Adaptive Java Optimisation using Instance-Based Learning

Shun Long and Michael O'Boyle
Institute for Computing Systems Architecture
The University of Edinburgh
Edinburgh UK
Shun.Long@ed.ac.uk, mob@inf.ed.ac.uk

ABSTRACT

This paper describes a portable, machine learning-based approach to Java optimisation. This approach uses an instance-based learning scheme to select good transformations drawn from Pugh's Unified Transformation Framework[11]. This approach was implemented and applied to a number of numerical Java benchmarks on two platforms. Using this scheme, we are able to gain over 70% of the performance improvement found when using an exhaustive iterative search of the best compiler optimisations. Thus we have a scheme that gives a high level of portable performance without any excessive compilations.

General Terms

Languages, Performance

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimisations*; I.2.6 [**Artificial Intelligence**]: Learning

Keywords

Java, optimisation space, adaptive optimisation, instance-based learning

1. INTRODUCTION

The demand for ever greater performance has led to an exponential growth in hardware performance and architecture evolution. Such a rapid rate of architectural change has placed enormous stress on optimising compiler technology. However, traditional approaches to compiler optimisations are based on hardwired static analysis and transformation which can no longer be used in a computing environment that is continually changing. What is required is an approach which evolves and adapts to applications and architectural change, without sacrificing performance.

This paper describes a machine learning-based approach to Java optimisation which utilises prior knowledge of successful optimisation. Java is a highly portable language yet this is often at the cost of poor performance [10][22]. As our approach automatically builds an optimisation strategy based on platform-specific experience, we now have a compiler framework that allows portable performance as well. Unlike feedback-directed and iterative compilation [12] approaches, we do not require multiple compilations and runs of a particular program in order to find the best optimisation. Rather, we record the execution time behaviour of each program with respect to selected transformations when it is executed. As this information is accumulated over time, our knowledge of how programs, transformations and this particular platform interact grows, allowing a compiler strategy that adapts and improves with time.

Our approach uses instance-based learning within Pugh's Unified Transformation Framework [11]. This framework provides a systematic description of a large optimisation space that includes most loop and array transformations. Using this scheme we are able to gain over 70% of the performance improvement found when using an exhaustive feedback-directed search using iterative compilation. Thus we have a scheme that gives a high level of portable performance without any excessive compilations.

The outline of this paper is as follows. Section 2 provides a motivating example and section 3 describes the features used by the learning technique. Section 4 presents the instance-based learning optimisation approach. Section 5 evaluates the performance of this approach, followed by a review of related work in section 6 and some concluding remarks.

2. MOTIVATION AND EXAMPLE

The basic idea is to make a decision on how to optimise a particular program based on previous experience. Each time a program is optimised, the transformations applied and the resulting performance are stored along with a description of the program. On encountering a new program to optimise, the database of previous cases are searched for similar programs. Transformations that were beneficial to similar programs are then considered for the current program at hand. This idea has long been used in static compilation analysis which usually examines a few key features of a program to see if it fits a model for which an optimisation is known.

To illustrate this, consider the highly simplified example

ICS'04 June 26–July 1, 2004, Malo, France

A	**B**

`for (i=0; i<100; i++) {`
`0: a[i] *= b[i];`
`1: c[i] += d;`
`}`
	`for (i=0; i<100; i++)`
	`for (j=0; j<100; j++) {`
	`0: a[i][j] += b[i][j];`
	`1: c[i][j] *= a[i][j+1];`
	`}`

Features: *lnd*=1; *a*=3; *r*=3; *s*=2; *c*=false	Features: *lnd*=2; *a*=3; *r*=4; *s*=2; *c*=true
Transformation 0: (reversal) T_0: [i]→[0,-i,0] T_1: [i]→[0,-i,1] Speedup: 0.95 Transformation 1: (statement reordering) T_0:[i]→[0,i,1] T_1:[i]→[0,i,0] Speedup: 1.04	Transformation 0: (skewing) T_0: [i,j]→[0,i,0,i+j,0] T_1: [i,j]→[0,i,0,i+j,1] Speedup: 0.87 Transformation 1: (tiling with 20 and 15) T_0:[i,j]→[0,20*(i div 20),0,15*(j div 15),0,i,0,j,0] T_1:[i,j]→[0,20*(i div 20),0,15*(j div 15),0,i,0,j,1] Speedup: 1.15
C	**D**

`for (i=1; i<100; i++)`
`for (j=1; j<100; j++) {`
`0: a[i][j] += b;`
`1: a[i][j] *= c[j-1]+c[j]+d[i][j];`
`}`
	`for (i=1; i<100; i++)`
	`for (j=1; j<100; j++) {`
	`0: a[i][j] /= b[i][j];`
	`1: b[i][j] *= a[i][j-1]+a[i][j+1];`
	`}`

Features: *lnd*=2; *a*=3; *r*=5; *s*=2; *c*=false	Features: *lnd*=2, *a*=2; *r*=5; *s*=2; *c*=true
Transformation 0: (reversal) T_0:[i,j]→[0,i,0,-j,0] T_1:[i,j]→[0,i,0,-j,1] Speedup: 0.92 Transformation 1: (distribution) T_0:[i,j]→[0,i,0,j,0] T_1:[i,j]→[1,i,0,j,0] Speedup: 1.16	

Figure 1: This figure shows four programs (A, B, C and D) together with their program features. These features are *lnd* (loop nest depth), *a* (number of arrays used), *r* (number of array references used), *s* (number of statements) and *c* (containing dependencies). The transformations applied and the speedup obtained are shown for A,B and C. The compiler must now determine the transformation to apply to D.

shown in Figure 1. The four main boxes show the code and details for four simple loops A, B, C and D. A, B and C have already been executed by the system and their performance noted for different transformations. Given this information, the question is how should we transform program D?

