Systematic search within an optimisation space based on Unified Transformation Framework *

Shun Long**
Institute for Computing Systems Architecture,
School of Informatics, The University of Edinburgh
Edinburgh EH9 3JZ, United Kingdom
E-mail: slong@inf.ed.ac.uk
**Corresponding author

Grigori Fursin
INRIA Futurs, Parc Club Orsay Universite,
91893 Orsay Cedex, France
E-mail: grigori.fursin@inria.fr

Abstract: Modern compilers have limited ability to exploit the performance improvement potential of complex transformation compositions. This is due to the ad-hoc nature of different transformations. Various frameworks have been proposed to provide a unified representation of different transformations, among them is Pugh's Unified Transformation Framework (UTF) (Kelly and Pugh (1993)). It presents a unified and systematic representation of iteration reordering transformations and their arbitrary combination, which results in a large and complex optimisation space for a compiler to explore. This paper presents a heuristic search algorithm capable of efficiently locating good program optimisations within such a space. Preliminary experimental results on Java show that it can achieve an average speedup of 1.14 on Linux+Celeron and 1.10 on Windows+PentiumPro, and more than 75% of the maximum performance available can be obtained within 20 evaluations or less.

Keywords: iterative compilation, adaptive optimisation, Unified Transformation Framework, Java, optimisation space, heuristic search.

Biographical notes: Shun Long is a member of Compiler and Architecture Design group at the the University of Edinburgh, Edinburgh, United Kingdom. He completed his PhD degree from the University of Edinburgh in 2004. Since 1999 he has been researching in iterative optimizations, learning-based optimizations and Java optimizing compilers.
Grigori Fursin is a member of the Alchemy group at INRIA Futurs, France. He completed his PhD degree from the University of Edinburgh in 2004. Since 1999 he has been researching in iterative and adaptive optimizations, performance prediction techniques, self-tuning compilers and machine-learning techniques.

1 Introduction

The demand for greater performance has led to an exponential growth in hardware performance and architecture evolution. In order to fully exploit the hardware potential in search for high performance, an optimising compiler (Kennedy and Allen (2002)) usually applies various transformations at different levels. Previous work (Parello et al. (2002)) demonstrates that complex transformation compositions can bring significant performance improvement. However, traditional optimising compilers are based on static analysis and a hardwired compilation strategy.

*A preliminary version of this paper, titled "A heuristic search algorithm based on Unified Transformation Framework", has been presented in the 7th International Workshop on High Performance Scientific and Engineering Computing (ICPP-HPSEC 2005), Norway, 2005

They have difficulties in coping with this complexity of transformation combination, which usually appears in the form of a large and complex optimisation space. Iterative optimisation (Bodin et al. (1998), Kisuki et al. (1999), Fursin et al. (2002), Fursin (2004)) is therefore introduced to explore such spaces. However, many current iterative optimisation approaches either target kernels or consider only a small set of transformations, therefore difficult to extend and perform long sequences of composed transformations.

A unified representation (Bastoul et al. (2003)) of various transformations allows the compiler to explore an optimisation space in a systematic manner. Various representations have been purposed, among them is the Unified Transformation Framework (UTF) (Kelly and Pugh (1993)). It presents a unified and systematic representation of iteration reordering transformations and their arbitrary combinations, which aim to improve memory locality and explore parallelism. This results in a large and complex optimisation space for a compiler to explore, as demonstrated later.

Java's architecture independent design makes it ideal for software development in a modern computing environment. However, this means that Java is frequently unable to deliver high performance. Many approaches (Adl-Tabatabai et al. (1998), Alpern et al. (1999), Bik and Gannon (1997), Moreira et al. (1998)) have been proposed to improve Java's runtime performance. In Long and O'Boyle (2001), we show the performance improvement available for loop reordering transformations on Java programs.

This paper presents a heuristic search algorithm to explore the UTF-based optimisation space. The experimental results on Java show that this algorithm is able to locate good points in a remarkably small number of attempts. However, when optimizing large programs, the cost of the iterative search can still be high and therefore, such programs should be profiled first to focus the search on their hotspots only. To further lower the search cost, we discuss the potential of using machine learning techniques and suggest to use our heuristic search algorithm to select training examples in large optimization spaces.

The outline of this paper is as follows. Section 2 specifies the optimisation space with the help of the UTF. The heuristic search algorithm and its experimental results are presented in section 3. Section 4 discusses related work. Section 5 briefly discusses the potential of using this algorithm to select good training examples for an instance-based learning optimisation approach. It is followed by some concluding remarks in section

2 The Problem

We wish to search a large program transformation space and develop a search algorithm which can find the transformation sequence(s) that give the best performance improvement with the fewest number of evaluations. The critical issues are: what transformation space are we to consider and how is it to be represented? It must be significant and large enough to contain useful minima points and have a representation that allows a systematic search. Previous work (Bodin et al. (1998), Fursin et al. (2002)) has focused on search strategies based on highly restricted optimisation spaces.

The Unified Transformation Framework (UTF) (Kelly and Pugh (1993)) provides a uniform and systematic representation of iteration reordering transformations (loop interchange, reversal, skewing, distribution, fusion, alignment, interleaving, tiling, coalescing, scaling, together with statement reordering and index set splitting) and their arbitrary combinations. It encompasses nearly all the high level loop and array based transformations found in the literature and state-of-the-art commercial compilers.

