A heuristic search algorithm based on Unified Transformation Framework

Shun Long and Grigori Fursin
Institute for Computing Systems Architecture, The University of Edinburgh, United Kingdom
Email: slong@inf.ed.ac.uk, grigori.fursin@ed.ac.uk

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)[10]. 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.

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 usually applies various transformations at various levels. Previous work[18] demonstrates that complex transformation compositions can bring significant performance improvement. However, traditional optimising compilers are based on static analysis and a hardwired compilation strategy. 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[12] is therefore introduced to explore such a space. However, many current iterative optimisation approaches[8][12] target just a small set of transformations. They can neither be easily extended nor perform long sequences of composed transformations.

A unified representation[3] 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)[10]. It presents a unified and systematic representation of iteration reordering transformations and their arbitrary combinations, which aims 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[1][2][4][16] have been proposed to improve Java's runtime performance. In [14], 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 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. This demonstrates the effectiveness of this algorithm.

The outline of this paper is as follows. Section 2 specifies the optimisation space with the help of the UTF. The heuristic search algorithm is presented in section 3, before the experimental results is presented in section 4. Section 5 discusses related works. It is followed by some concluding remarks in section 6.

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 con-

sider 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[8][12] has focused on search strategies based on highly restricted optimisation spaces.

The Unified Transformation Framework (UTF)[10] 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 are shown in A). If loop interchange, distribution and skewing are applied in turn, the resulting schedules and codes shown in B), C) and D) of Figure 1 are obtained.

It is worth noting that no UTF transformation, 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. [15] 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 where

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) Step 1: 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) Step 2: 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) Step 3: 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 and its mappings before and after some iteration reordering transformations. 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 0 is then applied, the *i* in T_1 in C) is changed to *i+j*, as shown in D).

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

Such an optimisation space is not only large but also complex. [12] demonstrates 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 resource, 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, it 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*×*M* where *I* is an (*x*+2)-dimensional vector of the iteration variables and *M* an (*x*+2)×(*x*+1) matrix of integer constants. For example, the mappings of D) in Figure 1 can be represented as follows.

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

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

Given a loop vector *LV*, its *default schedule* transforms the code block where 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:

Random Where 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 shown 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.

Simple first Although [18] claims 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, the majority of performance improvement comes from either one transformation or a combination of only a few[14]. 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 [18] indicates. This search will

continue and any arbitrary complex schedule UTF can represent will therefore be considered, as long as budget allowed.

Intuitively, this "simple first" strategy means 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 of no tiling or just tiling once before considering those of multiply tiling.

Window search The search algorithm should be flexible 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 [12]. 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

Using the above notations and search strategies, our heuristic search algorithm is a two phase process in which the subspace of loop vectors and syntactic vectors are explored by L-Search and S-Search respectively, as explained below. As mentioned earlier, the loop vector is associated with a larger set of transformations than that of syntactic vector. This means that arbitrary combinations of these loop vector transformations indicate more varieties than those associated with the syntactic vector subspace. Therefore, the algorithm attempts to decide the loop vector first before considering the variety of syntactic vectors, i.e. it prefers L-Search to S-Search.

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.

L-Search The initial L-Search generates a certain number 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*, which are in turn decided by tile size(s) and unrolling factor(s) randomly chosen from a suitable range, if tiling and/or unrolling is included. These decisions are made randomly and with a bias to simple decisions, as explained above.

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, whose transpose matrix is M^T=(m a), 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 [15]. 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.

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 Initially, S-Search 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, modifies it with two basic operations (swapping values between two randomly chosen columns, and randomly assigning new values to a randomly selected column). 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 [15]. *SM'* is then combined with *I* and the resulting schedule *S'* will be tested.

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

Legality and Duplicity UTF provides a legality test[10] 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 processes are stored accordingly. They are used to check whether matrix and vector newly generated have been tried before. If so, they are simply abandoned. Furthermore, the steering module configures the search before it starts. It sets as constants range of tile size and unrolling factors to consider, and other default options.

A comprehensive description of the algorithm can be found in [15].

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 [14]) 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 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 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)[5] and *Livermore*[13]. For each benchmark, the algorithm evaluated the first 100 points it reached in the corresponding optimisation space. This search process takes about 20 to 50 minutes, depending on the benchmarks.