To do this we must first determine which program is most similar to D. Each program has a small list of features summarising key characteristics. In our example just five features are shown for each loop to simplify explanation: lnd is the loop nest depth, a the number of distinct arrays, r the number of array references, s the number of statements and c a flag denoting whether there are flow dependencies.

Comparisons are made between D and the three loops A, B and C. D matches A on just one feature $s = 2$, i.e. both have two statements in their loop bodies. B and C also have two statements in their loop bodies. In addition, like D, they both are double nested loops ($lnd(B) = lnd(C) = lnd(D) = 2$). In deciding between B and C, B matches D in another feature $c = true$ (i.e. there exists flow dependencies between the two statements in their corresponding loops), whilst C matches D in feature $r = 5$ (i.e. both contains five array references in the loop body). The flow dependency feature, c, is considered more important and we therefore we assign it greater weight. Hence B is considered the most similar program to D among the three.

We consider the two transformations applied to B and the resulting speedup. (The details of the transformation representation are given in section 3). The first transformation, skewing, has a speedup of 0.87, a slowdown, while the second transformation, tiling, has a speedup of 1.15. We therefore apply this tiling transformation to D.

This is a highly simplified example, where the program segments are just simple loops. However, this approach can be readily applied to general programs and sets of transformations.

The key characteristic of this approach is that optimisation decisions are based on real platform performance rather than a static model or compiler heuristic. Furthermore, as the number of programs encountered increases, so will the database of examples improving the performance of the scheme. The main technical questions are: how to describe a program in a manner that is useful for optimisation; how can we determine if one program is similar to another and how is an appropriate transformation selected? These are addressed in the following two sections.

3. FEATURES

A learning-based compiler has to correlate programs, the optimisations applied and the resulting performance improvement in a systematic manner. In machine learning terms, the inputs or features of the problem are a description of the program and transformation with the output being execution time. The problem features have to be formally specified to allow the application of instance-based learning. These features not only reveal the important details of the program, but also help a compiler to classify for later retrieval, as shown in [16][17].

3.1 Program features

The most suitable program abstraction depends on circumstances. Call-graphs, for instance, may be appropriate for an inlining-based optimisation. As we focus on array-based Java applications, it makes sense to concentrate on the loop and array structure of the program. In this paper we consider an abridged set of features used in [16], as listed below.

1. memory access
   (a) number of array references
   (b) number of arrays used
   (c) linear array access
   (d) array elements reuses across iteration
   (e) uniform data dependency
2. loop structure
   (a) loop nest depth
   (b) loop size (inner-most)
   (c) perfectly nested loop(s)
   (d) loop step(s)
   (e) abnormal exit(s)
3. code
   (a) number of arithmetic operations
   (b) number of method calls
   (c) conditional control structure in the loop
   (d) number of statements in the loop nest

These features can be readily obtained from either the program representation used in Pugh's Unified Transformation Framework described below or most compiler internal representations of a program. Our compiler automatically extracts these features from the Java program.

3.2 Transformations

Pugh's Unified Transformation Framework (UTF) [11] provides a uniform and systematic representation of iteration reordering transformations and their arbitrary combinations. It encompasses nearly all high level iteration reordering transformations found in the literature and state-of-the-art commercial compilers. Most importantly, it has a formal mechanism to represent and reason about each transformation and their arbitrary combinations, although no dataflow optimisation is included.

UTF considers a statement's iteration space as the set of iterations for which the statement will be executed. All the points in the space will be executed in a lexicographic order. Therefore, a loop reordering transformation or transformation sequence can be considered as a mapping from the old iteration space to a new one. For each statement in an n-nested loop, its mapping is expressed[11] as below:

$$T : [i_0, ..., i_m] \rightarrow [f_0, ..., f_n] | C \quad (1)$$

where i_0,..., i_m are iteration variables, f_0, ... f_n are functions (usually quasi-affine functions) of iteration variables, and C is an optional restriction. It represents the fact that, if condition C is true, iteration [i_0, ..., i_m] in the original iteration space is mapped to iteration [f_0, ..., f_n] in the new iteration space.