A transformation is considered by UTF as a *schedule* mapping the old iteration space to the new one. For each statement in an n-nested loop, its mapping has n loop components (quasi-affine functions of iteration variables) in odd-numbered levels, and $n+1$ syntactic components (integer constants) in even-numbered levels. An example is shown in Figure 1 where the original double nested loop and its default schedule (T_0 and T_1) are shown in A). When loop interchange is applied, i and j in both T_0 and T_1 in A) are swapped to denote the interchange, as shown in B). If distribution is then applied, the *0* and *1* in the last and 5th column of T_0 and T_1 in B) are moved to the 3rd column, as shown in C). If skewing on statement 1 is then applied, the i in T_1 in C) is changed to $i+j$, as shown in D).

It is worth noting that no UTF transformations, except tiling, changes a mapping's length if applied. For example, the mappings of program B, C and D are of the same length as those of the original program A, as shown in Figure 1. In addition, only unrolling duplicates the loop body when applied, which introduces more coefficients to the schedule.

Figure 1 shows that, using the schedule notation, UTF can represent a sequence of iteration reordering transformations as a sequence of parameters (integers in the syntactic components and coefficients in the loop components). In this manner, the optimisation space composed of arbitrary combinations of these transformations is turned into a polyhedral space composed of all the integer parameters in the loop and syntactic components. This polyhedral space is considered more convenient for a systematic exploration than the original one.

A compiler has to explore this polyhedral space for performance improvement. This space is large, considering its dimension (the number of parameters) and the potential range of each dimension. For example, if the tile sizes are allowed to vary from 1 to 10, unrolling factors from 1 to 20, the integer coefficients from -5 to 5, there are over 10^{10} points to consider for the original loop in Figure 1. Long (2004) presents an exhaustive scan algorithm which can reach every single point within this space, if given enough time and resources. However, as the space contains many points either illegal or degrade performance, it is clear that any realistic search algorithm will have to focus on areas

A) original program

for (int i=0; i<1024; i++)
  for (int j=0; j<2048; j++) {
  0: b[i][j] = c[i] + d[j];
  1: a[i][j] = c[j] + d[j];
  }

T_0: [i, j] → [0, i, 0, j, 0] T_1: [i, j] → [0, i, 0, j, 1]
B) interchange is applied

for (int j=0; j<2048; j++)
  for (int i=0; i<1024; i++) {
  0: b[i][j] = c[i] + d[j];
  1: a[i][j] = c[j] + d[j];
  }

T_0: [i, j] → [0, j, 0, i, 0] T_1: [i, j] → [0, j, 0, i, 1]
C) distribution is applied

for (int j=0; j<2048; j++) {
  for (int i=0; i<1024; i++) {
  0: b[i][j] = c[i] + d[j]; }
  for (int i=0; i<1024; i++) {
  1: a[i][j] = c[j] + d[j]; }
}

T_0: [i, j] → [0, j, 0, i, 0] T_1: [i, j] → [0, j, 1, i, 0]
D) skewing is applied

for (int j=0; j<2048; j++) {
  for (int i=0; i<1024; i++) {
  0: b[i][j] = c[i] + d[j]; }
  for (int i=j; i<j+1024; i++) {
  1: a[i-j][j] = c[j] + d[j]; }
}

T_0: [i, j] → [0, j, 0, i, 0] T_1: [i, j] → [0, j, 1, i+j, 0]

Figure 1: Loop transformations and resulting codes

where legal points aggregate, and if possible, where points of performance improvements aggregate.

Such an optimisation space is not only large but also complex. Kisuki et al. (1999) demonstrate that even with only two transformations in tiling and unrolling, the resulting subspace is highly non-linear and contains many local minima as well as some discontinuities. It is infeasible to analyse or predict the performance and to pick good points from the space using static approaches. A reasonable alternative is a search algorithm which uses its prior results and heuristics to direct its search, in order to locate good points quickly.

3 Heuristic Search

Due to the size of the above optimisation space, it is essential to develop an efficient search algorithm. To be portable, this algorithm should not contain any hardwired knowledge about the architecture and environment. As there is no prior knowledge about where the good points locate in the space, it should theoretically consider all points in the space if given unlimited time and resources, although practically this is infeasible and unnecessary. Therefore, the search algorithm has to make a tradeoff between efficiency and coverage. In order to find good points in the space quickly, the algorithm should direct its search based on runtime feedback.

Furthermore, a compiler can use appropriate machine learning techniques to accumulate optimisation experience from these good transformations. Later when a new program is encountered, the compiler can apply its experience to find good transformation without any iterative search. This approach will be discussed in section 4.

3.1 Additional notation

The loop and syntactic components of a mapping are grouped into two vectors named *loop vector* and *syntactic vector* respectively. The syntactic vector SV is a vector of integer constants. Loop vector LV is a vector of linear functions of all the original iteration variables or derived ones introduced by tiling, namely i_0, i_1, ... i_x. Intuitively, varieties in loop vectors are associated with transformations such as tiling, unrolling, skewing, reversal, alignment, and scaling etc., and varieties of syntactic vectors are associated with transformations such as loop fusion, distribution and statement reordering, etc.. For instance, in Figure 1, when loops i and j in A) are interchanged, the syntactic vectors ((0,0,0) and (0,0,1)) remain unchanged whilst the loop vectors change from (i,j) to (j,i), as shown in B). When the loop is then distributed, the syntactic vectors are changed to (0,0,0) and (0,1,0), as shown in C), whilst the loop vectors (j,i) remain unchanged.

