach is faster at reaching the lowest common error between the two approaches the graph shows that using all samples is actually superior. In hindsight a better evaluation methodology would have included a means for coping with this case.

Algorithm 2 an active learning algorithm modified to reduce the number of samples, where F contains all optimisation configurations in the space, n_{init} and n_{max} specify the initial and total number of training examples to record, n_c the number of candidates per iteration, and n_{obs} the number of samples thought to be needed to reduce the affects of noise in the output/performance measurements.

```
procedure ActiveLearn($F, n_{init}, n_{max}, n_c, n_{obs}$)
    $X \leftarrow$ sample($F, n_{init}$)
    $Y \leftarrow$ getObservations($X, n_{obs}$)
    $M \leftarrow$ dynaTree($X, Y$)
    $D \leftarrow \emptyset$
    for $i = n_{init}, n_{max}$ do
        $C \leftarrow$ sample($F - X, n_c$)
        for all $k \in$ keys($D$) do
            if $D[k] < n_{obs}$ then $C \leftarrow C \cup k$
            end if
        end for
        $x \leftarrow \emptyset$
        $v_{min} \leftarrow$ MAX_DOUBLE
        for all $c \in C$ do
            $v \leftarrow$ predictAvgModelVariance($M, c$)
            if $v < v_{min}$ then
                $v_{min} \leftarrow v$
                $x \leftarrow c$
            end if
        end for
        $y \leftarrow$ getObservations($x, 1$)
        $M \leftarrow$ updateModel($M, x, y$)
        $X \leftarrow X \cup x$
        if $k \in$ keys($D$) then
            $D[k] \leftarrow D[k] + 1$
        else
            $D[k] \leftarrow 1$
        end if
    end for
    return $M$
end procedure
```

Table 5.1: lowest common RMSE achieved by both competitive approaches, profiling time needed to reach this error level, and training speed-up for all 11 benchmarks

benchmark	search space	lowest common RMSE	cost of the baseline (sec)	cost of this approach (sec)	speed-up
adi	3.78×10^{14}	0.087	2.62×10^{4}	9.08×10^{4}	0.29
atax	2.57×10^{12}	0.097	3.33×10^{3}	2.39×10^{2}	13.93
bicgkernel	5.83×10^{8}	0.065	1.35×10^{4}	3.76×10^{3}	3.59
correlation	3.78×10^{14}	0.589	57.46	8.13	7.07
dgemv3	1.33×10^{27}	0.067	1.75×10^{2}	7.44	23.52
gemver	1.14×10^{16}	0.342	2.99×10^{3}	1.15×10^{2}	26.00
hessian	1.95×10^{7}	0.006	5.76×10^{3}	1.56×10^{3}	3.69
jacobi	1.95×10^{7}	0.076	3.04×10^{3}	8.57×10^{2}	3.55
lu	5.83×10^{8}	0.013	2.57×10^{3}	7.09×10^{2}	3.62
mm	3.18×10^{9}	0.042	9.87×10^{4}	8.89×10^{4}	1.11
mvt	1.95×10^{7}	0.002	2.59×10^{3}	2.20×10^{3}	1.18
				geometric mean	**3.97**

Table 5.2: this table gives an indication of the spread of the variance and 95% confidence interval relative to the mean for all benchmarks tested; the latter is given for two sample sizes, 5 and 35 observations. The values shown illustrate that although noise can be low for many benchmarks it is high for others.

benchmark	**variance**			**35-sample 95% C.I. / mean**			**5-sample 95% C.I. / mean**		
	min	**mean**	**max**	**min**	**mean**	**max**	**min**	**mean**	**max**
adi	8.44×10^{-10}	2.34×10^{-3}	0.14	4.10×10^{-6}	2.25×10^{-3}	0.05	2.77×10^{-6}	0.01	0.16
atax	7.54×10^{-10}	9.72×10^{-5}	0.03	2.22×10^{-5}	2.31×10^{-3}	0.06	1.79×10^{-5}	0.01	0.25
bicgkernel	2.06×10^{-10}	1.06×10^{-4}	0.05	1.17×10^{-5}	1.52×10^{-3}	0.07	1.02×10^{-5}	4.64×10^{-3}	0.29
correlation	2.27×10^{-10}	0.42	8.02	2.13×10^{-5}	0.03	0.34	4.42×10^{-6}	0.13	2.41
dgemv3	1.15×10^{-9}	5.60×10^{-5}	0.03	3.31×10^{-5}	2.25×10^{-3}	0.08	2.24×10^{-5}	0.01	0.28
gemver	1.19×10^{-9}	5.91×10^{-3}	0.47	1.18×10^{-5}	4.81×10^{-3}	0.10	9.34×10^{-6}	0.02	0.42
hessian	2.35×10^{-11}	1.03×10^{-6}	1.99×10^{-4}	3.89×10^{-5}	1.33×10^{-3}	0.06	1.63×10^{-5}	4.15×10^{-3}	0.24
jacobi	2.54×10^{-10}	1.20×10^{-4}	0.09	1.32×10^{-5}	1.29×10^{-3}	0.09	4.12×10^{-6}	3.83×10^{-3}	0.39
lu	1.84×10^{-11}	8.45×10^{-7}	1.09×10^{-4}	2.03×10^{-5}	6.89×10^{-4}	0.02	5.76×10^{-6}	2.10×10^{-3}	0.11
mm	2.76×10^{-10}	4.87×10^{-6}	1.31×10^{-3}	2.26×10^{-5}	7.44×10^{-4}	0.02	1.36×10^{-5}	2.37×10^{-3}	0.09
mvt	9.97×10^{-12}	1.07×10^{-8}	7.87×10^{-6}	6.29×10^{-5}	8.28×10^{-4}	0.03	3.98×10^{-5}	2.44×10^{-3}	0.11

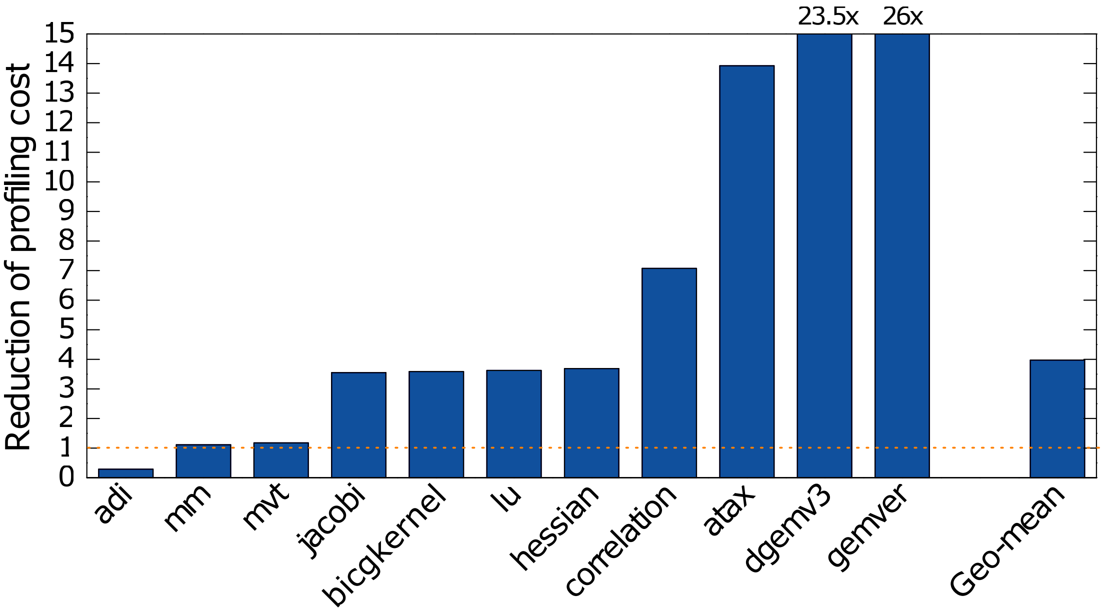


Figure 5.7: this bar chart shows the average reduction of profiling overhead of the proposed approach compared to a baseline.

5.4.2 Per Benchmark Performance

This subsection presents the experimental findings in greater detail. Figures 5.8–5.18 show the RMSE against evaluation time (cumulative profiling and compilation cost in seconds) averaged over 10 runs for all benchmarks. To make a fair comparison each graph shows the range of time over which all three sampling plans are simultaneously active in processing up to 2,500 training samples. What follows is a qualitative summary of these results.

adi Figure 5.8 gives error against time for the three different sampling techniques evaluated in this work for the `adi` benchmark. It seems self-evident that there is some considerable noise in the underlying data since a single sample per training example plateaus in error fairly quickly and cannot achieve the same results as the other two sampling plans. Although the variable sampling approach is also unable to keep up with a high, fixed number of samples per example it does achieve comparably low error throughout.

atax, bicgkernel The data for benchmark `atax` in Figure 5.9 is quite different to that in Figure 5.8 and appears to represent a case where the underlying noise in performance measurements is relatively low. This is exemplified by the fact that one

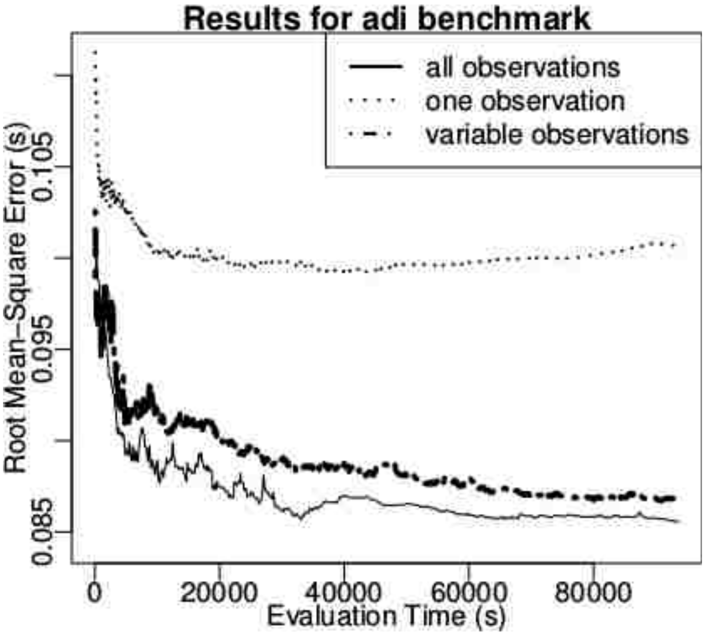


Figure 5.8

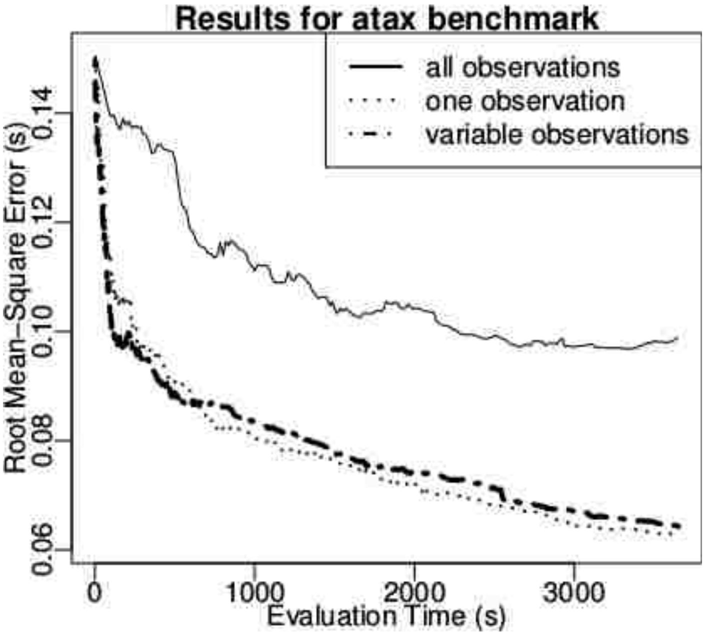


Figure 5.9

sample per unique instance is enough to do well, and indeed the adaptive sampling technique presented in this chapter appears to detect this; compare these plot-lines to the 35-samples approach and it its clear that substantial time can be saved through sequential analysis. Figure 5.10 shows similar results for `bicgkernel`.