More specifically, a mapping has $n/2$ loop components (quasi-affine functions of iteration variables) in odd-numbered levels, and $n/2+1$ syntactic components (integer constants)

```
for (i=0; i<100; i++)
 for (j=0; j<100; j++) {
0:  a[i][j] += b;
 }
for (i=0; i<100; i++)
 for (j=0; j<100; j++) {
1:  a[i][j] *= c[j-1]+c[j]+d[i][j];
 }
```

Figure 2: Program C after loop distribution

in even-numbered levels. For example, consider program C in Figure 1. The second transformation is of the form:

$$T_0 : [i, j] \rightarrow [0, i, 0, j, 0], T_1 : [i, j] \rightarrow [1, i, 0, j, 0]$$

Each of the two mappings represents what happens to the two statements 0 and 1 after transformation with respect to the iterators i and j. The integer entries on the right hand side represent the syntactic structure after transformation. As they differ in the first entry 0 vs 1, this means that they do not share any outer iterators i.e. they have been loop distributed as shown in Figure 2. For further details on this notation, please see [11].

The variety within the mapping notation results in an optimisation space, in which each point stands for an arbitrary combination of iteration reordering transformations in a unified manner. This space is significant and large enough to contain useful minima points. The learning-based compiler is to explore this space in its search for the best points. The program features and transformation representation provide a formal, succinct method of describing the task of program optimisation in terms suitable for general machine learning. The next section describes how such a representation is used.

4. INSTANCE-BASED LEARNING OPTIMISATION (IBLO)

Instance-based learning [18] models a complex target function by a collection of less complex approximations. In the learning-based compiler case, when a new program is encountered, the instance-based learning optimisation (IBLO) approach identifies programs closest to it from previous evaluations. It then selects a suitable transformation based on prior information about them, before applying it to the new program. The three main tasks of IBLO are therefore classification, knowledge storage and transformation selection.

4.1 Classification

Classification is used to both record information about a program for later use and as means to detect similar previous cases when trying to optimise the current program. It is based on the observation that *programs sharing common characteristics usually benefit from the same transformation(s)*[14][15][16].

IBLO defines the *similarity* of a given program P to a given category C as in Equation (1).

$$similarity(P, C) = \sum_{i=0}^{n} (w_i \times match(f_i(P), f_i(C))) \quad (2)$$

where n+1 is the number of program features, w_i are weights assigned to these features, $f_i(P)$ and $f_i(C)$ are the values of feature f_i for P and C respectively. $match$(a, b) is 1 if a equals b, 0 otherwise.

If, $similarity(P, C) = \sum_{i=0}^{n} w_i$, then P *belongs to* category C, otherwise, P is *similar to* C. Clearly, the higher the value of *similarity(P,C)*, the more similar P is to C, and vice versa.

For example, consider the four code segments in Figure 1, if equal weight 1 is given to all 5 features, we have similarity(A,D)=1, similarity(B,D)=3, from which we conclude that B is more similar to D than A is. If there is another program E whose features are lnd=2, a=2, r=5, s=2 and c=true, we have similarity(D,E)=5, i.e. D and E belong to the same category.

Similarity between two programs and two categories are defined in a similar manner.

4.2 Knowledge storage

For each category, the compiler records all the transformations that have been applied to a program of this category as well as their results. They are grouped into *area*s according to their similarities. For instance, loop unrolling transformations differing only in the unrolling factor will be naturally grouped together. Each area stores information about the use of a specific transformation or transformation sequence. For example, one area is created for loop tiling, one for loop unrolling and another for transformation sequence of tiling and unrolling. Each point in the area stands for a transformation or transformation sequence in one program of this category, alone with a set of parameters and the corresponding runtime feedback.

This compiler database is currently implemented as a hash table which enables a quick program comparison with low overhead. If space utilisation is at a premium, only a fixed number of positive transformations that bring performance improvement are stored.

A category will not be created and stored until the compiler has encountered a candidate programs; this also applies to areas within each category. It is worth noting that the maximum number of categories depends on the number of program features as well as their values, rather than the number of training examples.

This storage approach is simple and efficient. It provides a fast and low-overhead classification based on a hash function of program features. This approach can be further improved via storing information selectively, merging similar categories, storing only the model built on the training examples when necessary, or deleting the least useful information periodically. If the amount of data were to grow excessively, it may need to be replaced by a more sophisticated data management mechanism, but this is beyond the scope of this paper.

4.3 Transformation selection

In order to select an appropriate transformation for a new program P, IBLO first locates the category P belongs to, or is most similar to, using the above classification mechanism. If there exists such a category C, the transformation for P will be chosen directly from its prior experience with programs in C. Otherwise, IBLO will find the most similar categories to base its decision on.

For a given category, the probability that a transformation provides a performance improvement to a program can be

considered as a function defined on the optimisation space. According to the k-nearest neighbour algorithm[18], for any point in this space, its performance improvement is determined by its neighbouring points. If the majority of them improve performance, it is statistically likely that the transformation this point represents can also bring performance improvement. Thus the more crowded an area a point is in, the more likely the transformation it stands for can improve the performance of a program in this category.

Given that different transformations have different performance impact to different programs, the significance S of a given area X is simply defined as below:

$$S(X) = \frac{1}{m+1} \sum_{i=0}^{m} f(x_i) \quad (3)$$

where X consists of a set of transformation points x_0, x_1, ... x_m and f is the performance improvement of each point.