Loop vector LV is presented as $LV{=}I{\times}M$ where I is an $(x{+}2)$-dimensional vector of the iteration variables and M an $(x{+}2){\times}(x{+}1)$ matrix of integer constants. For example, the mappings of D) in Figure 1 can be represented as follows.

$$T_0 : LV_0 = (j,i) = (i,j,1) \times \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix}, SV_0 = (0,0,0) \tag{1}$$

$$T_1 : LV_1 = (j,i{+}j) = (i,j,1) \times \begin{pmatrix} 0 & 1 \\ 1 & 1 \\ 0 & 0 \end{pmatrix}, SV_1 = (0,1,0) \tag{2}$$

Given a loop vector LV, its *default schedule* transforms the code block in a way that all the statements in the loop(s) remain in their original positions. Intuitively, this means that transformations such as statement reordering, loop distribution and fusion are not considered in the default schedule.

3.2 Search Strategy

In order to find good points in the above optimisation space quickly, the heuristic search algorithm uses the following search strategies:

2-phase mapping construction Using the above notations, mapping construction can be considered as a two phase process. In the first phase, the iteration variable vector I and the matrix M are decided and the loop vector LV is then obtained by $LV{=}I{\times}M$. The second phase decides the syntactic vector SV. The mapping is then constructed by interleaving the elements of SV and LV.

UTF requires that each statement in the loop be assigned a separate mapping. Theoretically, their mappings do not necessarily share a common loop vector as in the default schedule case discussed above. In such cases, for the mapping of each statement, its loop vector must be decided separately, before its syntactic vector.

Random When there is no prior knowledge of the search space, random points are realistic starting points for the search algorithm. The search process over time is biased on weights to options found to be good in the previous attempts. For example, a random decision is made in each attempt on whether loop tiling should be included.

Runtime feedback (speedup in our case) is used to periodically review the decision bias during the search process. In each review, the weight of each option may be given a small increment if performance improvement is found, or a small decrement if degradation is found or when illegal schedule is constructed. In addition, these weights will be reset to default after a much longer period, in order to balance the tradeoff between efficiency and coverage.

Rotation The separation of loop and syntactic components divides the target search space into two subspaces associated with the loop and syntactic vectors, which are explored by *L-Search* and *S-Search* respectively. The L-Search focuses on loop vector subspace exploration. By default, it generates various loop vectors, and tests their default schedules. The S-Search explores the syntactic vector subspace. By default, it picks out a good loop vector and tests various schedules constructed by combining the loop vector with various syntactic vectors.

Since the L-Search and S-Search focus on different aspects of mapping construction, neither of them shall take sole control of the search process permanently. They are explored in roughly alternating manner. In each round, L-Search or S-Search evaluates a number of points in the space and collects the runtime profile, before relinquishing the control.

Simple first Although Parello et al. (2002) claim that complex transformation combinations can bring significant performance improvement, if we focus on the iteration reordering transformations UTF includes, we find that in most cases in Java, the majority of performance improvement comes from either one transformation or a combination of only a few (Long and O'Boyle (2001)). Therefore, the heuristic search algorithm should try simple schedules first. If no significant performance improvement is achieved, it will then consider complex schedules which may bring further improvement, as Parello et al. (2002) indicate. This search will continue and any arbitrary complex schedule UTF can represent will therefore be considered, as long as budget allows.

Intuitively, this strategy means that shorter and simpler transformation sequences are preferred to longer and more complex ones. For instance, it is known that tiling could be applied once or repeatedly, as UTF allows. The search algorithm shall consider cases where no tiling is applied or where tiling is applied only once before considering those of multiple tiling.

Loop before syntactic The loop vector variety is associated with transformations such as tiling, unrolling, skewing, reversal, alignment and scaling etc. Their arbitrary combinations indicate more varieties than those of transformations associated with syntactic vector subspace, i.e. fusion, distribution and statement reordering, etc. Therefore, the algorithm attempts to decide the loop vector first before considering the variety of syntactic vectors.

Window search The search algorithm should be flexible enough so that, if a good point is found, it will explore the surrounding subspace where even better points may reside. This strategy is very similar to the grid-based search algorithm used in Kisuki et al. (1999). It is worth noting that a balance shall be maintained between flexibility, efficiency and coverage, so that when no further improvement is found in the subspace, the search algorithm will turn to other areas within the optimisation space.

3.3 Search Algorithm

During the search process, both the L-Search and S-Search are explored in roughly alternating manner. In each round, L-Search or S-Search evaluates a number of points in the space and collects the runtime profile. This is coordinated by a steering module which keeps adjusting its decision according to runtime profile, as the pseudo code in Figure 2 demonstrates. *Budget*, *Lbudget* and *Sbudget* are compiler configuration constants. In the prototype, *Budget* is the number of iteration the heuristic search plans to take before stop. It is set 100 as explained later in Experimental Results.

```
HeuristicSearch(...)  {
  try and evaluate the 1st round of L-Search;
  try and evaluate the 1st round of S-Search;

  // while optimisation budget (Budget) allows
  repeat
  {  decide next search round;
     // based on evaluation results
     if (decision is "try L-Search")
       { try and evaluate L-Search; }
     else   // the decision is "try S-Search"
       { try and evaluate S-Search; }
  }
}
L-Search(...) {
  // for a certain number of times (Lbudget)
  repeat
  {  decide iteration variable vector I;
     decide transform matrix M;
     create a new loop vector LV=IxM;
     evaluate LV's default schedule S;
  }
}
S-Search(...) {
  choose a good schedule S;
  derive its loop vector LV;
  derive its syntactic matrix SM;
  // for a certain number of times (Sbudget)
  repeat
  {  derive new syntactic matrix SM' from SM;
     construct new schedule S' from LV and SM';
     evaluate S';
  }
}
```