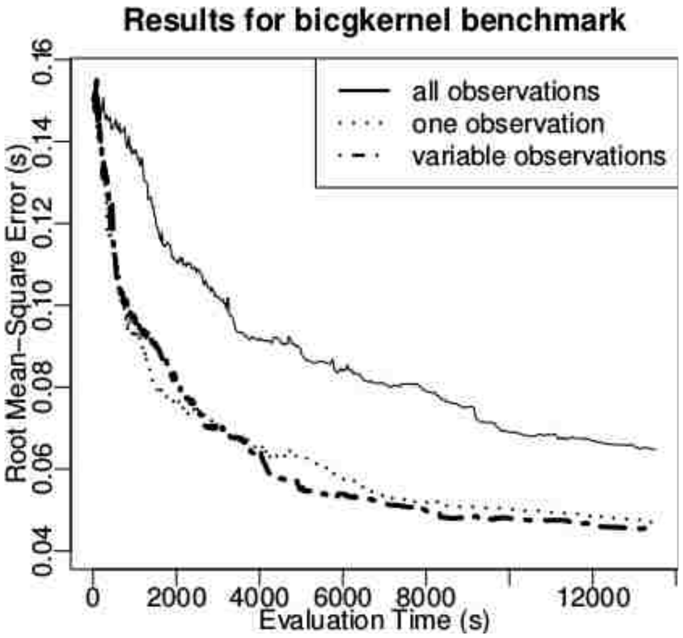


Figure 5.10

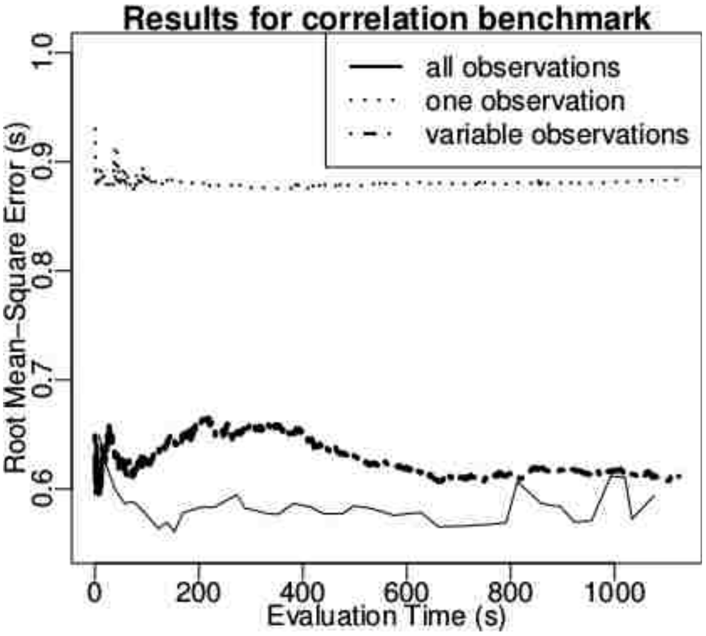


Figure 5.11

`correlation` Figure 5.11, showing the results of the `correlation` benchmark, is interesting since the error remains high irrespective of the sampling technique used. As in Figure 5.8, there must be noise present since one observation performs worst. A variable approach is not quite as good as using a large number of samples per train-

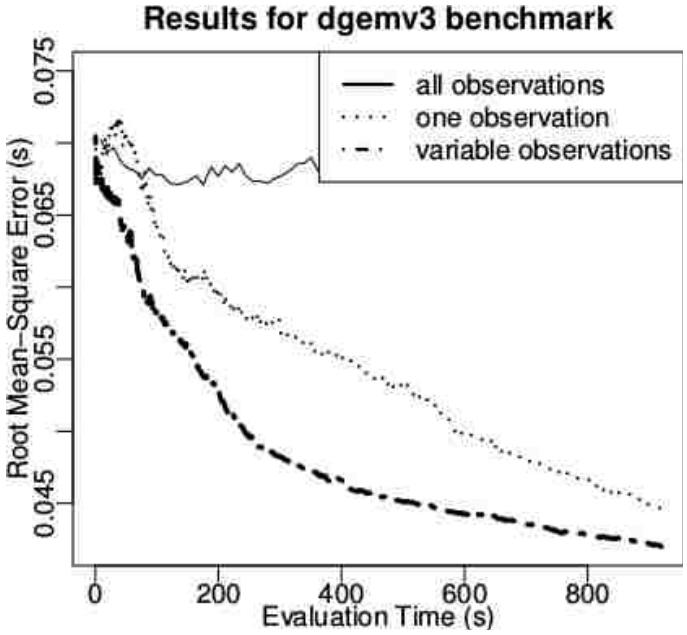


Figure 5.12

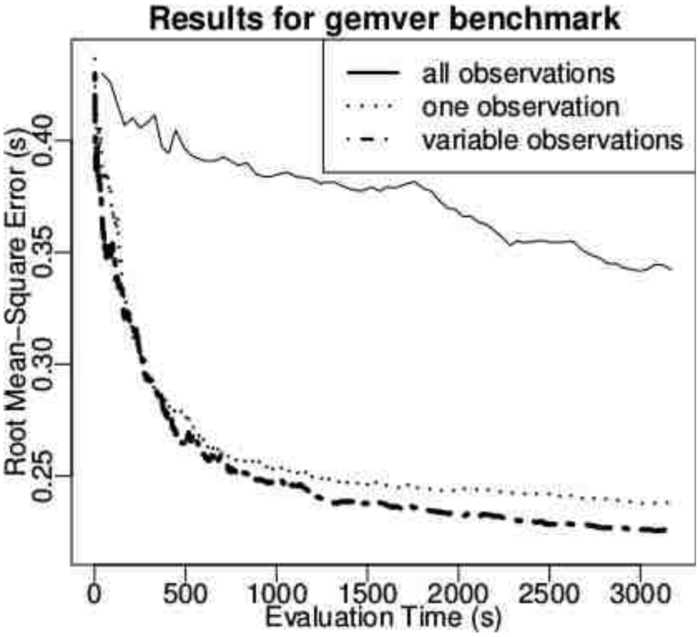


Figure 5.13

ing example but is competitive and within a few hundredths of a second in terms of average RMSE by the end of the displayed time.

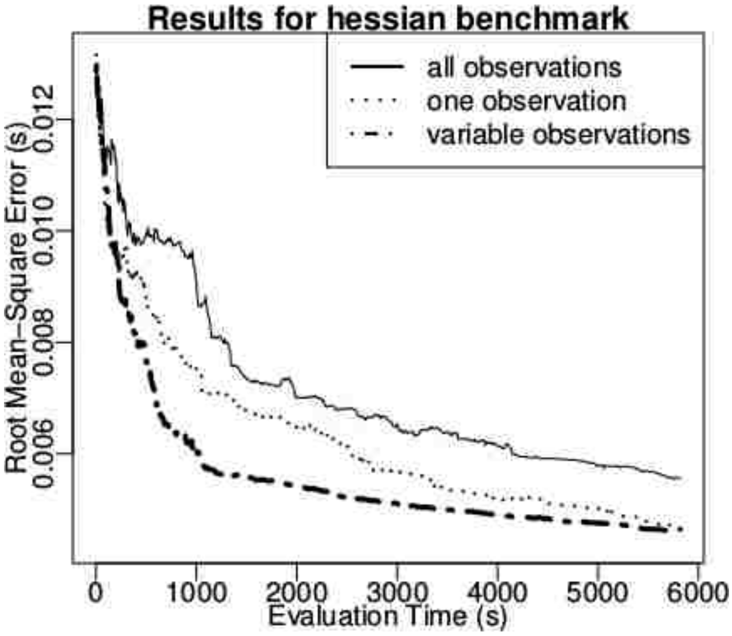


Figure 5.14

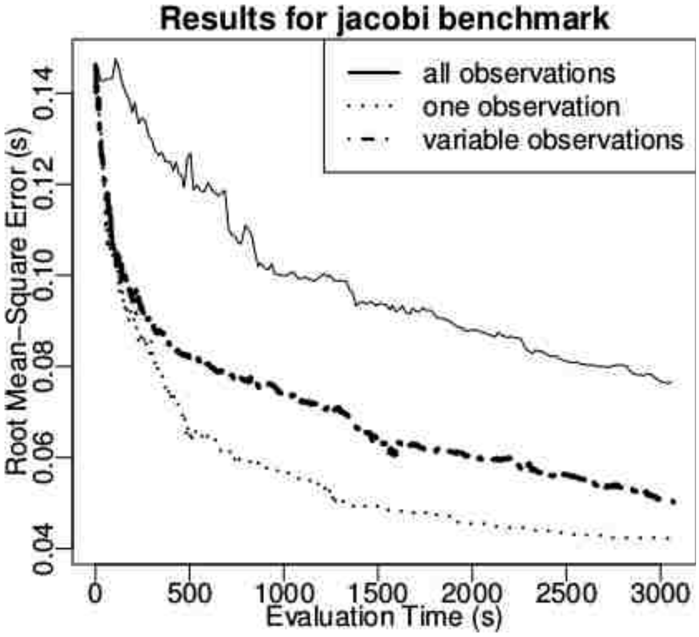


Figure 5.15

dgemv3, gemver, hessian In Figure 5.13 the variable approach is much faster than the classical method and the simple but potentially noisy variant, similarly for the results of dgemv3 and hessian—Figures 5.12 and 5.14, respectively.

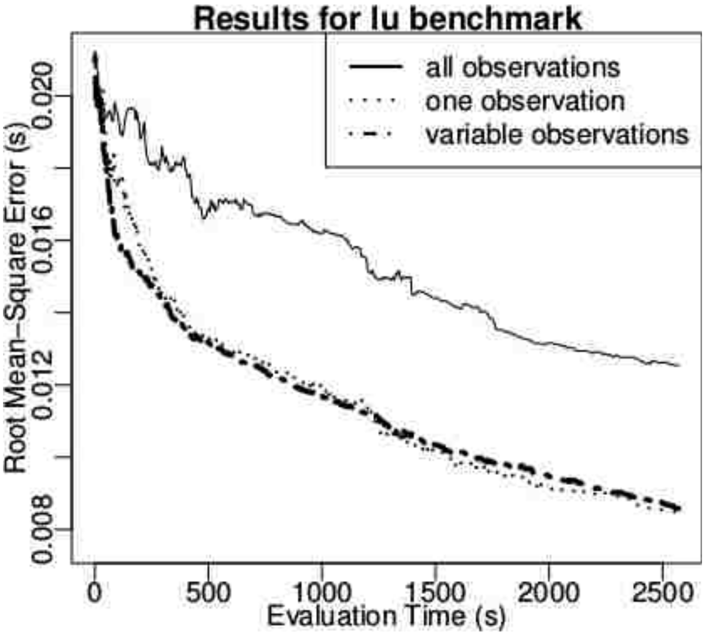


Figure 5.16

jacobi, lu, mm, mvt The data for the `jacobi` benchmark (Figure 5.15), which is also generally representative of `lu` (Figure 5.16), are interesting since they show the adaptive algorithm in this chapter to be slightly too cautious but still much more efficient than a fixed sampling plan. The `mm` benchmark gives a graph akin to that of `mvt`, showing a variable sampling approach as giving slight speed-ups over the classical methodology, see Figures 5.17–5.18.

Table 5.2 details the distributions of the runtimes measured during these experiments as well as the spread of the variance and confidence intervals relative to those mean runtimes. The level of noise across this set of benchmarks varies across applications. Moreover, the variance is not constant across all parts of the space for even a single benchmark in isolation: some parts of the space suffer from extreme noise and others are comparatively noiseless. An adaptive algorithm such as the one proposed in this work is necessary to make the best of these conditions.

For `adi`, where the speed-up runs counter to expectations, a longer experiment was carried out but the outcome did not change. Thus, it is believed that the relatively poor performance for `adi` is due to the shape of the noisy regions in this space.

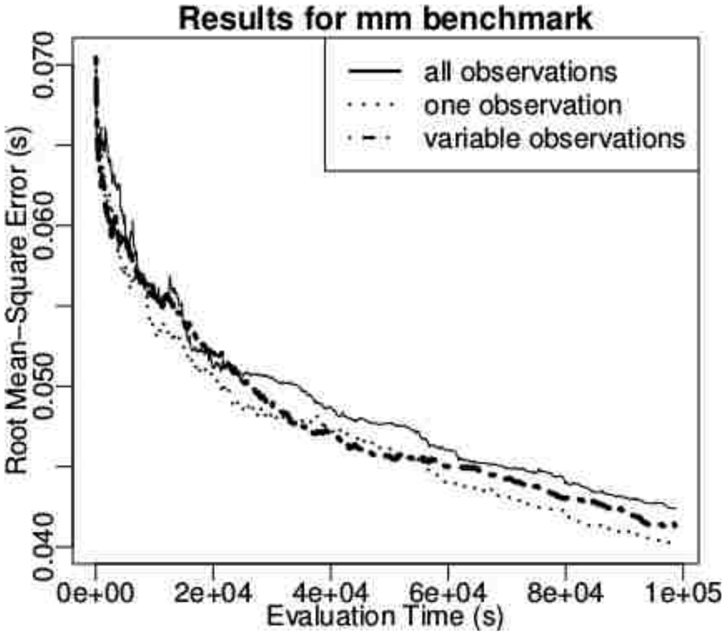


Figure 5.17

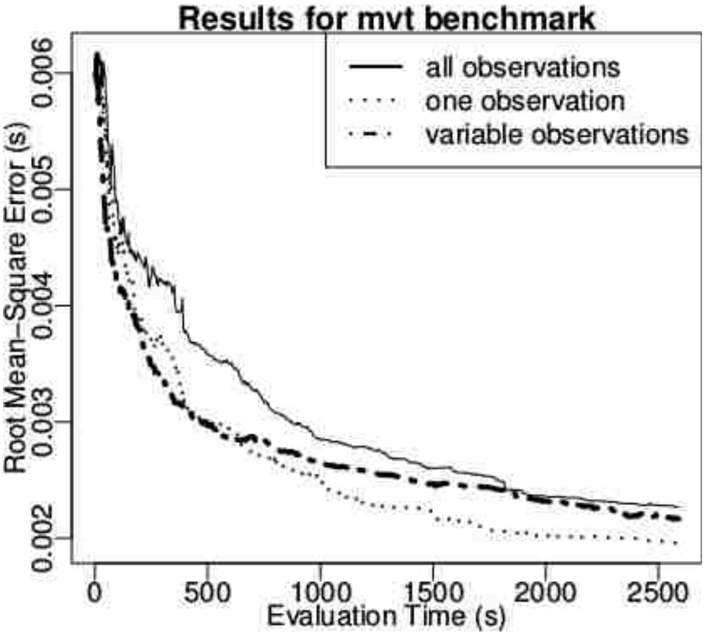


Figure 5.18

5.5 Discussion

Upon reflection, two criticisms of this work are outlined and discussed in the following subsections; namely, that the methodology used is sequential, not parallel, and that the baseline number of samples is 35, which some may consider to be excessive.