In the cases where more than one category is chosen for the given program, a virtual category is constructed whose areas are obtained by merging the corresponding areas of these categories. For example, if, for a new program, IBLO finds two most similar categories C_0 (with 2 areas t and u) and C_1 (with 2 areas u and s), A virtual category will be constructed with 3 areas $t = C_0.t$, $u = C_0.u \bigcup C_1.u$, and $s = C_1.s$.

IBLO then selects a transformation from the chosen category by dividing area X evenly into a number of sub-areas in order to find the most crowded sub-area. This is repeated until the densest sub-area contains a small number (e.g 5 or 6) of points, from which the transformation is selected randomly. This heuristic random selection prevents overfitting [18] of the the optimisation space. Figure 3 demonstrates how this algorithm works. Suppose, for program D in Figure 1, the algorithm locates from its most similar program B a category, whose most crowded area X is a two-dimensional area corresponding to schedules of tiling the double-nested loop with various tile size combinations. X is divided evenly into four subareas which have 8, 3, 2 and 1 point respectively, as shown on the left. The most crowded subarea $X0$ is chosen for further division and the result is shown on the right. The resulting subareas $X00$, $X01$, $X02$ and $X03$ contain 2, 1, 2 and 3 points respectively. No more division is needed as each subarea now contains only a small number of points. Because $X03$ is the most crowded of all, the algorithm randomly selects one from these 3 points and uses the corresponding tiling schedule as the schedule for D.

4.4 Strategy

In order to optimise a new program effectively, there should be sufficient prior knowledge of previous optimisations on which to base a judgment. There are many ways this can be achieved. One reasonable approach would be to initially allow an existing high level restructurer to optimise each new program and to record its behaviour. After a suitable number of cases were optimised, the learning approach could be applied. This has the advantage that if the existing approach is reasonable, no time is wasted selecting transformations that incur significant slowdown. The main problem with this approach is that although a wide range of different programs will be encountered, the transformation selection is hard-wired into the restructurer preventing a large enough dataset to evaluate transformation behaviour.

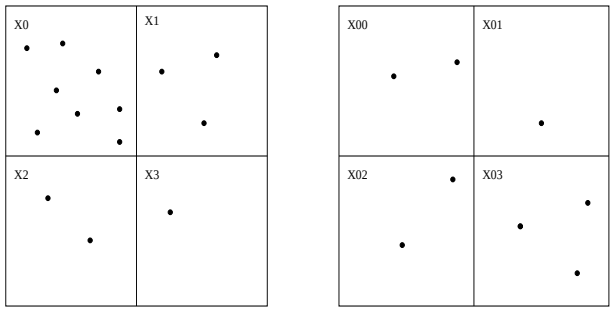

Figure 3: Example of transformation selection algorithm: an area X (left) is evenly divided into 4 subareas, among which X0 is the most crowded one. Therefore X0 is further divided into 4 subareas, from which the transformation is chosen from the most crowded one, i.e. X03.

An alternative approach is to try many transformations on a set of suitably chosen programs or *training examples*. The transformations could be selected randomly, or by a range of competing pre-existing analyses. This has the benefit of exploring the program and transformation space at the expense of potentially many worthless optimisations. However, as the training examples were suitably short in execution time, this learning phase could be considered as part of porting the system and is a one off cost.

We adopt this later scheme and use IBLO in a continuous process which allows adaptation to system upgrade. During the initial training phase, various transformations are randomly selected and applied to a set of programs, and the results are used as training examples. Later, when a new program is encountered, a transformation will be selected and applied, the result of which is used as a new training example.

4.5 Discussion

It is worth noting that once trained, IBLO is easy and quick. It simply extracts program features from the program, makes comparisons to locate the best category (constant time is needed with a hash table), splits the target area for several rounds, before selecting a transformation from the most crowded area.

Although IBLO can automatically adapt to a new machine, it does not apply the heuristics learned on one machine when optimising for another. To allow this, we would need to introduce a set of features that sufficiently characterise the architecture.

Once IBLO has been trained, it applies the learned heuristic to each program encountered but does not update its heuristic once training has been completed. Transformations for a new program are always selected from prior examples of the initial phase and do not utilise information about programs encountered after training. For instance, if IBLO has not learned the combined transformation of loop tiling and unrolling during its training phase, it will never provide such a transformation afterward. Therefore, the set of available transformations remains intact although the set of program categories may keep increasing. This drawback of IBLO could be amended by a hybrid approach which integrates IBLO and heuristic random search[15], so that the

Program	From
kernel3	Livermore
kernel5	Livermore
kernel6	Livermore
kernel7	Livermore
kernel8	Livermore
kernel9	Livermore
kernel10	Livermore
kernel11	Livermore
kernel12	Livermore
kernel19	Livermore
doIteration	JGF::euler::doIteration(...)
runF	JGF::euler::calcutateF(..)
runG	JGF::euler::calculateG(..)
runR	JGF::euler::calcutateR(..)
runS	JGF::euler::calculateStateVar(..)
mm	300x300 matrix multiplication, also kernel21 of Livermore

Figure 4: Summary of Benchmarks Used

compiler can consider not only the transformation chosen by IBLO, but also its variants with different parameters or with additional transformations.