Figure 2: Pseudo code of the heuristic search algorithm

L-Search The initial L-Search generates a certain number (*Lbudget* in Figure 2) of loop vectors and evaluates them using their default schedules. As described above, loop vector LV is determined by the iteration variable vector I and transform matrix M. I is in turn decided by tile size(s) and unrolling factor(s) randomly chosen from a suitable range, if tiling and/or unrolling is included.

Loop tiling is the only transformation that introduces derived iteration variables, with one new variable introduced by applying tiling to one loop once. Most compiler apply tiling just once. However, in the presence of a multi-level cache hierarchy, performance can be further improved if tiling is applied repeatedly. Therefore, for each loop in a loop nest, tile depth (the number of times tiling can be applied) is an arbitrary integer within a certain range, depending on the tile size(s) chosen and the original loop size(s). Random decisions are made on tile depth, favouring simple ones such as 1 (no tiling is applied) or 2 (tiling is applied once).

Loop unrolling is a transformation that duplicates the number of mappings for each statement via index splitting. Theoretically, all of the loops in the loop nest are considered potential candidates for unrolling. If it is included, the search algorithm must decide how many loops to unroll, which of them to unroll and the corresponding unroll factors. Following the "simple first" strategy, it will first consider unrolling only the innermost loop in the loop nest before considering unrolling other loops and more than one loops. The unrolling factor(s) will be chosen randomly from a certain range.

In order to follow the "simple first" strategy, it is preferable that simple transformation matrices are generated before complex ones. We consider a matrix $M = (ma)^T$ simple if m is an identity matrix or a matrix that can be obtained by applying one or a few steps of linear transformations on an identity matrix. Therefore, the transform matrix M is constructed by starting from an M^T=(m a) (where m is an identity matrix and a a zero vector), iteratively applying either linear transformations to m or assigning new value to a (both randomly decided) until a new one is generated. The loop vector LV is then constructed by multiplying I and M, as demonstrated in Equation (1) and (2).

In order to generate the default schedule of LV, each statement in the loop nest must be assigned a separate syntactic vector. This is done by constructing a default syntactic matrix SM, each row of which stands for a syntactic vector for a statement. For example, for loop vector LV = (j,i) in Figure 1(B), its default syntactic matrix is as shown in Equation (3).

$$SM = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3}$$

With both loop and syntactic vectors decided, LV's default schedule is constructed by allocating different syntactic vectors in SM to different statements in the original loop. Details of the algorithms to generate the default syntactic matrix and the default schedule can be found in Long (2004). This default schedule of LV is then tested for legality before the corresponding code being generated and tested. The profile (speedup) is collected and used to adjust the decision bias as explained above. This process is repeated for (*Lbudget*) times in each L-Search round.

Subsequently, the L-Search selects loop vectors of good performance found in previous rounds, constructs similar ones by keeping the I unchanged and combining it with different Ms generated in the same manner presented above. The resulting similar loop vectors are also tested using their default schedules, with the results used to adjust the decision bias.

S-Search The S-Search aims to explore the syntactic vector subspace. Initially, it chooses from prior profile a loop vector I whose default schedule brings good performance improvement. The syntactic matrix SM of the schedule which brings LV its best performance improvement is divided into submatrices, each of which contains several successive rows. The S-Search randomly picks one submatrix and modifies it with two basic operations. The first one is *hoisting*, which swap values between two randomly chosen columns (source and target) within the chosen submatrix. The other one is *reordering*, which randomly assigns new values to a randomly selected column in the chosen submatrix. Random decisions are made on how to divide the matrix, which submatrix or matrices to modify and other operation parameters, favouring simple decisions such as dividing the matrix evenly into 2 or 3 submatrices. These steps are repeated until a new matrix *SM'* is found. Details of this syntactic matrix generation algorithm can be found in Long (2004).

SM' is then combined with I and the resulting schedule *S'* will be tested. This process is repeated for *Sbudget* times in each round of S-Search.

Subsequently, the S-Search algorithm chooses different loop vectors for each statement in the original loop, and then constructs syntactic vectors for each loop vector separately.

Legality and Duplicity UTF provides a legality test (Kelly and Pugh (1993)) for all generated schedules. The steering module stores all schedules generated during the search process in order to prevent duplicate visits. Matrices and vectors generated during the search process are stored accordingly. They are used to check whether the newly generated matrix and vector have been tried before. If so, they are simply abandoned.

It is worth noting that the heuristic search algorithm reassesses the weights of all the options it uses to make random decisions periodically, i.e. at the end of each L-Search and S-Search round. These weights are given various increments and decrements, based on both the latest profile collected and those collected before. This not only biases the search to the "more promising" directions, but also helps to turn the exploration back to "less promising" directions when the "more promising" directions have been thoroughly expored. This enables the search algorithm to follow the "window search" strategy and tradeoff between efficiency and coverage.

Furthermore, the steering module configures the search before it starts. It sets as constants the values of Budget, Lbudget, Sbudget, range of tile size and unrolling factors to consider, and all remaining default options.