5.5.1 Serialising the Learning Algorithm

A fair criticism that could be made about this work is that although passive learning can be inefficient it does have the advantage of being implementable in an embarrassingly parallel fashion. Therefore, although active learning has been demonstrated as being faster for all but one of the benchmarks in this chapter it could be argued that the proposed technique may actually slow things down overall since it necessitates some sequential processing. This is true, at least in part, but could be countered by using one of the previously researched methods of performing *Batch Active Learning* [Settles, 2013], where the basic idea is that instead of selecting one example from the candidate set on which to learn, per loop iteration, m examples can be chosen instead. Indeed, the active learning work compared against throughout this chapter uses batch learning to parallelise the learning process [Balaprakash et al., 2013a]. It is only for simplicity (and expediency) of experimentation and analysis that this approach was not followed in the preceding algorithm presented in this chapter, but in hindsight its inclusion would have made for a more holistic work.

5.5.2 Excessive Samples in the Baseline

A second criticism that could be made of this work is that the choice of 35 samples per training example to compare against would appear somewhat arbitrary, and may unnecessarily hinder the baseline approach. Indeed, this was a concern during experimentation. However, there is both a qualitative and a quantitative defence to this choice.

Qualitatively, as the introduction to this chapter has already mentioned, it is difficult, if not impossible, to know how many samples one should use in a particular environment *a priori*, so any value one sets will be inherently arbitrary; moreover, adjustments to the environment could change the heuristic in such a way as to make it sub-optimal, since by definition it is learning under ‘laboratory’ conditions that which must be applied in the ‘real world’. Furthermore, it is debatable, for this reason, whether companies would feel comfortable learning in an environment which is so dissimilar from production.

Quantitatively, there is no statistical criterion that will sufficiently determine the correct number of samples that should be taken in an experiment at the outset, with no data yet gathered. Instead, *post hoc* analysis can be performed, for example, by calculating the ratio of the confidence interval to the mean and rejecting that sample count as insufficient if that ratio breaches some pre-defined threshold. Typically this validation is not presented in papers, if it is done at all. When it is done the standard procedure appears to be to use the 95% confidence interval and a 1% $CI/mean$ threshold. The adaptive approach in this research is compared against a constant sampling plan of 35 samples as that is what is used in the closest comparable work [Balaprakash et al., 2013a]. Even though this could appear excessive 35 samples is actually not always enough. Which is to say, across the evaluated benchmarks even though on the majority of examples there was often very little noise a significant number did experience it. In terms of numbers, fully 5% of examples broke the 1% $CI/mean$ threshold. When a more generous threshold of 5% was chosen instead 0.5% failed. Moreover, with fewer samples the problem is worse. At five samples 3.3% fail the more generous threshold of 5%; and at two samples (the minimum required to perform any statistical calculation) 5% fail. This finding is corroborated by Leather et al. [2009b], in which samples are taken until a threshold is met, where it was discovered that for timing small code sequences it is sometimes necessary to take hundreds of samples.

5.6 Summary

Concluding this chapter, the results here have demonstrated that it is possible, and indeed substantially profitable, to employ a sequential learning technique alongside active learning when generating heuristics for optimising compilers. In particular, the headline results for this implementation are that training has been accelerated by 4x on average, and up 26x, when compared to an active learning approach alone. This means that the process of automatically producing heuristics for disparate architectures need not be as expensive as is presently the case, which will go some way to resolving the problem of out-of-date optimisation strategies in currently popular compilers.

Chapter 6

Active Learning with Active Feature Acquisition

This chapter proposes a methodology which can accelerate one of the more popular methods used to produce machine learning based heuristics for program optimisation; in particular, where hardware performance counters are used as features to characterise a code, such as in Cavazos et al. [2007]; Dubach et al. [2009]; Wang and O'Boyle [2009]; Chen et al. [2010], and Park et al. [2013].

Hardware performance counters are a set of special registers built into a processor which can be used to record how many times a given event occurs during run-time, for example, the number of cache misses, branch misprediction rates or load/store operation counts. The intuition behind their use is that two programs which have similar event values may share enough characteristics that they are optimisable in a similar way. An advantage of using these performance counters is that it is possible to create models which are agnostic to the source code language in which programs have been written, and that they can also take into consideration dynamic run-time behaviour. That said, generating optimisation models in this way is also particularly expensive. This is because recording these values necessarily involves profiling each application on which one wishes to train, and there are sometimes hundreds of events which can be recorded but only relatively few registers which can be used to do so: for example, the Intel i7 4770k CPU used for experimentation in this work has over 200 events but only 4–8 hardware counters, depending upon OS configuration [Intel, 2016]. Although it is possible to multiplex the recording of these events during run-time this is not desirable since it inevitably decreases accuracy, therefore, for *m* events, *n* registers and *o* samples to ensure statistical rigour $\lceil m/n \rceil \times o$ executions are required, per program.

To accelerate the learning process researchers traditionally use a subset of events to produce a good correlation between optimisation strategies and their effects [Cavazos et al., 2007]. This makes sense since it would require excessive training time to do otherwise, and, indeed, feature selection is usually performed explicitly anyway during any machine learning process irrespective of the domain. This helps to ensure quality heuristics are more likely to be produced since not all features help characterise a program sufficiently to differentiate it from others in terms of how it should be best optimised. The problem with this approach, however, is that it is extremely difficult to know *a priori* which features to 'buy'. Instead, feature selection is performed at the end of an unnecessarily expensive data collection phase. Moreover, researchers often rely on good feature sets chosen by others in the field, but this is problematic since it has been shown that a good subset of events to record changes depending not only upon the processor involved but also on the choice of compiler [Park et al., 2013].

Active feature acquisition was first proposed in a paper by Veeramachaneni and Avesani [2003] as a means to combat recording expensive features values unnecessarily during data collection. Inspired by active learning, the authors discuss how one might most efficiently choose between competing candidate features to add to an existing training set *a priori*: that is, before the relevant features have been recorded for all samples. In situations where it can take time to gather values for a particular feature, where they use apple tree disease prediction as an example, it would be beneficial to work out which features might improve the quality of a model the most whilst minimising the time to make this determination. They achieve a better than random subsampling policy by ranking candidate features based on their usefulness in terms of maximising the absolute change of entropy in class distribtuion. This work was later superseded by Deng et al. [2013], however, and it is this latter paper which forms the foundation of this research.

In the following sections a novel algorithm is presented which combines active learning [Settles, 2013] and active feature acquisition [Saar-Tsechansky et al., 2009]. The advantage of such a pairing is that this algorithm can select both good hardware events and good training instances to train on simultaneously during learning-time, in such a way as to minimise training cost whilst still producing quality results. In particular, the approach demonstrated in this chapter is based on the absolute biased round-robin with entropy technique presented by Deng et al. [2013], but adapted to include an ϵ-greedy probabilistic training example selector. With this unique combination this algorithm is able to construct a predictor which can propose good optimisations to apply for each benchmark in the PolyBench/C suite [Pouchet, 2012] 50% faster than the current state of the art whilst maintaining equally high-quality outcomes.

The rest of this chapter is structured as follows: Section 6.1 will give a motivating example for this research; Section 6.2 will explain the overall approach taken; Section 6.3 will give details of the experimental setup, where Section 6.4 analyses the results; Section 6.5 offers a brief self-reflective discussion on this contribution; and Section 6.6 concludes with proposals of future directions.

6.1 Motivation

To motivate this work a simple experiment is conducted to demonstrate the importance of feature selection in this domain, both in terms of eliminating attributes which may not be useful in characterising programs and in reducing training time.

To begin, 36 hardware event occurrences are counted using the Performance API or PAPI [Mucci et al., 1999] for unoptimised versions of all PolyBench/C benchmarks. More specifically, for each benchmark 5 runs were executed to aggregate performance counts in order to avoid any potential inaccuracies caused by experimental noise, and each individual count was normalised with respect to the total number of instructions executed per run. This data served as the oracle for all experiments in this Chapter. For details of which performance events were chosen (based on previous literature [Park et al., 2013]), the hardware specifications of the test bed machine, and the software used please refer to Section 6.3.

To establish the extent to which feature selection can have an impact on the efficacy of machine learning models the learning cost and quality of two classification based speed-up predictors are compared: the first uses all 36 hardware events while the other is trained using just 7. These 7 were chosen based on how much relative information they appeared to carry. This was found by creating a Linear Regression model evaluated using 10-fold cross-validation and taking those features from the whole training set which were assigned the largest magnitude coefficients. The task of these predictors was to forecast the performance an optimisation strategy would have on any given code relative to a baseline optimisation level—i.e. `-O3`. The feature vector itself contained both a set of normalised performance counts to characterise a program and a set of optimisations applied to that program—see Figure 6.2. The machine learning model used in these experiments was the J48 Decision Tree from the Weka Toolkit [Hall et al., 2009; Frank et al., 2016], which itself is a Java implementation of the C4.5 algo-

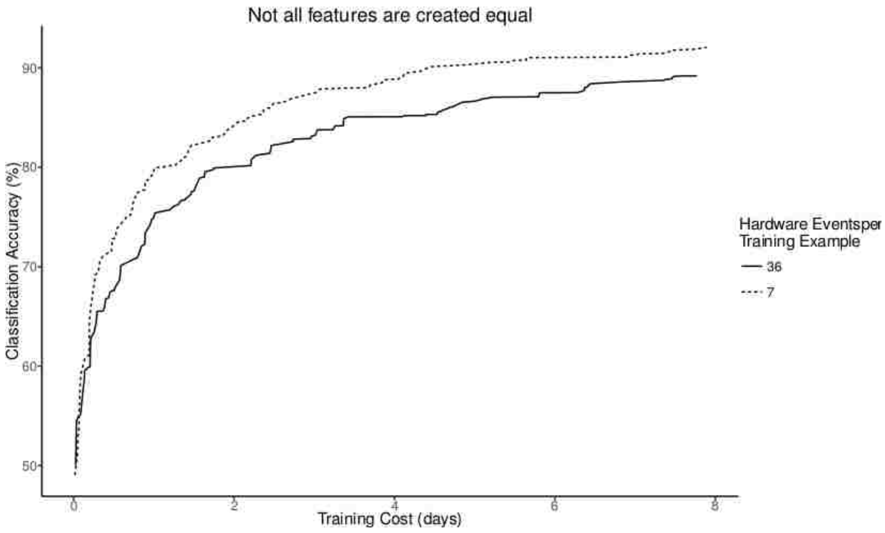


Figure 6.1: by using only 7 relatively more informative hardware performance event counts as features the learning speed-up of a predictor can be accelerated by 70% without harming quality

rithm [Quinlan, 1993]. The selection of training instances over time was random, and the experiments were repeated ten times to give an aggregate result.

As can be seen in Figure 6.1, recording much fewer performance events can substantially increase training efficiency—by 70% in this case—without harming quality; the challenge lies in finding which features to record at run-time for a given microarchitecture–compiler combination. In the next section a novel algorithm is outlined which can select during learning-time not only which features will most improve a performance model but also which benchmark/optimisation combination should be executed, thus reducing training cost and making machine learning based heuristic construction more appealing in the process.

6.2 Methodology

As outlined in the introductory text to this chapter, the goal of this work is to reduce the training overhead associated with creating any optimisation heuristic based on hardware performance counters; however, in order to do that it is first necessary to understand how these are currently modelled in state-of-the-art implementations. Therefore, Subsection 6.2.1 describes the *speed-up predictor* [Park et al., 2011] approach as it relates to this topic and how the method here differs. This explanation will ultimately state that the novelty and success of this contribution stems from its ability to select both good features and good training examples simultaneously at learning-time, where the details of how this is achieved are outlined in Subsections 6.2.2 and 6.2.3, respectively.