5. EVALUATION

The learning technique described in this paper was implemented in a Java restructuring compiler and evaluated on two separate platforms across a range of benchmarks. The experimental method used and the results for each platform are described below.

5.1 Method

The first platform contains a Celeron processor (533MHz) and 128M RAM with the Java 2 Runtime Environment and Java Hotspot Client VM (1.3.0) running under Redhat Linux 6.3. The second platform again has the Java 2 Runtime Environment with Java Hotspot Client VM (1.4.1.1_01) but contains a PentiumPro (200MHz) with 96M RAM running MS-Windows 2000.

Sixteen methods were chosen from the well known *Livermore* benchmark and *Java Grande Forum Benchmark Suite* (JGF)[6] for evaluation. They are summarised in Figure 4. The experiments were conducted on these sixteen benchmarks in a cross-validation[18] manner, a typical method of evaluating machine learning approaches. This means that for each benchmark, the system has previously seen and optimised the other fifteen benchmarks which act as *training examples*. This mimics the behaviour in a live system where an optimisation is based on previous experience.

Initially, the training examples were classified into different categories. Figure 5 summarises the resulting categories as well as their similarities. Later when considering a program to be optimised, if the category it belongs to contains other programs, IBLO simply chooses them as its starting point for selecting an appropriate transformation. Otherwise IBLO chooses as its source of transformation three benchmarks selected from similar categories which have most benefited from previous optimisations.

For example, if we wish to optimise kernel3, it belongs to category 0 which contains another program kernel5, so the

Category	Programs included	Most similar categories
0	kernel3, kernel5	6, 7, 11
1	kernel6	6
2	kernel7	4
3	kernel8	8, 9, 10
4	kernel9	2, 5, 7
5	kernel10	4
6	kernel11, kernel12	0, 7
7	kernel19	0, 6
8	doIteration	10
9	runF, runG, runS	4
10	runR	8
11	mm	9

Figure 5: Categories, benchmarks and the most similar categories

Program	Transformation source
kernel3	kernel5
kernel5	kernel3
kernel6	kernel11, kernel12
kernel7	kernel9
kernel8	runF, runG, doIteration
kernel9	kernel7, kernel10, kernel19
kernel10	kernel9
kernel11	kernel12
kernel12	kernel11
kernel19	kernel3, kernel5, kernel11
doIteration	runR
runF	runG, runS
runG	runF, runS
runR	doIteration
runS	runF, runG
mm	runF, runG, runS

Figure 6: Benchmarks and their transformation sources

transformation for kernel3 is based on kernel5. However, if we are optimising kernel9 (category 4), it has no other program in its category, thus we select three programs from categories 2, 5 and 7, i.e kernel7, kernel10 and kernel19. Figure. 6 summarises the transformation sources for these sixteen benchmarks in our cross-validation experiment.

Once one or more programs have been selected as suitable candidates to base an optimisation on, a transformation is selected as described in section 4.3. In this experiment, each of these training examples has previously been optimised and executed 100 times using a simple search strategy[15] with the resulting speedups recorded. A transformation is therefore selected from the area of the transformation space giving the best prior performance.

5.2 Linux

The experimental results for the Linux platform are shown in Figure 7 and 8. The *Average* column shows the speedup obtained on average when using IBLO [1]. In every case speedup

[1]As the final selection of transformation is stochastic, average behaviour is presented

Program	IBLO			Search
	Best	**Average**	SD	**Best**
kernel3	1.04	1.03	0.0035	1.09
kernel5	1.09	1.07	0.0106	1.14
kernel6	1.20	1.16	0.0111	1.17
kernel7	1.04	1.02	0.0075	1.06
kernel8	1.29	1.24	0.0255	1.29
kernel9	1.09	1.07	0.0172	1.21
kernel10	1.09	1.06	0.0165	1.13
kernel11	1.41	1.33	0.0604	1.45
kernel12	1.07	1.03	0.0258	1.08
kernel19	1.07	1.03	0.0258	1.07
doIteration	1.05	1.04	0.0053	1.06
runF	1.05	1.03	0.0072	1.07
runG	1.08	1.07	0.0049	1.09
runR	1.09	1.05	0.0096	1.09
runS	1.01	1.01	0.0020	1.02
mm	1.69	1.35	0.0784	1.21
Average	1.15	**1.10**	0.0195	**1.14**

Figure 7: Comparison between heuristic search and learning in Linux, which shows that on average, 71% of the performance improvement found via the iterative search can be obtained by IBLO within just one attempt.