A comprehensive description of the algorithm can be found in Long (2004).

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 3: Summary of Benchmarks Used

3.4 Experimental Results

To the best of our knowledge, no Java compiler currently available provides the UTF transformations considered in this paper. There is no published work (except Long and O'Boyle (2001)) about the potential of these transformations on Java optimisation. Therefore, no direct comparison can be made between the heuristic search algorithm and the others. Instead, we give absolute performance improvement and evaluate how quickly good points are found.

In order to evaluate the search algorithm, we develop an Adaptive Optimisation Framework for Java (AOF-Java), which is a source-to-source Java restructurer using iterative optimisation. It interprets the UTF schedules into transformation sequences and applies them to the target program. The program will then be executed, with execution time recorded.

The experiments were conducted within two environments, one is Java 2 Runtime Environment with Java Hotspot Client VM (1.3.0) running on RedHat Linux 6.3 in Intel Celeron (533MHz) with 128M RAM. The other is Java 2 Runtime Environment with Java Hotspot Client VM (1.4.1.1_01) running on MS-Windows 2000 in PentiumPro (200MHz) with 96M RAM.

Sixteen code segments were chosen from two widely-used benchmark suites, namely *Java Grande Forum Benchmark Suite* (JGF) (Bull et al. (2000)) and *Livermore* (Livermore benchmark). They are summarised in Figure 3. For each benchmark, the algorithm evaluated the first 100 (*Budget* in Figure 2) legal points it reached in the corresponding optimisation space. This search process takes about 20 to 50 minutes, depending on the benchmarks.

As the heuristic search algorithm may explore the polyhedral space via different directions in different search runs, this experiment is repeated 10 times in order to ensure the results are not achieved by coincidence. In ad-

Code	Speedup		
	after 100	after 20	Percentage
kernel3	1.09	1.05	56%
kernel5	1.14	1.11	79%
kernel6	1.17	1.13	76%
kernel7	1.06	1.06	99%
kernel8	1.29	1.29	99%
kernel9	1.21	1.09	43%
kernel10	1.13	1.13	92%
kernel11	1.45	1.40	89%
kernel12	1.08	1.06	75%
kernel19	1.07	1.06	86%
runF	1.07	1.06	86%
runG	1.09	1.09	99%
runR	1.09	1.07	78%
runS	1.02	1.00	18%
mm	1.21	1.20	95%
doIteration	1.06	1.04	67%
Average	1.14	1.12	78%

Figure 4: Summary of results in Linux+Celeron

dition, it minimises the impact of noise caused by factors such as the virtual machine.

3.4.1 Linux+Celeron

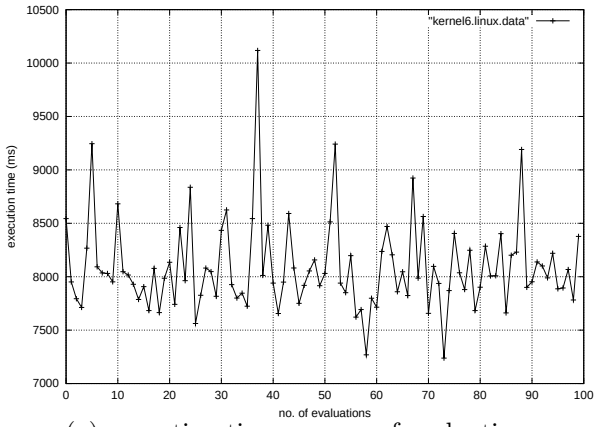
Figure 4 demonstrates that the heuristic search algorithm improves the performance of all of these benchmarks in Linux+Celeron. It achieves an average speedup of 1.14, and this achievement can be obtained quickly.

The best improvements found within the first 20 and 100 evaluations are presented in the table. They show that, on all benchmarks except *kernel3*, *kernel9*, *doIteration* and *runS*, the algorithm needs only about 20 evaluations to achieve most of the speedup achieved within 100 evaluations. In the case of *kernel9*, it takes more than 60 attempts. The *Percentage* column shows that, on average, 78% of the speedup can be obtained within 20 evaluations.

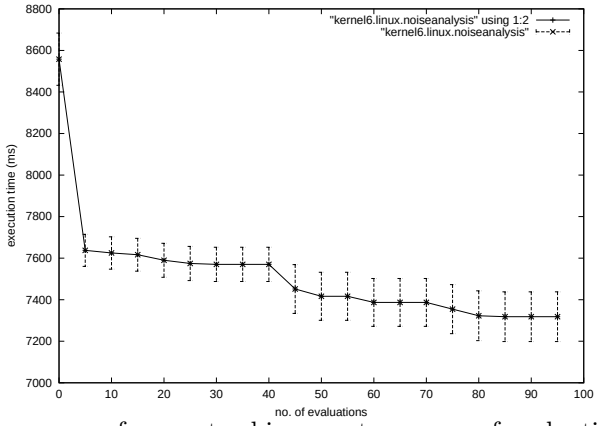
To examine how the algorithm behaves during the search, consider the diagram in Figure 5(a) which shows the execution time of *kernel6* against the number of evaluations during one iterative search on *kernel6*. It shows large variation in performance caused by different transformations, which demonstrates the complexity of the optimisation space considered in this paper.