6.2.1 Speed-up Prediction with Hardware Performance Counters

First proposed by Cavazos et al. [2006, 2007], speed-up predictors based on hardware performance events use the number of times each event is recorded by CPU counters as a surrogate for program characterisation; together with a sequence of feature values which enumerate the optimisations applied to a given benchmark during compilation these constitute a complete feature vector. This feature vector is then mapped to the performance speed-up achieved by those optimisations over a baseline strategy (such as `-O2` or `-O3`) to form a complete training example—see Figure 6.2. The typical procedure for labelling an instance may be summarised as

1. compiling a benchmark with a base optimisation level only;
2. measuring all hardware event counts during multiple executions, normalising these to the total number of instructions per each execution, and recording the average runtime and average event counts;
3. optimising the benchmark in a particular way and recording the average runtime of the optimised binary;
4. encoding a feature vector with the normalised hardware event counts and the optimisation decisions; and associating that with the ratio of average optimised runtime relative to the baseline.

As in all passive learning scenarios many instances are labelled in this way and fed into a machine learning algorithm in one step, and this has been the methodology followed in all previous literature. In contrast, the novel approach presented in this chapter selects iteratively what to learn next based on what is already known: first by determining which hardware event to monitor and then what benchmark and set of optimisations to try.

When new information needs to be learnt by the model the hardware performance counter whose occurrences are to be recorded next is selected. Based on this, a random selection of candidate training instances are drawn together where either the hardware performance counters are all missing, i.e. this training instance has never been seen before, or where some hardware event counts are present but not the one being looked for this time. If the instance has not been seen before the process of labelling is similar to that outlined previously, except that in step 2 a single hardware event count is measured. If the instance has been seen before then only the unoptimised binary needs to be executed and the count value averaged. This process is summarised in Figure 6.3.

6.2.2 Absolute Biased Round-Robin Active Feature Acquisition

In this work the Absolute Biased Round-Robin (ABRR) algorithm proposed by Deng et al. [2013] is leveraged for feature selection at learning-time. As the name suggests this is based on the idea that missing features may be iterated through in a round-robin fashion, with an alteration that one should continue to 'purchase' the hardware event i of n to learn next if it is deemed to still be profitable, else move onto $(i+1)$ mod n. A purchase at time t is defined as being profitable if the current model has changed by more than some fixed amount $\delta(t) > \Delta$. This change is defined as

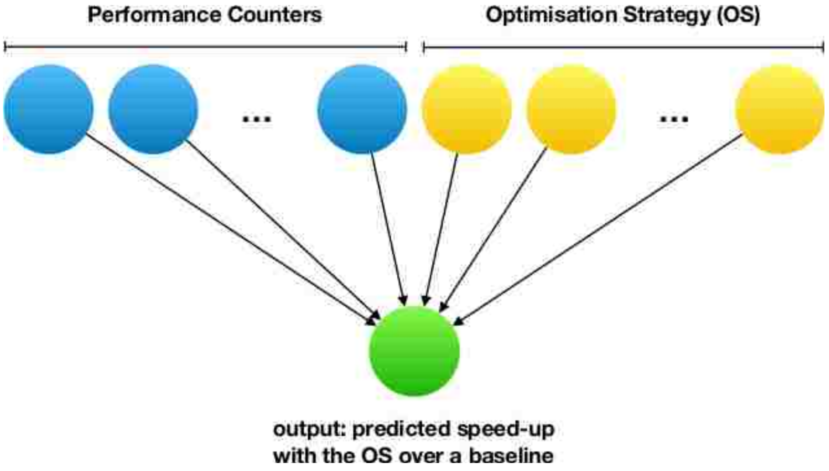
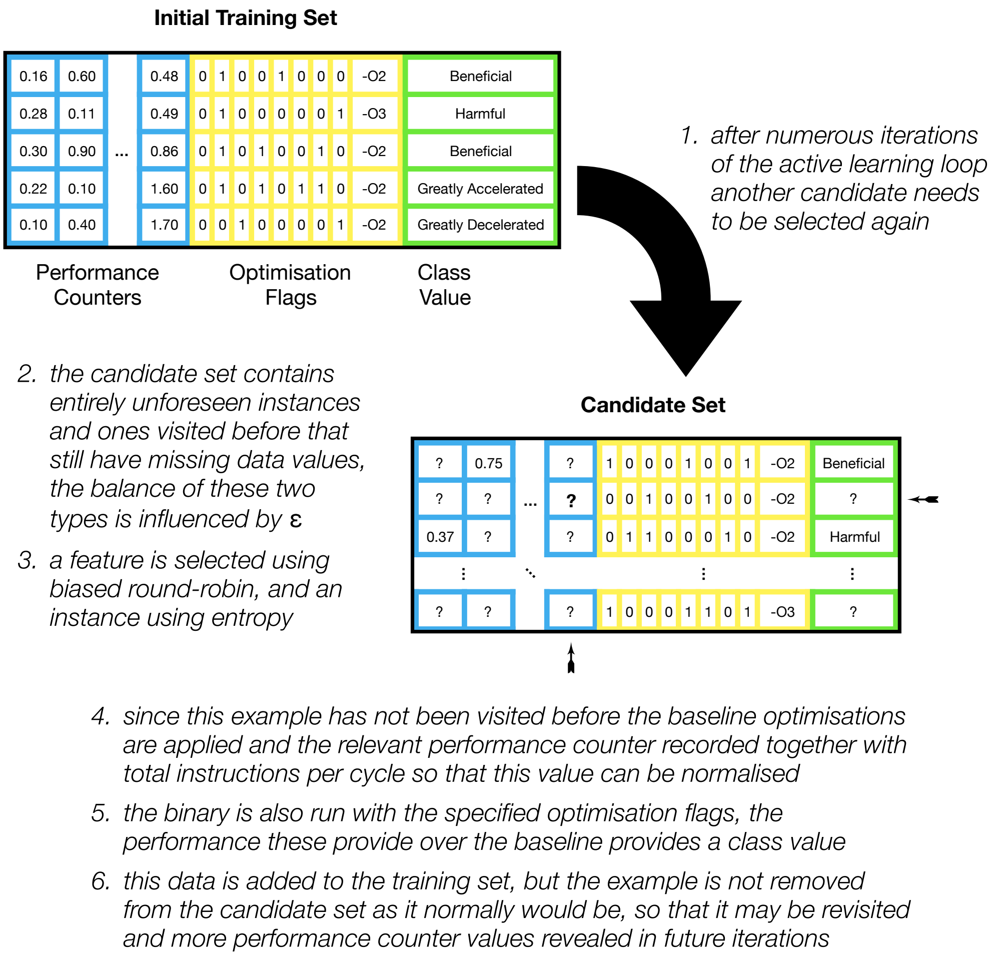

Performance Counters
Optimisation Strategy (OS)
...
...
output: predicted speed-up
with the OS over a baseline

Figure 6.2: adapted from Park et al. [2013], speed-up predictors which use performance counters record some set of hardware event occurrences normalised to the total number of machine instructions for a given program in order to identify the characteristics of that program, together with a finite set of optimisation options supplied to a compiler in order to form a feature vector; these values are then mapped to the relative performance achieved by the given program optimised with those options, with respect to a more established strategy such as `-O2` or `-O3`, and in this way an unseen program's performance with some optimisation strategy can be read off from the model

$$\delta(t) = \frac{\sum_{m=1}^{M} \sum_{j=1}^{J} |P_t(j|x_m) - P_{t-1}(j|x_m)|}{M}$$

where $P_t(j|x_m)$ is the probability of instance m of M, with feature vector x_m, belonging to class j of J at the t-th iteration of the learning loop. Similarly, $P_{t-1}(j|x_m)$ is the probability of x_m belonging to class j at the $(t-1)$-th iteration. In other words, $\delta(t)$ is the absolute change in class probabilities between the current model and the last model, averaged over all training instances.

Although ABRR will be demonstrated to work well within the systems domain it does require a slight change in the way one would typically model speed-up. That is to say, speed-up is normally measured as a ratio of the performance of an optimisation



Initial Training Set

0.16	0.60		0.48	0	1	0	0	1	0	0	0	-O2	Beneficial
0.28	0.11		0.49	0	1	0	0	0	0	0	1	-O3	Harmful
0.30	0.90	...	0.86	0	1	0	1	0	0	1	0	-O2	Beneficial
0.22	0.10		1.60	0	1	0	1	0	1	1	0	-O2	Greatly Accelerated
0.10	0.40		1.70	0	0	1	0	0	0	0	1	-O2	Greatly Decelerated

Performance Counters

Optimisation Flags

Class Value

1. *after numerous iterations of the active learning loop another candidate needs to be selected again*

2. *the candidate set contains entirely unforeseen instances and ones visited before that still have missing data values, the balance of these two types is influenced by ε*

3. *a feature is selected using biased round-robin, and an instance using entropy*

Candidate Set

?	0.75		?	1	0	0	0	1	0	0	1	-O2	Beneficial
?	?	...	**?**	0	0	1	0	0	1	0	0	-O2	?
0.37	?		?	0	1	1	0	0	0	1	0	-O2	Harmful
⋮		⋱					⋮						⋮
?	?		?	1	0	0	0	1	1	0	1	-O3	?

4. *since this example has not been visited before the baseline optimisations are applied and the relevant performance counter recorded together with total instructions per cycle so that this value can be normalised*

5. *the binary is also run with the specified optimisation flags, the performance these provide over the baseline provides a class value*

6. *this data is added to the training set, but the example is not removed from the candidate set as it normally would be, so that it may be revisited and more performance counter values revealed in future iterations*

Figure 6.3: the process of choosing a new candidate training instance to learn from involves both completely new unseen examples and those visited before which still have missing data.

strategy versus a baseline, but as this form of active feature acquisition requires classification in order to work the speed-up predictor was altered so that these speed-up ratios are binned into categories. This will be explained in more detail in Section 6.3.

6.2.3 Epsilon-Greedy Entropy Based Training Instance Selection

In the same paper as Deng et al. [2013] proposed that ABRR could be used to select features they also suggested that it be combined with an uncertainty based instance selector, specifically one that uses entropy. This type of selector is common in the literature [Settles, 2013]. It is also similar to the instance selector used in QBC, except (with reference to Equation 4.1) that $p(x_i)$ should be substituted with $p(x_i|m)$. This corresponds to the probability that instance i should belong to class m, which can be easily calculated from some types of models. Unfortunately, upon experimentation this technique was found to be insufficient at producing learning speed-ups over a baseline approach so an alteration had to be made.

When the data was analysed after using the approach as stated by Deng et al. [2013] the problem appeared to lie in the fact that when deciding which instance to label the algorithm would more-often-than-not select an instance where none of the other hardware events had been counted yet either. This resulted in a situation whereby the training set comprised training instances primarily with a single hardware event measurement, and where the rest of these event measurement were set as missing. Since the model was unable to identify which training information most closely resembled any given test instance the performance of the heuristic was poor.

In order to remedy this situation and encourage the selection of candidate instances where hardware event occurrences had already been measured the ε-greedy strategy (as previously discussed in Subsection 4.5.1) was applied to bias exploitation over exploration. More specifically, a new training instance with no previously revealed information is chosen with a probability of ε and, conversely, an example which has had at least one hardware event counter value is selected with probability $1-\varepsilon$—see Figure 6.3 for a graphical overview.

6.3 Experimental Setup

As in previous chapters, this section details the experimental setup used to validate the efficacy of the algorithm proposed in this work. In particular, the test bed machine is described along with the benchmarks from which training instances are derived to form the basis of the final heuristic. Next, the variables involved in the machine learning process are enumerated. Finally, evaluation methodology is discussed in the last subsection.

6.3.1 Platform and Benchmarks

Platform The hardware of the server used for recording performance was the same as in Chapters 4–5 but the software was updated. More specifically, the processor was an Intel Core i7-4770k CPU running at 3.4 GHz with 16 GB of RAM. The operating system was a fresh install of OpenSuse Leap v42.2, the machine was dedicated to the task of performance measurements with no other users or extraneous processes besides those enabled by default, and the compiler was GCC v4.8.5. Version 5.5 of the Performance API (PAPI) library [Mucci et al., 1999] was used to record the frequency of hardware events.

Benchmarks Table 6.1 lists the applications from PolyBench/C v4.2.1 [Pouchet, 2012] which were used for testing and evaluating the machine learning algorithm described in this work. For each benchmark, the hardware events used to form part of a feature vector for training were a subset drawn from the article by Park et al. [2013]; specifically, those events which are supported by the server—see Table 6.2. The optimisation strategies applied with GCC were taken from Ashouri et al. [2016], which itself was based on an earlier work by Chen et al. [2012b], and were selected because they have been found to have a significant impact on the performance of codes; that is to say, either `-O2` or `-O3` was selected along with any combination of the flags presented in Table 6.3.