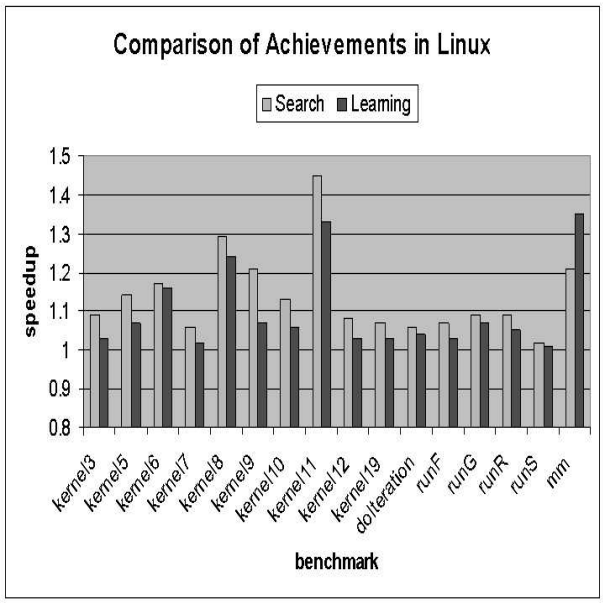

Figure 8: In Linux, IBLO achieves similar results to the heuristic search algorithm

is achieved. Given the small standard deviation of performance (as shown in the SD column), this demonstrates that a learning technique that selects an optimisation based on prior knowledge is capable of delivering consistent performance improvement. However, there are still some benchmarks which IBLO fails to provide significant performance improvement, for instance, kernel3, kernel7, kernel12, kernel19, runF and runR. Yet higher performance is available as shown in the *Best* column. For example, kernel12 has a modest average performance improvement, 1.03, but IBLO can select a transformation with double the performance, 1.07.

We wish to compare the performance of IBLO against other approaches. As there are no commercially available Java restructurers, we compare the algorithm against the extensive iterative search of the optimisation space using the simple search strategy[15]. This strategy selects transformation in a random manner, favouring simple and short transformation(s) than complex and longer ones. It gives excellent performance but at the cost of potentially 1000s of compile+run cycles, making it prohibitively expensive. It is unlikely that any compiler would approach the performance of such an extensive scheme and thus this is a rigorous test of our approach. The results of such a search are given in the column labelled *Search Best* where the best speedup found after 1000 evaluations is presented. The results show that, in six cases, IBLO is capable of achieving more than 85% of the performance of the exhaustive search and over 60% for three others with just one compilation. On average, the search technique gives a speedup of 1.14 and IBLO 1.10. Thus, if properly trained, IBLO is capable of achieving over 70% of the performance of the extensive search with just one compilation. Furthermore, if we examine the best performance of IBLO rather than the average, we see that it can outperform the search-based approach in some cases.

5.3 Windows

The experimental results in Windows are presented in Figure 9 and 10, which show that IBLO is again able to find performance improvement in the majority of cases. However, unlike Linux, there are a few cases where IBLO is unable on average to find improvement; even the best transformation fails to improve that of kernel19, runG and runS. If we examine the exhaustive search performance, we also see that it is unable to find a significant improvement in the majority of cases. Thus IBLO only fails to provide a significant improvement in those cases where there is none available.

The results show that, in eight cases, IBLO is capable of achieving more than 85% of the performance of the exhaustive search and over 60% for three others with just one compilation. Overall, it finds 90% of the available performance.

5.4 Comparison between Linux and Windows

IBLO achieves an average speedup of 1.10 in Linux and 1.09 in Windows. In some cases there is a significant difference in the performance obtained. In the case of kernel9, for example, there is a difference of 0.28 (1.35 vs 1.07). IBLO achieves significantly higher speedup for kernel3, kernel9 and kernel12 under Windows than Linux. For the remaining four (kernel6, kernel8, kernel11 and mm), the average speedup is higher in Linux than in Windows. We believe this difference in performance improvement between these two platforms is mainly due to that the relatively cost of memory latency on Linux is greater and benefit more from cache restructuring based optimisations. However, on average the performance improvements are remarkably similar and show the general robustness of our technique. They show that, if properly trained, IBLO is able to deliver significant improvement relative to an exhaustive search with just one compilation.

Program	IBLO			Search
	Best	**Average**	SD	**Best**
kernel3	1.17	1.17	0.0000	1.18
kernel5	1.10	1.10	0.0000	1.10
kernel6	1.03	0.98	0.0041	1.09
kernel7	1.05	1.05	0.0000	1.05
kernel8	1.13	1.12	0.0050	1.14
kernel9	1.35	1.35	0.0020	1.37
kernel10	1.07	1.06	0.0028	1.06
kernel11	1.18	1.18	0.0000	1.18
kernel12	1.18	1.18	0.0000	1.19
kernel19	1.00	0.99	0.0040	1.01
doIteration	1.03	1.03	0.0028	1.05
runF	1.01	1.00	0.0035	1.02
runG	1.00	0.99	0.0040	1.01
runR	1.09	1.07	0.0087	1.09
runS	0.99	0.99	0.0000	1.01
mm	1.11	1.10	0.0020	1.14
Average	1.09	**1.09**	0.0024	**1.10**

Figure 9: Comparison between heuristic search and learning in Windows, which shows that, on average, 90% of the performance improvement found by the iterative search can be obtained by IBLO within just one attempt.