The best execution time found so far in each search run are obtained for all 10 runs. The average of these achievements are plotted against the number of evaluations in Figure 5(b). This demonstrates that although the optimisation space is complex, the heuristic search algorithm can find good points in it quickly. In addition, the standard deviations of these achievements are low, as the error bars in Figure 5(b) indicate. This shows that although these 10 runs explore the space in different direction, they achieve similar results on *kernel6*.

The search results show that most of the legal points



(a) execution time vs. no. of evaluations



(b) average of current achievements vs. no. of evaluations

Figure 5: Heuristic search in Linux+Celeron

reached by the search algorithm use short and simple transformations, for instance, tiling only. To a certain extent, this justifies the "simple first" strategy of the algorithm. On the other hand, this is partly due to the fact that the relatively simple nature of these benchmarks restrains the applicability of more complex transformation combinations. The heuristic search algorithm biases its search toward simple transformations accordingly, which demonstrates its adaptability to the program it is to optimise.

3.4.2 Windows+PentiumPro

The experimental results in Windows+PentiumPro are summarised in Figure 6. They demonstrate that the heuristic search algorithm can also bring performance improvement to many of these benchmarks in this environment. It achieves an average speedup of 1.10 and 89% of the speedup can be obtained within 20 evaluations.

The search algorithm finds, within the first 20 evaluations, most of the speedup achieved within 100 evaluations for *kernel7*, *kernel8*, *kernel9*, *kernel11*, *kernel12*, *kernel19*, *runF*, *runR* and *mm*. The negligible standard deviations (shown the error bars in Figure 7(b)) show that all 10

Code	Speedup		
	after 100	after 20	Percentage
kernel3	1.18	1.14	78%
kernel5	1.10	1.07	70%
kernel6	1.09	1.07	78%
kernel7	1.05	1.05	99%
kernel8	1.14	1.14	99%
kernel9	1.37	1.36	97%
kernel10	1.06	1.05	83%
kernel11	1.18	1.17	94%
kernel12	1.19	1.19	99%
kernel19	1.01	1.01	99%
runF	1.02	1.02	99%
runG	1.01	1.00	81%
runR	1.09	1.09	99%
runS	1.01	1.00	75%
mm	1.14	1.13	93%
doIteration	1.05	1.04	80%
Average	1.10	1.09	89%

Figure 6: Summary of results in Windows+PentiumPro

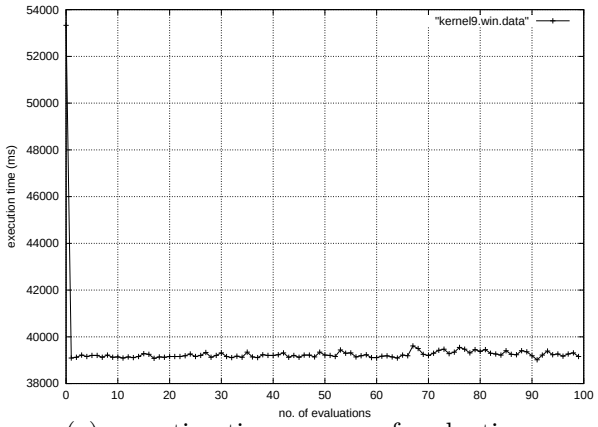
searches on *kernel9* achieve similar performance improvements. The search algorithm achieves very similar results on other benchmarks.

The search results show that some programs are less sensitive to the transformations in Windows+PentiumPro. Figure 7(a) demonstrates one search on *kernel9*. The curve shows the execution time against the number of evaluations during one search on *kernel9*. This indicates that, regardless of the transformations applied, its performance is almost invariant. To find out whether this is just a coincidence, we derive the best execution time found so far during each of the 10 search runs, and plot the average of these achievements against the number of evaluations. The result is shown in Figure 7(b). The error bars show that the standard deviations of these 10 search runs are low, i.e. they all achieve similar results on *kernel9*. Similar characteristic is also found on *kernel11* and *kernel12*, whilst the others' curves are still highly irregular and sensitive to transformations applied, like in Figure 5(a).

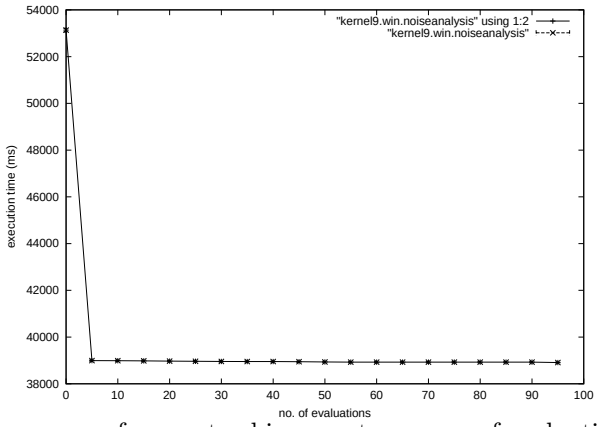
It is worth noting that the achievement in this environment is less significant than that in the Linux+Celeron environment. This may be due to the fact that the small L2 cache of Celeron makes the relative cost of memory latency on Linux+Celeron greater, and therefore it benefits more from cache restructuring-based transformations.