6.3.2 Active Learning Settings

Learning Models The article by Deng et al. [2013] (which forms the basis for this contribution) suggests using a Bayesian model for learning through active feature acquisition; however, during experimentation it was found that the `J48` algorithm from Weka Toolkit v3.8.1 [Hall et al., 2009; Frank et al., 2016] produced superior results, as verified through cross-validation. As a necessity, `J48` also provides the ability to obtain the class probabilities required by the algorithm as well as the capacity to learn from instances with missing values.

Table 6.1: the 30 benchmarks in PolyBench/C v4.2.1, taken from Pouchet [2015]

Benchmark	**Description**
2mm	2 Matrix Multiplications (D=A.B; E=C.D)
3mm	3 Matrix Multiplications (E=A.B; F=C.D; G=E.F)
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
cholesky	Correlation Computation
covariance	Covariance Computation
deriche	Edge detection filter
doitgen	Multi-resolution analysis kernel (MADNESS)
durbin	Toeplitz system solver
fdtd-2d	2-D Finite Different Time Domain Kernel
gauss-filter	Gaussian Filter
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
gramschmidt	Gram-Schmidt decomposition
head-3d	Heat equation over 3D data domain
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
lu	LU decomposition
ludcmp	LU decomposition
mvt	Matrix Vector Product and Transpose
nussinov	Dynamic programming algorithm for sequence alignment
seidel	2-D Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k operations
syrk	Symmetric rank-k operations
trisolv	Triangular solver
trmm	Triangular matrix-multiply

Table 6.2: the Performance Counters (PC) collected using the PAPI library to uniquely identify each benchmark in Table 6.1, adapted from Park et al. [2013]

Category of PCs	**List of PCs selected**
Cache Line Access	CA_CLN, CA_ITV, CA_SHR
Level 1 Cache	L1_DCM, L1_ICM, L1_LDM, L1_STM, L1_TCM
Level 2 Cache	L2_DCA, L2_DCM, L2_DCR, L2_DCW, L2_ICA, L2_ICH, L2_ICM, L2_LDM, L2_STM, L2_TCA, L2_TCM, L2_TCR, L2_TCW
Level 3 Cache	L3_TCA, L3_TCM
Branch Related	BR_CN, BR_INS, BR_MSP, BR_NTK, BR_PRC, BR_TKN, BR_UCN
Interrupt / Stall	RES_STL
Translation Lookaside Buffer	TLB_DM, TLB_IM
Total Cycle or Instruction	TOT_CYC, TOT_INS
Load / Store Instruction	LD_INS, SR_INS

Initial Training Set For all experiments training began with a set of 5 randomly chosen instances, where each instance held values for all features. Which is to say, to label an instance *fully* the selected benchmark was first compiled using the `-O3` optimisation flag only. This binary was then was profiled enough times to get the average counts for all 36 hardware events normalised to the average total number of instructions. Next, either `-O2` or `-O3` was selected as a base optimisation level together with a random combination of flags chosen from those in Table 6.3. This optimised version of the program was profiled to calculate the mean runtime over 5 executions. The ratio of optimised runtime to base runtime was then binned into one of four categories—see Table 6.4. Together the compiler flags, normalised performance counters, and speed-up classification represent a complete training instance.

Candidate Sets The number of random candidate instances evaluated with each iteration of the learning loop was either $1,000$ or the total number of remaining possible candidates, whichever was lower. The instances in the candidate set were either those whose hardware performance counters had been partially revealed already or those which had never been profiled before, depending upon a random roll within the range $[0,1]$ and the value of ε, where different values were attempted.

Table 6.3: together with setting an optimisation level of either `-O2` or `-O3`, a boolean array denoted which, if any, of the following compiler flags were enabled for a given benchmark; this constituted the second portion of the feature vector used in this work.

Optimisation Flag	**Description**
-fno-ivopts	stops high-level loop induction variable optimisations
-fno-tree-loop-optimize	disables loop optimisations on trees
-fno-inline-functions	prevents inlining of functions, except those explicit marked with the `always_inline` attribute
-funroll-all-loops	unrolls every loop, even in cases where the number of loop iterations is uncertain at the beginning of the loop
-fno-guess-branch-probability	prohibits the guessing of the probability of a branch being taken or not based on heuristics
-funsafe-math-optimizations	allows float-point arithmetic optimisation that may violate IEEE or ANSI standards

Table 6.4: in order to apply this active feature acquisition technique to the problem of code optimisation the relative performance is binned, essentially turning the regression problem into a classification one; the table below lists the ranges of each classification as well as the prevalence of this classification in the data.

Relative Speed-up	**Classification**	**Instances with this Classification**
$x < 0.75$	Greatly Decelerated	205
$0.75 \leq x < 1$	Harmful	2331
$1 \leq x < 1.25$	Beneficial	1023
$1.25 \leq x$	Greatly Accelerated	281

Termination Criterion These experiments were terminated when the total training time of this novel learning technique exceeded the time taken for a baseline approach to achieve an accuracy of at least 90% on a test set, so that the learning speed-up could be calculated. This test set was made up of 20% of all possible training instances that could be produced given the optimisation-space and number of benchmarks in the PolyBench/C suite. To ensure accuracy could be precisely measured the test set instances were excluded from the possibility of being trained upon.

6.3.3 Evaluation Methodology

Baseline Approach The previous state-of-the-art methodology for creating hardware performance counter based optimisation heuristics involved recording a subset of hardware events and combining this with profiling an optimisation strategy over a predefined compiler optimisation level—i.e. a passive learning derived speed-up model. Therefore, in this work a technique which collects training data at random, and for each obtains all 36 hardware event values, is compared against the proposed algorithm from this chapter. In particular, the time taken for all program executions required by the baseline approach to achieve an accuracy of 90% on the held out test set is compared to the time required by the novel strategy. As well as this passive learning comparison a comparison is also made against an active learning technique, whereby all hardware events are still recorded per training instance but the instances themselves are selected to be learnt from by considering their usefulness given the current information held by the model, as defined by an entropy calculation.

Evaluation Methodology Both the active and passive baselines discussed in the previous paragraph are used to compare this unique approach to the state of the art, however, the algorithm outlined in Section 6.2 is also parameterised by two variables. The first is the threshold over which it is deemed profitable to continue to select a given feature for learning Δ, versus moving on in a round-robin fashion. The second is the value ε which determines the probability that a completely new, previously unseen, instance should be selected for training next, as opposed to one for which at least one hardware performance counter value has been recorded. Different values for these variables are tried and evaluated against these baselines, and the results presented in Section 6.4.

Description of the Dataset The data used in the following experiments was collected ahead of time. First each benchmark was compiled with `-O3` and the average runtime and mean performance counters recorded over 5 runs. Next, the cross-product of these benchmarks with all possible optimisation combinations of `-O2` or `-O3` together with the flags in Table 6.3 were compiled and executed to calculate the relative performance of those settings versus the `-O3` runtime, where this ratio was binned into a category, as defined in Table 6.4.

6.4 Experimental Results

This section is broken down into two parts, the first presents the main findings of the experiments to date and the second examines the affect of parameters on efficiency.

6.4.1 Overall Efficiency Savings

Figure 6.4 summarises the main finding of this preliminary work, that by employing a novel approach which leverages an active feature selection technique the cost of training a predictor to select good optimisations for a previously unseen code can be reduced by 50% as compared to a state-of-the-art passive learning approach. Somewhat surprisingly, active learning on its own is actually slower than the random methodology. Since the input feature vector is dominated by characterisation of a program, this could be down to the fact that the active learner is naturally biased towards exploring more programs as opposed to more optimisation decisions per program, although this would need to be investigated further in future work.

6.4.2 Sensitivity to Parameters

During experimentation the parameters ε and Δ were varied to see what affect this might have on the efficiency of the algorithm with the results given in Table 6.5. Interestingly, a low value for each parameter gives the best performance. This means that, ideally, even small changes in the structure of the model should be detected to determine whether a hardware performance event is useful for characterisation or not.

What is also clear from the data is that a relatively low value for ε is most helpful, meaning exploitation is more beneficial than exploration in this particular task, which makes intuitive sense since until some number of performance counters have been recorded for a particular optimisation–benchmark combination it is difficult to generalise about how that training instance relates to other programs.

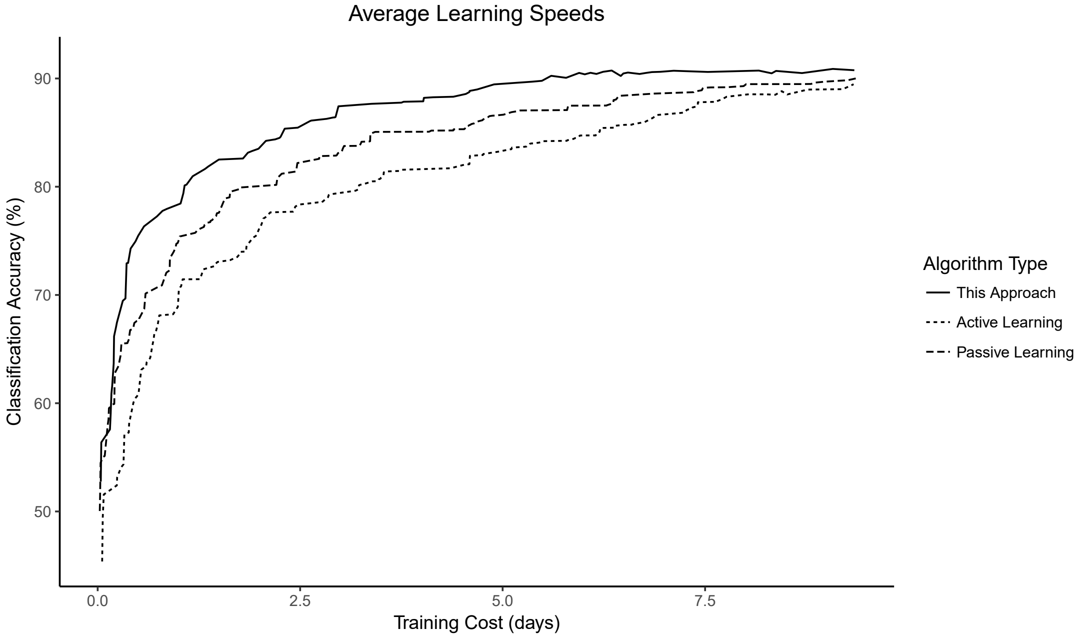


Figure 6.4: the novel approach presented in this chapter is able to produce a model with a 90% accuracy 50% faster than the previous state of the art. The ε and Δ values were 0.05 and 0.001, respectively, based on a search of the parameter-space. The 90% completion criterion was chosen before the alternative approaches were tested, based upon the maximum accuracy achieved by the passive approach.

Table 6.5: varying ε and Δ has some effect on the efficiency of the algorithm, with a low value for each giving the best performances. This indicates that even small changes in the model structure should be used as an indicator of a useful feature, and that exploitation is generally more helpful than exploration in this task.

Delta	Epsilon	Max. Accuracy (%)	Training Cost (days)
0.001	0.05	92.19	7.94
	0.10	87.51	3.47
	0.15	74.37	6.42
	0.20	71.55	5.81
0.010	0.05	89.96	8.00
	0.10	77.45	6.70
	0.15	72.66	6.40
	0.20	70.96	5.91
0.100	0.05	87.07	7.68
	0.10	69.20	5.64
	0.15	70.17	6.44
	0.20	70.87	5.82

6.5 Discussion

The work in this chapter is preliminary and there are at least two ways in which it is suggested that it might be improved upon. First, by the re-establishment of the regression model which offers more predictive power than classification in this case, and secondly by selecting multiple counters per iteration, thereby parallelising the methodology.

6.5.1 Re-establishing the Regression Model

By choosing to use the ABRR learning-time feature selection technique proposed by Deng et al. [2013] it was necessary to cast the speed-up prediction problem as one of classification. This was necessary because the calculation of δ_t, the change observed in the model structure at time t, relied upon the probability of class membership per instance; however, the disadvantage of this is that information is lost in the binning of runtime values. In other words, where classification permits choosing some optimisation strategy that it is predicted will increase runtime by at least 25% over `-O3`, regression allows the selection of the predicted *best* configuration.

There are two ways in which regression may be re-incorporated back into the technique. The first is to re-define δ in such a way as to calculate movement in the structure of the model in terms of predicted values, as opposed to class membership. The second is to use the described classification methodology to select which features and which instances to learn over but also train a regression based model in parallel with full runtime data. It is left to future work to see which of these two approaches would produce the best result.

6.5.2 Selection of Multiple Counters per Iteration

In the approach discussed in this chapter a single hardware event's average normalised value is selected to be learnt at each iteration of the learning process. However, since machines often have multiple registers to allow some number of events to be recorded simultaneously at run-time it is possible to record multiple events at once. That being the case, a clear improvement to the algorithm would be to use all of these available registers to search over the space more efficiently. Where the number of registers available is n, one approach would be to learn the first n of m hardware events supported by the architecture in the first loop iteration. Upon receiving these counts n copies of the existing training set can be created where each is updated with a distinct feature value each. From these n δ_t values can be calculated to see which features are worth selecting in the next iteration. Of those that are not worth continued selection an atomically updated integer can point to the next feature to learn in the round-robin algorithm. Again, due to a lack of time, this approach is left to future work.