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 2. Summary of the experimental results on Linux+Celeron

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

4.1 Linux+Celeron

Figure 2 demonstrates that the heuristic search algorithm improves the performance of all of these benchmarks on 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 3(a) which shows the execution time of *kernel6* against the number of evaluations during one iterative search. It shows large variation in performance caused by different transformations, which demonstrates the complexity of the optimisation space considered in this paper.

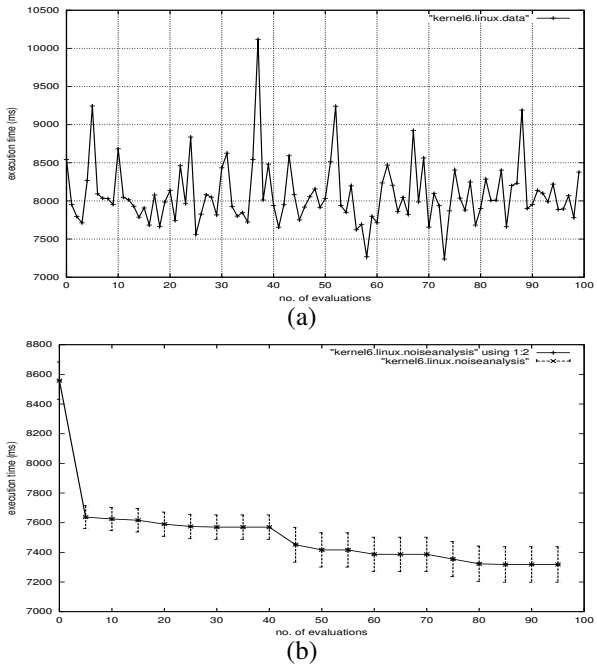

Figure 3. Heuristic search on Linux+Celeron (a) the curve shows the execution time against the number of evaluations during one search on kernel6. It demonstrates the complexity of the optimisation space. (b) the average of the best execution time currently found over 10 searches on kernel6 are plotted against the number of evaluations, as shown in the solid curve. The error bars show that the standard deviations are low, which indicates that all 10 searches achieve very similar results.

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 Figure3(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 3(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 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

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 4. Summary of the experimental results on Windows+PentiumPro

heuristic search algorithm biases its search toward simple transformations accordingly, which demonstrates its adaptability to the program it is to optimise.

4.2 Windows+PentiumPro

The experimental results on Windows+PentiumPro are summarised in Figure 4. 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 5(b)) show that all 10 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 on Windows+PentiumPro. Figure 5(a) demonstrates one search on *kernel9*. It shows 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

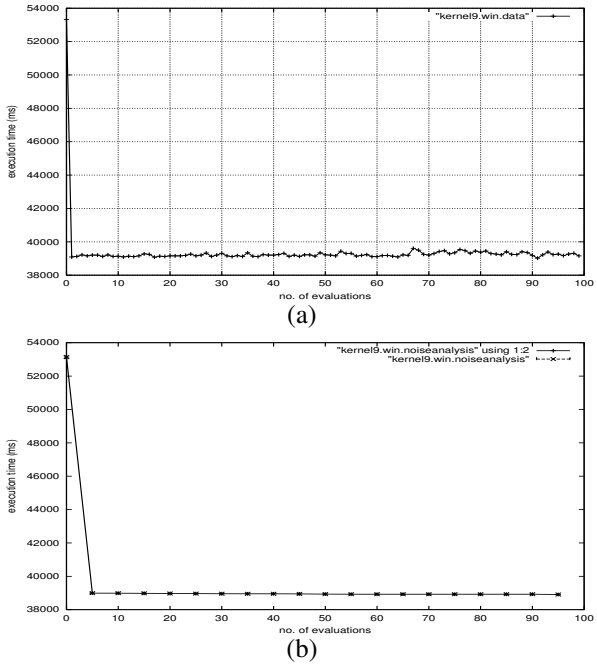

Figure 5. Heuristic search on Windows+PentiumPro (a) the curve shows the execution time against the number of evaluations during one search on kernel9; (b) the average of the best execution time currently found over 10 searches on kernel9 are plotted against the number of evaluations, as shown in the solid curve. The error bars show that the standard deviations are low, which indicates that all 10 searches achieve very similar results.

result is shown in Figure 5(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 3(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.