3.4.3 Summary

The above results demonstrate that the algorithm is capable of achieving Java performance improvement in both environments. On the 16 benchmarks, it achieves an average speedup of 1.14 in Linux+Celeron and 1.10 in Windows+PentiumPro. In addition, the algorithm achieves this improvement quickly. Within 20 evaluations, 78% of its achievement in Linux+Celeron can be obtained, and



(a) execution time vs. no. of evaluations



(b) average of current achievements vs. no. of evaluations

Figure 7: Heuristic search in Windows+PentiumPro

89% of that can be obtained in Windows+PentiumPro. On average, it takes less than 5 seconds to find a legal point during the search process, and the vast majority of search time is actually spent on evaluation of these points. This justifies the strategies used by this heuristic search algorithm.

4 Related Work

Bastoul et al. (2003) and Cohen et al. (2004) specify an optimisation space in a manner similar to UTF. It considers a static control part (SCoP) as a maximum set of consecutive statements without *while* loops, and a program transformation on this SCoP may cause modification in its iteration domain, or its iteration schedule, or its memory access function. A polyhedron is used to represent this modification, and a set of primitives are used to modify the polyhedron. This results in a larger optimisation space than UTF can represent. However, no approach is given to explore it for performance improvements.

There have been some work in iterative optimisation. Kisuki et al. (1999) optimise a few kernels by repeatedly

execute different versions of them and using the feedback to decide further optimisation. Fursin et al. (2002) and Fursin (2004) extend the iterative search for good points within optimisation space to larger applications. It introduces several search strategies to speedup the exploration. However, only a few parameterised transformations (tiling, unrolling and array padding) are considered in various fixed phased orders. Therefore, the optimisation spaces are quite small and regular-shaped, and the search algorithm is difficult to extend and perform long sequences of composed transformations.

Franke et al. (2005) propose a probabilistic feedback-driven search algorithm in search for good transformation sequence in a large optimisation space consisting of 81 source-level transformations. It combines two search approaches, namely random search and localised search within a good area, and chooses the best of them at the final merge stage. This is very similar to the random and window search strategies our heuristic search algorithm uses to tradeoff efficiency and coverage.

Kelly and Pugh (1993a) and Nisbet (2001) consider search in a large UTF-based space. The algorithm of Kelly and Pugh (1993a) constructs the mapping for each statement in a level by level manner. At each level, an estimate is made on the partly specified mapping, which is then augmented if and only if the estimate is good. The main drawback of this algorithm is that no runtime feedback can be used to bias the mapping construction, as no code can be generated from a mapping only partly constructed. Nisbet (2001) uses genetic algorithm to optimise programs for parallel architectures. However, the efficiency of genetic algorithm is poor (it takes several hours to find a good transformation within the space).

Cooper et al. (2004) use a biased random search algorithm to explore a large space consisting of a pool of dataflow transformations. Its efficiency remains unknown. The phase order problem it aims to solve is relatively simple compared to what this and the above papers consider. OSE (Triantafyllis et al. (2003)) considers an optimisation space composed of various compilations and configurations. It uses compiler writer's prior experience to prune many points before a breadth first tree search starts. Experimental results show that OSE can yield significant performance improvement. Pinkers et al. (2004) present an approach to turn on or off compiler options in order to find the optimal set of them. It uses orthogonal arrays in statistical profile analysis to calculate the main efforts of these options.

Java optimisations (Shirazi (2002)) are achieved via an efficient virtual machine (Alpern et al. (1999)), or optimisation techniques such as JIT compilation (Adl-Tabatabai et al. (1998)) and parallelisation (Bik and Gannon (1997)). The virtual machine approach is inevitably architecture-specific, JIT compilation considers only light-weighted optimisations, whilst parallelisation relies on architecture support. Moreira et al. (1998) provide a package supporting true multi-dimensional arrays needed in high performance computing. But it is not sufficiently flexible to allow creation of arrays of arbitrary classes and of arbitrary dimension.

5 Discussion

Although experimental results show that the proposed heuristic search algorithm can quickly find good transformations in the large UTF-based optimisation space, the iterative search cost may still be unacceptable in some circumstances. Furthermore, because UTF focuses on loop reordering transformations, this search algorithm optimises codes at loop level. It is expensive to apply it to a large application containing many loops, as each loop has to be optimised respectively. Therefore, we profile the application, find its runtime hotspots and then use the search algorithm to optimise those loops within the hotspots.

We consider to speed up the iterative search by using dynamic versioning techniques (Fursin et al. (2005)) and by applying machine learning techiques (Mitchell (1997)) to further narrow down the optimization space. We are enhancing AOF-Java with an instance-based learning optimisation approach (Long and O'Boyle (2004)), and train it with the good points found by the above heuristic search algorithm. Preliminary experimental results show that after training, it can, within just one attempt, achieve on average over 70% of the best performance improvement found by the search algorithm after 100 evaluations. Therefore, our search algorithm can be useful for the selection of training examples in large optimization spaces.

6 Conclusion and future work

This paper uses UTF to specify a large and complex optimisation space of iteration reordering transformations. It presents a heuristic random search algorithm independent of architecture, language and environment, as no such information is hardwired in the algorithm. The experimental results show that this algorithm is capable of locating good points within this space quickly. This demonstrates that, by better exploring the potential of high-order transformations, it is possible to make Java a more realistic option for portable high performance computing. Furthermore, we suggest using this heuristic search algorithm to select training examples for a machine learning-based optimisation approach.

Future work will further explore the potential of machine learning techniques for program optimisation, consider languages other than Java, and investigate optimisations outside the UTF framework.