6.6 Summary

In this chapter a novel algorithm which combines active learning and active feature acquisition has been presented. At each iteration of the learning loop a hardware performance event is selected in a round-robin fashion, but biased towards using the same event if the model is changing significantly. Upon selection of a hardware event, the instance itself is chosen based on an entropy utilisation estimate. In this way, the algorithm can select which feature, optimisation, and benchmark to measure during learning that will most improve the model. In an evaluation against the current state of the art this approach is able to accelerate learning by 50% without negatively affecting heuristic performance results.

Chapter 7

Conclusions

This thesis has attempted to address a substantive and timely problem in computer systems research; that is, how to quickly and accurately produce heuristics which can predict good compile-time or run-time decisions that ultimately result in an efficient program execution. Although these works have explicitly looked at optimising execution time the techniques described here are equally applicable to either power or energy conservation, smaller code sizes, or some combination of those same.

This research was motivated by the appearance of outdated heuristics in modern compilers [Kulkarni and Cavazos, 2012], which have been the result of increasingly complicated hardware as well as the tradition of manually fine-tuning heuristics for each platform in turn; despite this, this expert-driven process inexplicably continues to be followed in the main despite machine learning having been shown to outperform it [Dubach et al., 2009; Kulkarni and Cavazos, 2012]. The intuition on which the works in this thesis are based is that it is the speed of training machine learning based auto-tuning, or indeed the lack of, that is *the* hindrance to it being adopted as the *de facto* standard heuristic generation procedure.

To tackle this challenging problem, Chapter 4 proposes that active learning be used in place of the random, passive machine learning technique (utilised extensively in prior literature) to concentrate on only those training examples which are predicted to provide the most information. Next, Chapter 5 demonstrates that it is possible to further substantially reduce this training overhead through the inclusion of sequential analysis: which can optimise the number of samples per example rather than just the number of examples. Finally, Chapter 6 demonstrates that where recording more feature values results in an increase in training time a combination of active learning and active feature selection can be used to further speed-up model creation.

In combination it is hoped that these works will go some way towards making heuristic generation less costly, and hence help ensure that good heuristics are always available for any code, on any platform. For the remainder of this chapter, Section 7.1 will briefly summarise each of these individual contributions in turn, and Section 7.2 will conclude with a discussion of possible directions for future work.

7.1 Contributions

This thesis has made three principle contributions to the topic of auto-tuning optimisation heuristics: two peer-reviewed and one preliminary. The following subsections will summarise each of these in the order they appeared in the text.

Heuristic Generation with Active Learning

It is difficult to always ensure a run-time or compile-time optimisation heuristic is up-to-date at the time of use because there are usually many ways in which optimisations can be applied. Indeed, the spaces can be unimaginably large, for example, the popular GCC compiler has on the order of 10^{400} *combinations* of optimisations, and this is before one considers the much greater number of *permutations*—i.e. the phase ordering problem [Kulkarni and Cavazos, 2012]. Moreover, the task for engineers is not just to tune heuristics for a single target, but for multiple targets. Even worse, there is finite time in which one can concentrate on each because new hardware is introduced each year. With this complexity and the sheer arduous nature of the work it is no wonder heuristics are often simply left outdated, since the effort to keep up is monumental.

Chapter 4 explained that machine learning has been shown to outperform expert-constructed heuristics and it was hoped since this is an automated process it might ameliorate this situation; however, despite this literature, the technique is still not popularly applied outside of academia. This thesis proposes that it is the slow nature of training that may to be to blame, and so has attempted to tackle that.

In particular, the vast majority of the literature on auto-tuning heuristics using machine learning has used passive learning techniques to generate their respective models—i.e. learning at random. This is often taught as the standard supervised machine learning process but ignores the fact that randomness can result in redundancy, and where each training example comes at a cost this is problematic. Active machine learning is specifically designed to tackle this issue, and in Chapter 4 an approach to active learning called Query-by-Committee (QBC) is used to show that it can indeed speed-up learning. Across four benchmarks QBC was able to accelerate the learning of inputs-dependent heuristics by 4x on average, and up to 8x at best, compared to a random-learning baseline approach.

Active Learning with Sequential Analysis

The intuition behind the work presented in Chapter 5 was that repeated samples are necessary to understand the consequences of an optimisation decision due to experimental noise, but that this number can be optimised on a per training example basis.

The particular challenge in this work was understanding when it is more beneficial to be more certain about a particular runtime estimate for a given optimisation strategy versus profiling a new strategy entirely. The approach assumes that the spatial locality of sample runtimes can give some indication about the certainty one should have about a particular set of measurements, and hence how much information might be gained from further increased precision versus trying uncharted optimisations. This assumption appears to be correct in that upon evaluation the presented approach was able to speed-up learning by up 26x compared to active learning alone [Balaprakash et al., 2013a], where across 11 benchmarks the average speed-up was 4x.

This research illustrates that not only can active learning substantially speed-up the creation of heuristics, but that by adding sequential analysis the learning acceleration can be further and significantly increased.

Active Learning with Active Feature Acquisition

Lastly, some preliminary work was provided in Chapter 6 which discusses the inefficiencies of previous machine learning based heuristic generation implementations which rely upon hardware performance counters to characterise programs. That is, that a subset of all performance events are chosen in practice to use in program characterisation because recording them all would take too long, but that it is difficult to know which events to use *a priori*. Due to this, authors tend to go with whatever events a previous study has used, but this is problematic since a good subset depends upon the architecture and the system software present on it. The way in which feature selection is performed currently involves collecting a data set and then working backwards to determine which hardware events were worth recording, but this wastes much time on profiling. In contrast, the approach discussed in Chapter 6 is able to dynamically determine at learning-time whether it is worth continuing to use a given hardware event for further training, thereby greatly accelerating learning. Indeed, the presented implementation is able to achieve a 50% speed-up over the previous state of the art and this is only an initial finding, where this work could be futher improved with some effort before publication.

7.2 Future Work

In terms of future work proceeding from these experiments, which has not already been discussed, there are a number of directions one could take. Firstly, it would be interesting to find out to what extent the methodology presented in Chapter 5, which combines active learning and sequential analysis, can withstand more extreme cases of environmental noise. No strenuous effort was made to reduce the potential noise that might be encountered while recording runtimes for the oracle for each benchmark, and this was a deliberate action since, as has already been stated, significantly altering the run-time environment of the target machine might invalidate the effectiveness of the produced heuristics; however, the machine used to record these runtimes was set aside specifically for that purpose. In an industrial setting where the optimisation-spaces might be even larger, and the ability to isolate a given number of machines for a particular task much harder, this proposal might be more desirable if it was shown that the technique could still produce effective heuristics within a cluster. Clusters are an interesting case since they are so prevalent in high-performance computing, and it would be convenient to be able to produce heuristics directly on them without having to isolate individual computers. That said, clusters also necessarily require extra software to handle inter-machine communications, which might introduce extraneous noise above that of a simple stand-alone machine. It remains to be seen whether the current implementation, without alteration, would be able to deal with this more extreme case, and if not what modifications could be made to allow for it.

Secondly, it must be acknowledged that although the speed-ups from the second technical chapter are substantial, and the methodology is proven to work on the benchmarks tested, the engineering of the process was regrettably somewhat *ad-hoc*. It would be interesting to see, therefore, if another implementation could be found which is more mathematically rigorous. There is an abundance of literature in the statistics community on sequential analysis techniques [Online, 2017], and in the Geosciences there is a methodology called Kriging [Chils and Delfiner, 2012] which is used to estimate where to dig for precious minerals given a limited set of sample borehole locations. A collaboration with an expert in either or both of these fields might yield even better results than those demonstrated here: it would certainly be fascinating to investigate.

Thirdly, embedded platforms often have multiple processors which are specialised to different workload types. For example, Qualcomm's recent Snapdragon 835 system on a chip contains a CPU, GPU and a DSP [Qualcomm, 2017]. Depending upon the application being run one could imagine that it might be best executed on one or a multitude of these processors simultaneously, in an ideal world. However, for that to be possible it would require a program to be written such that it could be compiled at run-time down to the relevant machine code for the specific processor, a compiler which understands which optimisations to apply per device, and a scheduler which can map code features and or workload to the most appropriate device using some precomputed heuristics. The benefits of such a scheduler could potentially be a significant increase in battery life for such chips, but would need to be researched.

Finally, the work by Cummins *et al.* [Cummins et al., 2017] is a natural progression from this thesis. Which is to say, the hypothesis on which this research was based is that it is the time required to collect training data which is a key reason why machine learning based heuristics are not seen more readily in production systems, however, that is only one part of it. Another problem is simply the lack of diversity when it comes to benchmarks on which heuristics can be trained. Generally speaking, the more training examples provided to a machine learning algorithm the better the final model is likely to be, although, as has been demonstrated, not all information is equally useful. That said, there is a small finite number of benchmarks currently available for learning. Since benchmarks require substantial effort to write it is not sensible to expect significantly more will appear in the near future, hence, artificial benchmark generation might provide a way to further increase the benefits and quality of using artificially derived heuristics.

Appendix A

An OpenCL Code Example

The following OpenCL code example, which performs vector addition, was adapted from Oak Ridge Leadership Computing Facility [2014]:

```
#include <CL/opencl.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

// an OpenCL Kernel, each work-item takes care of one element of c
const char *kSource =                                         "\n" \
"__kernel void vecAdd(__global double *a,                      \n" \
"                     __global double *b,                      \n" \
"                     __global double *c,                      \n" \
"                     const unsigned int n) {                  \n" \
"                                                              \n" \
"    // get the global thread ID                               \n" \
"    int id = get_global_id(0);                                \n" \
"                                                              \n" \
"    // make sure to keep within bounds                        \n" \
"    if (id < n)                                               \n" \
"        c[id] = a[id] + b[id];                                \n" \
"                                                              \n" \
"}                                                             \n"

int main(int argc, char* argv[]) {

    // length of vectors
    unsigned int n = 100000;
```

```
// host input vectors
double *h_a;
double *h_b;

// host output vector
double *h_c;

// device input buffers
cl_mem d_a;
cl_mem d_b;

// device output buffer
cl_mem d_c;

// required OpenCL pointers
cl_platform_id cpPlatform;
cl_device_id device_id;
cl_context context;
cl_command_queue queue;
cl_program program;
cl_kernel kernel;

// size, in bytes, of each vector
size_t bytes = n*sizeof(double);

// allocate memory for each vector on the host
h_a = (double*)malloc(bytes);
h_b = (double*)malloc(bytes);
h_c = (double*)malloc(bytes);

// initialise vectors on the host
int i;
for (i=0; i<n; i++) {
    h_a[i] = sinf(i) * sinf(i);
    h_b[i] = cosf(i) * cosf(i);
}

size_t globalSize, localSize;
cl_int err;

// number of work items in each local work group
localSize = 64;
```

```
// number of total work items - localSize must be deviser
globalSize = ceil(n/(float)localSize)*localSize;

// connect to the first installed platform
err = clGetPlatformIDs(1, &cpPlatform, NULL);

// get an ID for a GPU device
err = clGetDeviceIDs(cpPlatform,
                     CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);

// create the context
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

// create the command queue
queue = clCreateCommandQueue(context, device_id, 0, &err);

// create the compute program from the source buffer
program = clCreateProgramWithSource(context, 1,
                                    (const char **)&kSource,
                                    NULL, &err);

// build the program executable
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// designate the kernel
kernel = clCreateKernel(program, ”vecAdd”, &err);

// create the input and output arrays in device memory
d_a = clCreateBuffer(context,
                     CL_MEM_READ_ONLY, bytes, NULL, NULL);
d_b = clCreateBuffer(context,
                     CL_MEM_READ_ONLY, bytes, NULL, NULL);
d_c = clCreateBuffer(context,
                     CL_MEM_WRITE_ONLY, bytes, NULL, NULL);

// write the data into the input arrays
err  = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0,
                                bytes, h_a, 0, NULL, NULL);
err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0,
                                bytes, h_b, 0, NULL, NULL);
```

```
    // set the arguments to the compute kernel
    err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
    err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
    err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
    err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);

    // execute the kernel over the entire range of the data
    err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
                                 &globalSize, &localSize,
                                 0, NULL, NULL);

    // wait for the command queue to get serviced
    // before reading back results
    clFinish(queue);

    // read the results from the device
    clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0, bytes, h_c, 0,
                        NULL, NULL);

    // sum up vector c and print result divided by n,
    // this should roughly equal 1
    double sum = 0;
    for (i=0; i<n; i++)
        sum += h_c[i];
    printf(”final result: %f\n”, sum/n);

    // release OpenCL resources, and host memory
    clReleaseMemObject(d_a);
    clReleaseMemObject(d_b);
    clReleaseMemObject(d_c);
    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;

}
```

Bibliography

Aarts, B., Barreteau, M., Bodin, F., Brinkhaus, P., Chamski, Z., Charles, H.-P., Eisenbeis, C., Gurd, J., Hoogerbrugge, J., Hu, P., Jalby, W., Knijnenburg, P. M. W., O'Boyle, M. F. P., Rohou, E., Sakellariou, R., Schepers, H., Seznec, A., Stöhr, E., Verhoeven, M., and Wijshoff, H. A. G. (1997). OCEANS: Optimizing Compilers for Embedded Applications. In *Proceedings of the International European Conference on Parallel and Distributed Computing*, pages 1351–1356.