4.3 Summary

The above results demonstrate that the algorithm is capable of achieving Java performance improvement in both environments within a remarkably small number of attempts. 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 search strategies this algorithm uses.

5 Related Work

[3][6] 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 in search for performance improvement.

There have been some work in iterative optimisation[12], which attempts to optimise a program by repeatedly executing different versions of it and using the feedback to decide the next optimisation attempt. They all try to explore an optimisation space in search for good points, as our search algorithm does. [8] and [12] presents two efficient search algorithms. However, the search spaces they consider are small and regular-shaped, containing only a few parameterised transformations in fixed phase order.

[11][17] consider search in a large UTF-based space. The algorithm of [11] 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. [17] 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).

[7] uses a biased random search algorithm to explore a large space consisting of a pool of data-flow 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[20] 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. [19] presents 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 optimisation are achieved via an efficient virtual machine[2], or optimisation techniques such as JIT

compilation[1] and parallelisation[4]. The virtual machine approach is inevitably architecture-specific, JIT compilation considers only light-weighted optimisations, whilst parallelisation relies on architecture support. [16] provides 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.

6 Conclusions

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 exploring the potential of high-order transformations, it helps to make Java a more realistic option for portable high performance computing.

References

[1] A. Adl-Tabatabai et.al, Fast, effective code generation in a just-in-time Java compiler, proc. of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), 1998.

[2] B. Alpern et.al, Jalapeno - a compiler-supported Java virtual machine for servers, proc. of Workshop on Compiler Support for Software System (WCSSS99), 1999.

[3] C. Bastoul et.al, Putting polyhedral loop transformations to work, proc. of Workshop on Languages and Compilers for Parallel Computing (LCPC'03), 2003.

[4] A. Bik et.al, Javar - a prototype Java reconstructing compiler, Concurrency, Practice and Experience 9(11), 1997.

[5] M. Bull et.al, A benchmark suite for high performance Java, Concurrency, Practice and Experience, Vol.12, 2000.

[6] A. Cohen et.al, A polyhedral approach to ease the composition of program transformations, proc. of Europar International Conference on Parallel and distributed Computing (Europar'04), 2004.

[7] K. Cooper et.al, Adaptive optimizing compilers for the 21st century, Journal of Supercomputing, 2001.

[8] G. Fursin et.al, Evaluating iterative compilation, proc. of 15th Workshop on Languages and Compilers for Parallel Computers (LCPC'02), 2002.

[9] Java Grande Forum, Java Grande Forum report: making Java work for high-end computing, Java Grande Forum panel, SC98: High Performance Networking and Computing, 1998.

[10] W. Kelly et.al, A framework for unifying reordering transformations. Technical report of Univ. of Maryland, CS-TR-3193, 1993.

[11] W. Kelly et.al, Determining schedules based on performance estimation, Technical report of Univ. of Maryland, CS-TR-3108, 1993.

[12] T. Kisuki et.al, A feasibility study in iterative compilation, proc. of International Symposium of High Performance Computing (ISHPC'99), Lecture Notes in Computer Science, vol.1615, 1999.

[13] Livermore benchmark, http://www.netlib.org/benchmark/livermore.

[14] S. Long et.al, Towards an adaptive Java optimising compiler, an empirical evaluation of program transformations, proc. of 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] J. Moreira et.al, From flops to megaflops: Java for techincal computing, proc of the 11st International Workshop on Languages and Compilers for Parallel Computing (LCPC'98), 1998.

[17] A. Nisbet, Towards retargettable compilers - feedback directed compilation using genetic algorithm, proc. of the 9th International Workshop on Compilers for Parallel Computers (CPC2001), 2001.

[18] D. Parello et.al, On increasing architecture awareness in program optimisations to bridge the gap between peach and sustained processor performance? matrix-multiply revisited. proc. of SuperComputing'02. 2002.

[19] R. Pinkers et,al, Analysis of Compiler Options using Orthogonal Arrays, proc. of the 11st International Workshop on Compilers for Parallel Computers (CPC2004), 2004.

[20] S. Triantafyllis et.al, Compiler optimisation-space exploration, proc. of the 2003 International Symposium on Code Generation and Optimisation (CGO'03), 2003.