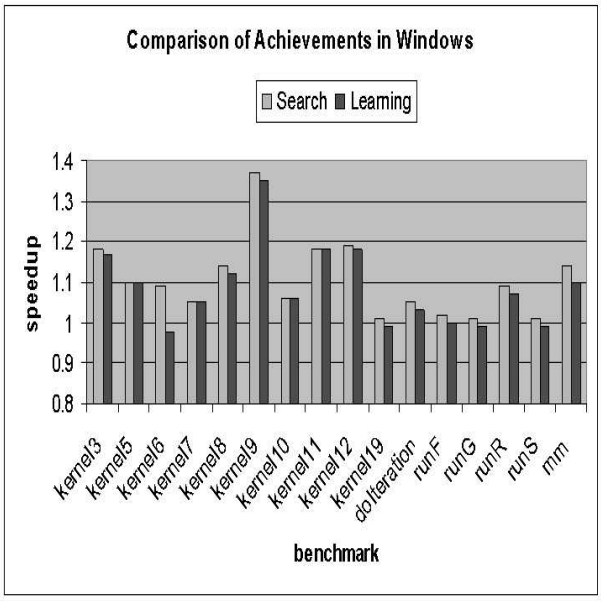

Figure 10: IBLO achieves similar results to heuristic searchin Windows

6. RELATED WORK

There have been a number of isolated attempts at employing machine learning within compiler research. Most of them aim at solving specific optimisation problems instead of steering optimising compiler at system-level. They vary in both cost and efficiency.

A greedy local instruction scheduling approach iteratively selects the best instruction from those available. This task is considered in [19] as a supervised process of learning preference relations over triples of partial schedules. A number of classic machine learning approaches are applied in [19]. This achieves good results though it relies on the hand coding of processor specific features that would not port to other platforms. It avoids the more difficult cyclic code structures and could not improve its performance over time.

In [17] a simple learning scheme has been used to determine whether unrolling is a useful transformation for any given loop. Although general loops are considered, the approach effectively models decisions as linear hyperplanes in a transformations space. Training cases sharing common characteristics are grouped into classes. Each class is then labelled either positive if unrolling improves performance on the majority of the class, or negative otherwise. In this way, the learning task is cast into building a decision tree, each leaf of which represents an unrolling heuristic and the corresponding test on values of loop features. This approach has a main drawback in that it works for simple single transformations with polynomial behaviour but may fail for more complex coupled spaces [12].

Many digital signal processing transforms can be represented as matrices. Different factorisations of these matrices vary significantly in runtime performance. They are represented as split trees in SPIRAL[23, 24], which explores the factorisation space of the Walsh Hadamard Transform (WHT) with a stochastic evolutionary algorithm. Reinforcement learning is used in [23] to learn to construct fast WHT formulae for many data sizes after training on one data size. This achievement comes at the cost of checking a large number of formulae, and its applicability has not been confirmed on other DSP transforms.

Genetic algorithm[8] or stochastic approaches are known to be effective machine learning approaches. For instance, [7] tries to solve the phase order problem by selecting transformations from a pool and applying them in a global manner. The selection is steered by an adaptive genetic algorithm which uses a probabilistic model built on runtime feedback. Genetic programming, a particular form of genetic algorithm, is used in [25, 26] to optimise priority functions which a compiler uses to decide the best optimisation heuristics. Another example of genetic algorithms is in the case of instruction scheduling. Many list scheduling approaches assign to each node a weighted sum of key parameters as its priority. 24 parameters are used in [4] for a genetic algorithm to search for the optimal weight combination for a given scheduler/architecture pair. The results show that genetic algorithm can bring adaptability to the instruction scheduler.

Case-based reasoning has been used in [16] to complement existing compilers and automatic parallelisers and help users with the performance tuning process. It is very similar in spirit to IBLO in that it uses a set of code properties as indices of loop structure and code patterns in order to identify potential optimisation opportunities. However, unlike IBLO, it is not fully automatic. The compiler writer needs to specify in advance the transformations and the conditions under which they should be applied. In addition, no new knowledge can be obtained from runtime feedback. Finally in the area of machine learning, reinforcement learning is used in [3] to adaptively optimise the performance of conventional garbage collection techniques, based on memory allocation information of the running application.

Closely related to learning the best optimisation is the task of searching for the best. In iterative compilation [12] the compiler tries to optimise a program by repeatedly executing different versions of it and using the feedback to decide the next optimisation attempt. Various approaches have been developed to explore different optimisation spaces, for instance, genetic algorithm [7][20], tree-/grid-based search [12][21], phasewise exhaustive search and random search [9], etc. These approaches vary in the efficiency but can require hundreds of iterations to gain a significant performance improvement. A compiler framework is presented in [13] in which program information can be collected and stored in a unified, low-level and typed representation in order to enable life-long code optimisation. The machine learning approach described in this paper would naturally fit into this scheme.

Finally, Java optimisation can be achieved via an efficient virtual machine [2], or optimisation techniques such as JIT-compilation[1] and parallelisation[5]. The virtual machine approach is inevitably architecture-specific, parallelisation approach also relies on architectural support, whilst JIT compilation considers only light-weighted optimisations.