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 *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305.

ARM (2016). big.LITTLE Technology. Retrieved 18 March 2018 from https://www.arm.com/products/processors/technologies/biglittleprocessing.php.

Ashouri, A. H., Mariani, G., Palermo, G., Park, E., Cavazos, J., and Silvano, C. (2016). COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *ACM Transactions on Architecture and Code Optimization*, 13(2):21.

Ashouri, A. H., Mariani, G., Palermo, G., and Silvano, C. (2014). A Bayesian Network Approach for Compiler Auto-tuning for Embedded Processors. In *Proceedings of the IEEE Symposium on Embedded Systems for Real-time Multimedia*, pages 90–97.

Balaprakash, P., Gramacy, R. B., and Wild, S. M. (2013a). Active-Learning-Based Surrogate Models for Empirical Performance Tuning. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 1–8.

Balaprakash, P., Rupp, K., Mametjanov, A., Gramacy, R. B., Hovland, P. D., and Wild, S. M. (2013b). Empirical Performance Modeling of GPU Kernels Using Active Learning. In *Proceedings of the International Conference on Parallel Computing*, pages 646–655.

Balaprakash, P., Wild, S. M., and Norris, B. (2012). SPAPT: Search Problems in Automatic Performance Tuning. In *Proceedings of the International Conference on Computational Science*, pages 1959–1968.

Beluja, S. (1994). Population-based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning. Technical Report CMU-CS-94-163, Carnegie Mellon University.

Berry, D. and Fristedt., B. (1985). *Bandit Problems*. Chapman and Hall.

Bodin, F., Kisuki, T., Knijnenburg, P. M. W., O'Boyle, M. F. P., and Rohou, E. (1998). Iterative Compilation in a Non-linear Optimisation Space. In *Proceedings of the Workshop on Profile and Feedback-Directed Compilation*.

Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1):5–32.

Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. A. (1984). *Classification and Regression Trees*. Chapman and Hall/CRC.

Callahan, D., Carr, S., and Kennedy, K. (1990). Improving Register Allocation for Subscripted Variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 53–65.

Callahan, D., Cocke, J., and Kennedy, K. (1988). Estimating Interlock and Improving Balance for Pipelined Architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358.

Carr, S. and Kennedy, K. (1994). Scalar Replacement in the Presence of Conditional Control Flow. *Software: Practice and Experience*, 24(1):51–77.

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 *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 24–34.

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 *Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197.

Charles, J., Jassi, P., Ananth, N. S., Sadat, A., and Fedorova, A. (2009). Evaluation of the Intel Core i7 Turbo Boost Feature. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 188–197.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.-H., and Skadron, K. (2009). Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 44–54.

Che, S., Sheaffer, J., Boyer, M., Szafaryn, L., Wang, L., and Skadron, K. (2010). A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 1–11.

Chen, C., Chame, J., and Hall, M. (2008). CHiLL: A Framework for Composing High-level Loop Transformations. Technical Report 08-897, University of Southern California.

Chen, Y., Fang, S., Eeckhout, L., Temam, O., and Wu, C. (2012a). Iterative Optimization for the Data Center. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–60.

Chen, Y., Fang, S., Huang, Y., Eeckhout, L., Fursin, G., Temam, O., and Wu, C. (2012b). Deconstructing Iterative Optimization. *ACM Transactions on Architecture and Code Optimization*, 9(3):21.

Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., and Wu, C. (2010). Evaluating Iterative Optimization Across 1000 Datasets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 448–459.

Chils, J.-P. and Delfiner, P. (2012). *Geostatistics*, chapter 3. John Wiley & Sons, Inc.

Chipman, H. A., George, E. I., and McCulloch, R. E. (1998). Bayesian CART Model Search. *Journal of the American Statistical Association*, 93(443):935–948.

Chipman, H. A., George, E. I., and McCulloch, R. E. (2002). Bayesian Treed Models. *Machine Learning*, 48(1–3):299–320.

Chung, I.-H. and Hollingsworth, J. K. (2004). Using Information from Prior Runs to Improve Automated Tuning Systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 30.

Cilibrasi, A. R. and Vitanyi, A. P. (2005). Clustering by Compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545.

Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37.

Cohn, D. A. (1996). Neural Network Exploration Using Optimal Experiment Design. *Neural Networks*, 9(6):1071–1083.

Coons, K. E., Robatmili, B., Taylor, M. E., Maher, B. A., Burger, D., and McKinley, K. S. (2008). Feature Selection and Policy Optimization for Distributed Instruction Placement Using Reinforcement Learning. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 32–42.

Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S., Subramanian, D., Torczon, L., and Waterman, T. (2005). ACME: Adaptive Compilation Made Efficient. *ACM SIGPLAN Notices*, 40(7):69–77.

Cooper, K. D., Schielke, P. J., and Subramanian, D. (1999). Optimizing for Reduced Code Space using Genetic Algorithms. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9.

Cooper, K. D., Subramanian, D., and Torczon, L. (2002). Adaptive Optimizing Compilers for the 21st Century. *The Journal of Supercomputing*, 23(1):7–22.

Ţăpuş, C., Chung, I.-H., and Hollingsworth, J. K. (2002). Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 1–11.

Cummins, C., Petoumenos, P., Steuwer, M., and Leather, H. (2016). Autotuning OpenCL Workgroup Size for Stencil Patterns. In *Proceedings of the International Workshop on Adaptive Self-tuning Computing Systems*.

Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. (2017). Synthesizing Benchmarks for Predictive Modeling. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 86–99.

Curtsinger, C. and Berger, E. D. (2013). STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 219–228.

de Oliveira, A. B., Petkovich, J.-C., and Fischmeister, S. (2014). How much does memory layout impact performance? A wide study. In *Proceedings of the International Workshop on Reproducible Research Methodologies*, pages 23–28.

Deng, K., Zheng, Y., Bourke, C., Scott, S., and Masciale, J. (2013). New Algorithms for Budgeted Learning. *Machine Learning*, 90(1):59–90.

Denning, P. J. (2005). The Locality Principle. *Communications of the ACM*, 48(7):19–24.

Denning, P. J. and Lewis, T. G. (2017). Exponential Laws of Computing Growth. *Communications of the ACM*, 60(1):54–65.

Dubach, C., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F., and Temam, O. (2007). Fast Compiler Optimisation Evaluation Using Code-feature Based Performance Prediction. In *Proceedings of the International Conference on Computing Frontiers*, pages 131–142.

Dubach, C., Jones, T. M., Bonilla, E., Fursin, G., and O'Boyle, M. F. P. (2009). Portable Compiler Optimisation Across Embedded Programs and Microarchitectures Using Machine Learning. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 78–88.

Duesterwald, E., Gupta, R., and Soffa, M. L. (1993). A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–77.

Emani, M. K., Wang, Z., and O'Boyle, M. F. P. (2013). Smart, Adaptive Mapping of Parallelism in the Presence of External Workload. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 1–10.

Epshteyn, A., Garzaran, M. J., DeJong, G., Padua, D., Ren, G., Li, X., Yotov, K., and Pingali, K. (2005). Analytic Models and Empirical Search: A Hybrid Approach to

Code Optimization. In *Proceedings fo the International Conference on Languages and Compilers for Parallel Computing*, pages 259–273.

Everitt, B. (2011). *Cluster Analysis*. Wiley.

Fang, S., Xu, W., Chen, Y., Eeckhout, L., Temam, O., Chen, Y., Wu, C., and Feng, X. (2015). Practical Iterative Optimization for the Data Center. *ACM Transactions on Architecture and Code Optimization*, 12(2):15:1–15:26.

Felsenstein, J. (2003). *Inferring Phylogenies*. Sinauer Associates.

Fisher, J. A., Faraboschi, P., and Young, C. (2005). *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*, chapter 7, page 288. Morgan Kaufmann.

Frank, E., Hall, M., and Witten, I. H. (2016). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., 4th edition.

Franke, B., O'Boyle, M. F. P., Thomson, J., and Fursin, G. (2005). Probabilistic Source-level Optimisation of Embedded Programs. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 78–86.

Fursin, G., Cohen, A., O'Boyle, M. F. P., and Temam, O. (2005). A Practical Method for Quickly Evaluating Program Optimizations. In *Proceedings of the International Conference on High-Performance Embedded Architectures and Compilers*, pages 29–46.

Fursin, G., Kashnikov, Y., Memon, A. W., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., Bodin, F., Barnard, P., Ashton, E., Bonilla, E., Thomson, J., Williams, C. K. I., and O'Boyle, M. F. P. (2011). Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming*, 39(3):296–327.

Fursin, G., O'Boyle, M. F. P., and Knijnenburg, P. M. W. (2002). Evaluating Iterative Compilation. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 362–376.

Fursin, G., O'Boyle, M. F. P., Temam, O., and Watts, G. (2004). A Fast and Accurate Method for Determining a Lower Bound on Execution Time. *Concurrency Practice and Experience*, 16(2–3):271–292.

Fursin, G. and Temam, O. (2010). Collective Optimization: A Practical Collaborative Approach. *ACM Transactions on Architecture and Code Optimization*, 7(4):20:1–20:29.

Gramacy, R. B. and Taddy, M. A. (2017). Dynamic Trees for Learning and Design. Retrieved 18 March 2018 from http://bobby.gramacy.com/r_packages/dynaTree/.

Gramacy, R. B., Taddy, M. A., and Wild, S. M. (2013). Variable Selection and Sensitivity Analysis Using Dynamic Trees with an Application to Computer Code Performance Tuning. *The Annals of Applied Statistics*, 7(1):51–80.

Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., and Cavazos, J. (2012). Auto-tuning a High-level Language Targeted to GPU Codes. In *Proceedings of the Innovative Parallel Computing Conference*, pages 1–10.

Grewe, D., Wang, Z., and O'Boyle, M. F. P. (2011). A Workload-aware Mapping Approach for Data-parallel Programs. In *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers*, pages 117–126.

Grewe, D., Wang, Z., and O'Boyle, M. F. P. (2013a). OpenCL Task Partitioning in the Presence of GPU Contention. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 87–101.

Grewe, D., Wang, Z., and O'Boyle, M. F. P. (2013b). Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 1–10.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, pages 10–18.

Haneda, M., Knijnenburg, P. M. W., and Wijshoff, H. A. G. (2005). Optimizing General Purpose Compiler Optimization. In *Proceedings of the Conference on Computing Frontiers*, pages 180–188.

Harke, R. R. (1998). Interactive-engagement versus Traditional Methods: A Six-thousand-student Survey of Mechanics Test Data for Introductory Physics Courses. *American Journal of Physics*, 66(1):64–74.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, chapter 7. Springer, 2nd edition.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*, chapter 1, page 55. Morgan Kaufmann Publishers Inc., 5th edition.

Herter, J., Backes, P., Haupenthal, F., and Reineke, J. (2011). CAMA: A Predictable Cache-Aware Memory Allocator. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 23–32.

Hoellwarth, C. and Moelter, M. J. (2011). The Implications of a Robust Curriculum in Introductory Mechanics. *American Journal of Physics*, 79(5):540–545.

Huang, W., Ghosh, S., Velusamy, S., Sankaranarayanan, K., Skadron, K., and Stan, M. (2006). HotSpot: A Compact Thermal Modelling Methodology for Early-stage VLSI Design. *IEEE Transactions on Very Large Scale Integration Systems*, pages 501–513.

Intel (2016). *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3B, System Programming Guide, Part 2*. Intel Corporation.

ITRS (2001). Executive Summary. Technical report, The International Technology Roadmap for Semiconductors. Retrieved 18 March 2018 from http://www.itrs2.net/itrs-reports.html.

ITRS (2015). Executive Summary. Technical report, The International Technology Roadmap for Semiconductors. Retrieved 18 March 2018 from http://www.itrs2.net/itrs-reports.html.

Jiménez, M., Llabería, J. M., and Fernández, A. (2002). Register Tiling in Nonrectangular Iteration Spaces. *ACM Transactions on Programming Languages and Systems*, 24(4):409–453.

Jolliffe, I. (2002). *Principle Component Analysis*. Springer, 2 edition.

Joshi, A., Phansalkar, A., Eeckhout, L., and John, L. K. (2006). Measuring Benchmark Similarity Using Inherent Program Characterisitcs. *IEEE Transactions on Computers*, 55(6):769–782.