REFERENCES

Adl-Tabatabai, A., Cierniak, M., Lueh, G., Parakh, V.M. and Stichnoth, J.M. (1998) 'Fast, effective code genera-

tion in a just-in-time Java compiler', *Proc. of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

Alpern, B., Cocchi, A., Lieber, D., Mergen, M. and Sarkar, V. (1999) 'Jalapeno - a compiler-supported Java virtual machine for servers', *Proc. of Workshop on Compiler Support for Software System (WCSSS).*

Arnold, M., Fink, S., Grove, D., Hind, M. and Sweeney, P. (2003) *A survey of adaptive optimisation in virtual machines*, IBM research report RC23143 (W0312-097).

Bastoul, C., Cohen, A., Girbal, S., Sharma, S. and Temam, O. (2003) 'Putting polyhedral loop transformations to work', *Proc. of Workshop on Languages and Compilers for Parallel Computing (LCPC).*

Bik, A. and Gannon, D. (1997) 'Javar: a prototype Java reconstructing compiler', *Concurrency, Practice and Experience*, 9(11).

Bodin, F., Kisuki, T., Knijnenburg, P., O'Boyle, M. and Rohou, E. (1998) 'Iterative compilation in a non-linear optimisation space', *Proc. ACM Workshop on Profile and Feedback Directed Compilation*, Organized in conjunction with PACT'98.

Bull, M., Smith, L., Westhead, M., Henty, D. and Davey, R. (2000) 'A benchmark suite for high performance Java', *Concurrency, Practice and Experience*, Vol.12.

Cohen, A., Girbal, S. and Temam, O. (2004) 'A polyhedral approach to ease the composition of program transformations', *Proc. of Europar International Conference on Parallel and distributed Computing (Europar).*

Cooper, K., Grosul, A., Harvey, T., Reeves, S., Subramanian, D., Torzon, L. and Waterman, T. (2004) 'Exploring the structure of the space of compilation sequences using randomized search algorithms', *Proc of the 2004 Los Alamos Computer Science Institute (LACSI) Symposium.*

Franke, B., O'Boyle, M., Thomson, J. and Fursin, G. (2005) 'Probabilistic source-level optimisation of embedded programs', *Proc. of the 2005 Conference on Languages, Compilers and Tools for Embedded Systems (LCTES).*

Fursin, G. (2004) *Iterative Compilation and Performance Prediction for Numerical Applications*, PhD thesis, School of Informatics, The University of Edinburgh.

Fursin, G., Cohen, A., O'Boyle, M. and Temam, O. (2005) 'A Practical Method For Quickly Evaluating Program Optimizations', *Proc. of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, LNCS-3793, Springer Verlag.

Fursin, G., O'Boyle, M. and Knijnenburg, P. (2002) 'Evaluating iterative compilation', *Proc. of 15th Workshop on Languages and Compilers for Parallel Computers (LCPC).*

'Java Grande Forum report: making Java work for high-end computing', *Java Grande Forum panel, SC98: High Performance Networking and Computing.*

Kelly, W. and Pugh, W. (1993) *A framework for unifying reordering transformations*, Technical report of Univ. of Maryland, CS-TR-3193.

Kelly, W. and Pugh, W. (1993a) *Determining schedules based on performance estimation*, Technical report of Univ. of Maryland, CS-TR-3108.

Kennedy, K. and Allen, R. (2002) *Optimizing compilers for modern architecture, a dependence-based approach*, Morgan Kaufmann.

Kisuki, T., Knijnenburg, P., O'Boyle, M., Bodin, F. and Wijshoff, H. (1999) 'A feasibility study in iterative compilation', *Proc. of International Symposium of High Performance Computing (ISHPC)*, Lecture Notes in Computer Science, vol.1615.

Livermore benchmark URL: http://www.netlib.org/benchmark/livermore.

Long, S. and O'Boyle, M. (2001) 'Towards an adaptive Java optimising compiler: an empirical evaluation of program transformations', *Proc. of the 3rd Workshop on Java for High Performance Computing.*

Long, S. and O'Boyle, M. (2004) 'Adaptive Java optimisation using instance-based learning', *Proc. of the 18th ACM International Conference for Supercomputing (ICS).*

Long, S. (2004) *Adaptive Java optimisation using machine learning techniques*, PhD thesis, School of Informatics, The University of Edinburgh.

Mitchell, T. (1997) *Machine learning*, McGraw-Hill.

Moreira, J., Midkiff, S. and Gupta, M. (1998) 'From flops to megaflops: Java for techincal computing', *Proc of the 11st International Workshop on Languages and Compilers for Parallel Computing (LCPC).*

Nisbet, A. (2001) 'Towards retargettable compilers - feedback directed compilation using genetic algorithm', *Proc. of the 9th International Workshop on Compilers for Parallel Computers (CPC).*

Parello, D., Temam, O. and Verdun, J. (2002) 'On increasing architecture awareness in program optimizations to bridge the gap between peach and sustained processor performance? matrix-multiply revisited', *Proc. of SuperComputing.*

Pinkers, R., Knijnenburg, P., Haneda, M. and Wijshoff, H. (2004) 'Analysis of Compiler Options using Orthogonal Arrays', *Proc. of the 11st International Workshop on Compilers for Parallel Computers (CPC)*.

Shirazi, J. (2002) *Java performance tuning (2nd edition)*, O'Reilly.

Triantafyllis, S., Vachharajani, M., Vachharajani, N. and August, D. (2003) 'Compiler optimisation-space exploration', *Proc. of the 2003 International Symposium on Code Generation and Optimisation (CGO)*.