7. CONCLUSIONS

This paper has described a machine learning based approach to optimising Java programs. It has shown that instance-based learning is a viable automatic approach to building an optimising strategy. It is inherently portable, adapting to any platform based on actual machine performance rather than static analysis. It has shown that, if properly trained (a modest fixed one-off training cost), such an approach can achieve over 70% of the performance available when using aggressive exhaustive search approaches.

Due to the program features used and the high-level transformations considered by UTF, IBLO is now restricted to loop- and array-intensive Java programs. With the help of a powerful compiler framework and other enhancements discussed below, it is believed applicable to generic programs. Furthermore, with the experience accumulated during and after the training period, it is believed able to optimise difficult real world applications over time.

Future work will investigate optimisations outside the UTF framework and consider more general Java programs. More program and architecture features will be included to examine IBLO's applicability across various platforms. In order to put IBLO into practice, we shall also consider training example selection in order to balance the tradeoff between the training set size and the efficiency of IBLO. We will also examine other machine learning approaches such as Gaussian process prediction as an alternative means of predicting performance.

8. REFERENCES

[1] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh and J. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. The 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), 1998.

[2] B. Alpern, A. Cocchi, D. Lieber, M. Mergen and V. Sarkar. Jalapeno - a compiler-supported Java virtual machine for servers. Workshop on Compiler Support for Software System (WCSSS99), 1999.

[3] E. Addreasson, F. Hoffman and O. Lindholm. To collect or not to collect? machine learning for memory management. The 2nd Java Virtual Machine Research and Technology Symposium (JVM'02), 2002.

[4] S. Beaty and S. Colcord. Using genetic algorithm to fine-tune instruction scheduling heuristics. International Conference on Massively Parallel Computer Systems, 1996.

[5] A. Bik and D. Gannon. javar - a prototype Java reconstructing compiler. Concurrency, Practice and Experience. 9(11), 1997.

[6] M. Bull, L. Smith, M. Westhead, D. Henty and R. Davey. A benchmark suite for high performance Java. Concurrency, Practice and Experience, 12, 2000.

[7] K. Cooper, D. Subramanian and L. Torczon. Adaptive optimizing compilers for the 21st century. Journal of Supercomputing, 2001.

[8] L. Davis. Handbook of genetic algorithm. Van Nostrand Reihold, 1991.

[9] G. Fursin, M. O'Boyle and P. Knijnenburg. Evaluating iterative compilation. The 15th Workshop on Languages and Compilers for Parallel Computers (LCPC'02), 2002.

[10] Java Grande Forum. Making Java work for high-end computing. SC98: High Performance Networking and Computing, 1998.

[11] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical report of University of Maryland, CS-TR-3193, 1993.

[12] T. Kisuki, P. Knijnenburg and M. O'Boyle. Combined selection of tile sizes and unroll factors Using iterative compilation. The 2000 International Conference on Parallel Architecture and Compilation Techniques (PACT'00), 2000.

[13] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. The 2004 International Symposium on Code Generation and Optimisation (CGO'04), 2004.

[14] S. Long and M. O'Boyle. Towards an adaptive Java optimising compiler, an empirical evaluation of program transformations. The 3rd Workshop on Java for High Performance Computing, 2001.

[15] S. Long. Adaptive Java optimisation using machine learning techniques. PhD thesis, School of Informatics, The University of Edinburgh. 2004.

[16] A. Monsifrot and F. Bodin. Computer Aided Hand Tuning (CAHT): applying case-based reasoning to performance tuning. The 15th ACM International Conference on Supercomputing (ICS'01), 2001.

[17] A. Monsifrot, F. Bodin and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. The 10th International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA), 2002.

[18] T. Mitchell. Machine learning. McGraw-Hill, 1997.

[19] E. Moss, P. Utgoff, J. Cavazos, D. Precup, D. Stefenovic, C. Brodley and D. Scheeff. Learning to schedule straight-line code. Neural Information Processing Systems, 1997.

[20] A. Nisbet. GAPS: iterative feedback directed parallelisation using genetic algorithms. The 12nd ACM International Conference for Supercomputing (ICS'98), 1998.

[21] M. O'Boyle, P. Knijnenburg and G. Frusin. Feedback directed iterative compilation. The 15th Workshop on Languages and Compilers for Parallel Computing (LCPC'02), 2002.

[22] J. Shirazi. Java performance tuning. O'Reilly, 2002.

[23] B. Singer and M. Veloso. Learning to generate fast signal processing implementations. The International Conference on Machine Learning (ICML-2001), 2001.

[24] B. Singer and M. Veloso. Stochastic search for signal processing algorithm optimization. Scientific Computing (SC2001), 2001.

[25] M. Stephenson, U. O'Reilly, M. Martin and S. Amarasinghe. Genetic programming applied to compiler heuristic optimisation. The 6th European Conference on Genetic Programming, 2003.

[26] M. Stephenson, S. Amarasinghe, M. Martin and U. O'Reilly. Meta optimisation: improving compiler heuristics with machine learning. The 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03), 2003.