Joshi, R., Nelson, G., and Randall, K. (2002). Denali: A Goal-directed Superoptimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 304–314.

Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., and Shippy, D. (2005). Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604.

Kisuki, T., Knijnenburg, P. M., O'Boyle, M. F. P., Bodin, F., and Wijshoff, H. A. (1999). A Feasibility Study in Iterative Compilation. In *International Symposium on High Performance Computing*, pages 121–132.

Kisuki, T., Knijnenburg, P. M. W., O'Boyle, M. F. P., and Wijshoff, H. A. G. (2000). Iterative Compilation in Program Optimization. In *Proceedings of the International Workshop on Compilers for Parallel Computing*, pages 35–44.

Knijnenburg, P. M. W., Kisuki, T., and O'Boyle, M. F. P. (2002). Iterative Compilation. In *Proceedings of Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation*.

Koza, J. R. (1990). *The Genetic Programming Paradigm: Genetically Breeding Populations of Computer Programs to Solve Problems*, pages 203–321. Wiley.

Kulkarni, S. and Cavazos, J. (2012). Mitigating the Compiler Optimization Phase-ordering Problem Using Machine Learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 147–162.

Leather, H., Bonilla, E., and O'Boyle, M. F. P. (2009a). Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 81–91.

Leather, H., O'Boyle, M. F. P., and Worton, B. (2009b). Raced Profiles: Efficient Selection of Competing Compiler Optimizations. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 50–59.

Li, Y.-T. S., Malik, S., and Wolfe, A. (1999). Performance Estimation of Embedded Software with Instruction Cache Modelling. *Design Automation of Electronic Systems*, 4(3):257–279.

Loveman, D. B. (1976). Program Improvement by Source to Source Transformation. In *Proceedings of the ACM SIGACT–SIGPLAN Symposium on Principles on Programming Languages*, pages 121–145.

MacKay, D. J. C. (1992). Information-based Objective Functions for Active Data Selection. *Neural Computation*, 4(4):590–604.

Martins, L. G., Nobre, R., Delbem, A. C., Marques, E., and Cardoso, J. M. (2014). Exploration of Compiler Optimization Sequences Using Clustering-based Selection. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, pages 63–72.

Martins, L. G. A., Nobre, R., Cardoso, J. M. P., Delbem, A. C. B., and Marques, E. (2016). Clustering-based Selection for the Exploration of Compiler Optimization Sequences. *ACM Transactions on Architecture and Code Optimizations*, 13(1):8:1–8:28.

Massalin, H. (1987). Superoptimizer: A Look at the Smallest Program. In *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 122–126.

Michael, J. (2006). Where's the evidence that active learning works? *Advances in Physiology Education*, 30(4):159–167.

Monsifrot, A., Bodin, F., and Quiniou, R. (2002). A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50.

Moore, D. S. and McCabe, G. P. (2005). *Introduction to the Practice of Statistics*. W. H. Freeman & Co.

Moss, E., Utgoff, P., Cavazos, J., Precup, D., Stefanovic, D., Brodley, C., and Scheeff, D. (1997). Learning to Schedule Straight-line Code. *Advances in Neural Information Processing Systems*, 10:929–935.

Mucci, P. J., Browne, S., Deane, C., and Ho, G. (1999). PAPI: A Portable Interface to Hardware Performance Counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*.

Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! In *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 265–276.

Neyman, J. (1937). Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 236(767):333–380.

Oak Ridge Leadership Computing Facility (2014). OpenCL Vector Addition. Retrieved 18 March 2018 from https://www.olcf.ornl.gov/tutorials/opencl-vector-addition/.

Online, T. F. (2017). Sequential Analysis: Design Methods and Applications. Retrieved 18 March 2018 from http://www.tandfonline.com/loi/lsqa20.

OpenCL (2012). *The OpenCL Specification (v1.2)*. Khronos OpenCL Working Group.

OpenMP (2011). *OpenMP Application Program Interface (v3.1)*. OpenMP Architecture Review Board.

Pan, Z. and Eigenmann, R. (2004). Rating Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 14.

Pan, Z. and Eigenmann, R. (2006). Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332.

Park, E., Cavazos, J., Pouchet, L.-N., Bastoul, C., Cohen, A., and Sadayappan, P. (2013). Predictive Modeling in a Polyhedral Optimization Space. *International Journal of Parallel Programming*, 41(5):704–750.

Park, E., Kulkarni, S., and Cavazos, J. (2011). An Evaluation of Different Modeling Techniques for Iterative Compilation. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 65–74.

Patterson, D. A. (2006). Future of Computer Architecture. Berkeley EECS Annual Research Symposium.

PCAST (2012). Engage to Excel: Producing One Million Additional College Graduates with Degrees in Science, Technology, Engineering, and Mathematics. Retrieved 18 March 2018 from https://obamawhitehouse.archives.gov/sites/default/files/microsites/ostp/pcast-engage-to-excel-final_2-25-12.pdf.

Petoumenos, P., Keramidas, G., Zeffer, H., Kaxiras, S., and Hagersten, E. (2006). STATSHARE: A Statistical Model for Managing Cache Sharing via Decay. In *Proceedings of the Annual Workshop on Modeling, Benchmarking and Simulation*.

Petoumenos, P., Mukhanov, L., Wang, Z., Leather, H., and Nikolopoulos, D. S. (2015). Power Capping: What Works, What Does Not. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems*, pages 525–534.

Pouchet, L.-N. (2012). PolyBench/C: The Polyhedral Benchmark Suite. Retrieved 18 March 2018 from http://polybench.sourceforge.net.

Pouchet, L.-N. (2015). PolyBench/C 4.0. Retrieved 18 March 2018 from https://sourceforge.net/p/polybench/wiki/Home.

Power, J., Basu, A., Gu, J., Puthoor, S., Beckmann, B. M., Hill, M. D., Reinhardt, S. K., and Wood, D. A. (2013). Heterogeneous System Coherence for Integrated CPU-GPU Systems. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 457–467.

Prince, M. (2004). Does active learning work? A Review of the Research. *Journal of Engineering Education*, 93(3):223–231.

Pusukuri, K. K., Gupta, R., and Bhuyan, L. N. (2012). Thread Tranquilizer: Dynamically Reducing Performance Variation. *ACM Transactions on Architecture and Code Optimization*, 8(4):46.

Qualcomm (2017). Snapdragon 835 Mobile Platform. Retrieved 18 March 2018 from https://www.qualcomm.com/products/snapdragon/processors/835.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc.

Rao, C. R. (1947). Factorial Experiments Derivable from Combinatorial Arrangements of Arrays. *Supplement to the Royal Statistical Society*, 9:128–139.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. The MIT Press.

Robbins, H. (1952). Some Aspects of the Sequential Design of Experiments. *Bulletin of the American Mathematical Society*, 58(5):169–177.

Ryan, C., Collins, J. J., and O'Neill, M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. In *Proceedings of the European Workshop on Genetic Programming*, pages 83–96.

Saar-Tsechansky, M., Melville, P., and Provost, F. (2009). Active Feature-value Acquisition. *Management Science*, 55(4):664–684.

Sanches, A. and Cardoso, J. M. P. (2010). On Identifying Patterns in Code Repositories to Assist the Generation of Hardware Templates. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 267–270.

Satterthwaite, F. E. (1946). An Approximate Distribution of Estimates of Variance Components. *Biometrics Bulletin*, 2(6):110–114.

Settles, B. (2013). *Active Learning*. Morgan & Claypool.

Seung, H. S., Opper, M., and Sompolinsky, H. (1992). Query by Committee. In *Proceedings of the Workshop on Computational Learning and Theory*, pages 287–294.

Shan, A. (2006). Heterogeneous Processing: A Strategy for Augmenting Moore's Law. *Linux Journal*, 2006(142):7.

Shannon, C. E. (1948). A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3):379–423,623–656.

Siebert, F. (2001). Constant-time Root Scanning for Deterministic Garbage Collection. In *Proceedings of the International Conference on Compiler Construction*, pages 304–318.

Siegmund, N., Kolesnikov, S. S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., and Saake, G. (2012). Predicting Performance via Automated Feature-interaction Detection. In *Proceedings of the International Conference on Software Engineering*, pages 167–177.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127.

Stehman, S. V. (1997). Selecting and Interpreting Measures of Thematic Classification Accuracy. *Remote Sensing of Environment*, 62(1):77–89.

Stephenson, M., Amarasinghe, S., Martin, M., and O'Reilly, U.-M. (2003). Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 77–90.

Student (1908). The Probable Error of a Mean. *Biometrika*, 6(1):1–25.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Taddy, M. A., Gramacy, R. B., and Polson, N. G. (2009). Dynamic Trees for Learning and Design. *Journal of the American Statistics Association*, 106(493):109–123.

Tartara, M. and Reghizzi, S. C. (2012). Parallel Iterative Compilation Using MapReduce to Speedup Machine Learning in Compilers. In *Proceedings fo the International Workshop on MapReduce and its Applications*, pages 33–40.

Thomson, J., O'Boyle, M. F. P., Fursin, G., and Franke, B. (2009). Reducing Training Time in a One-Shot Machine Learning-Based Compiler. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 399–407.

Tibshirani, R. (1996). Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statisitical Society, Series B (Methodological)*, 58(1):267–288.

Ting, P.-S., Tu, C.-C., Chen, P.-Y., Lo, Y.-Y., and Cheng, S.-M. (2017). FEAST: An Automated Feature Selection Framework for Compilation Tasks. In *Proceedings of the IEEE International Conference on Advanced Information Networking and Applications*, pages 1138–1145.

Tiwari, A., Chen, C., Chame, J., Hall, M., and Hollingsworth, J. K. (2009). A Scalable Auto-tuning Framework for Compiler Optimization. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12.

Triantafyllis, S., Vachharajani, M., Vachharajani, N., and August, D. (2003). Compiler Optimization-space Exploration. In *Proceedings of the Symposium on Code Generation and Optimization*, pages 204–215.

Tukey, J. W. (1977). *Exploratory Data Analysis*, chapter 2, page 43. Addison-Wesley Publishing.

Ullman, J. D. and Aho, A. V. (1977). *Principles of Compiler Design*, chapter 13, pages 471–472. Addison-Wesley Publishing Company.

Veeramachaneni, S. and Avesani, P. (2003). Active Sampling for Feature Selection. In *Proceedings of the IEEE International Conference on Data Mining*, pages 665–668.

Wald, A. (1944). Sequential Tests of Statistical Hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186.

Wang, Z. and O'Boyle, M. F. (2009). Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 75–84.

Wang, Z., Tournavitis, G., Franke, B., and O'Boyle, M. F. P. (2014). Integrating Profile-driven Parallelism Detection and Machine-learning-based Mapping. *ACM Transactions on Architecture and Code Optimization*, 11(1):2.

Weinstein, L. and Adam, J. A. (2008). *Guesstimation: Solving the World's Problems on the Back of a Cocktail Napkin*. Princeton University Press.

Welch, B. L. (1947). The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika*, 34(1–2):28–35.

Whaley, R. C., Petitet, A., and Dongarra, J. J. (2005). Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27:3–35.

Whiteson, S., Stone, P., Stanley, K. O., Miikkulainen, R., and Kohl, N. (2005). Automatic Feature Selection in Neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1225–1232.

Wolf, M. E. (1992). *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Computer Systems Laboratory, Stanford University, Stanford, CA.

Wolf, M. E., Maydan, D. E., and Chen, D.-K. (1996). Combining Loop Transformations Considering Caches and Scheduling. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 274–286.

Wolfe, M. (1987). Iteration Space Tiling for Memory Hierarchies. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361.

Wolfe, M. (1989). More Iteration Space Tiling. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 655–664.

Wu, J., Shen, Y.-L., Reinhardt, K., Szu, H., and Dong, B. (2013). A Nanotechnology Enhancement to Moore's Law. *Applied Computational Intelligence and Soft Computing*, page 2.

Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P., and Wu, P. (2003). A Comparison of Empirical and Model-driven Optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–76.

Yotov, K., Li, X., Ren, G., Garzaran, M. J., Padua, D., Pingali, K., and Stodghill, P. (2005). Is Search Really Necessary to Generate a High-Performance BLAS? *Proceedings of the IEEE*, 93(2):358–386.

Zhao, M., Childers, B. R., and Soffa, M. L. (2005). A Model-based Framework: An Approach for Profit-driven Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 317–327.

Zhuravlev, S., Blagodurov, S., and Fedorova, A. (2010). Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 129–142.

Zuluaga, M., Krause, A., and Püschel, M. (2016). ε-PAL: An Active Learning Approach to the Multi-objective Optimization Problem. *The Journal of Machine Learning Research*, 17(1):3619–3650.

Zuluaga, M., Krause, A., Sergent, G., and Püschel, M. (2013). Active Learning for Multi-objective Optimization. In *Proceedings of the Internation Conference on Machine Learning*, pages 462–470.