Towards an Adaptive Java Optimising Compiler: An Empirical Evaluation of Program Transformations

Shun Long and Michael O'Boyle
Division of Informatics, University of Edinburgh
Kings Buildings, Edinburgh EH9 3JZ, UK
{shun,mob}@dcs.ed.ac.uk

Abstract

Our overall aim is to develop an Adaptive Java Optimising Framework(AJOF) which dynamically optimises Java programs during execution. This paper focuses on one aspect of such a framework and investigates the impact of high level transformations on program performance. It shows that by combining transformations, it is possible to achieve speedups on a uni-processor of upto 1.86. This is followed by an evaluation of an adaptive feedback directed search strategy where it is is shown that it is possible to achieve 95% of the available performance with just 15 evaluations. Thus good transformation may be found relatively quickly and can be included in a on-line adaptive compilation system.

1 Introduction

The Java programming language has the desirable property of allowing programmers to write high level platform independent code. Compiling Java to bytecode once ensures that it may be run on any platform in the future which supports a suitable Java Virtual Machine (JVM). However, such portability without further consideration comes at a price, long execution time due to JVM interpretation. This has lead to a number of developments including improved virtual machines and Just In Time (JIT) compilation [5].

We are interested in developing a compiler based optimisation framework applicable to Java programs. In particular we are interested in going beyond current approaches to JIT compilation in adapting the executing code based on runtime behaviour. Not only are we interested in modifying the program at the binary level [8], we also wish to consider higher levels, namely bytecode and source code level.

This paper focuses on one aspect of such a framework, namely the impact of source to source level program transformations on program execution time and the feasibility of incorporating such high level program transformations into an on-line adaptive framework. A number of transformations are selected and the ensuing transformation space exhaustively explored in order to quantify the potential benefit. We show that such transformations can improve uni-processor performance by up to 86%. This is followed by an evaluation of a feedback directed optimisation technique [4] where initial results show that 95% of the speedup available can be obtained in 15 executions. Thus good transformation may be found relatively quickly and can be included in a on-line adaptive compilation system.

The outline of this extended abstract is as follows. Section 2 describes the motivation of developing an adaptive compiler and compares our approach to related work. This is followed in section 3 by a a brief description of the overall compiler infrastructure. Section 4 provides the main contribution of the paper. Here we evaluate the impact of program transformations on selected Java programs by exhaustively investigating certain optimisation spaces. Section 5 evaluates the performance of an iterative based compilation approach which is followed by some concluding remarks.

2 Motivation + Brief Summary of Related Work

Our overall goal is to develop an optimising compiler framework for Java that delivers performance at least equal to that of standard compiled imperative languages. Furthermore, we wish to use dynamic feedback information to drive further levels of optimisation unavailable to approaches based on static analysis. We would also like such a system to be flexible, adapting to different underlying platforms without having to rewrite the compiler's optimisation strategy. Thus we are interested in a developing a framework which *both adapts to program and architectural change*. Section 3 gives an outline of our proposed architecture, but the general idea is that as a program executes, the framework endeavours to find an improved version of the program that will run faster on that particular machine.

The ADAPT project [11], an adaptive compilation system, is closest in spirit to our work. It has a distributed architecture very similar to ours. The key is the de-coupling of the runtime code generation from the dynamic selection of generated variants at their points of use. This allows code generation to occur concurrently with program execution, and thus removes dynamic compilation overheads from the critical path. It differs primarily in that it is aimed at Fortran applications but more significantly its optimisation strategy is very ad hoc - try a selection of high level optimisations and see which works. Instead we use iterative compilation techniques [4, 9] developed in the context of off-line optimisation as a means of rapidly searching the program transformation space to determine good optimisations in a platform adaptive manner.

The use of high level transformations such as loop transformations and parallelisation in the context of Java has been extensively investigated in the Ninja [2] project and in [3]. In [2] analysis and a transformation technique, that enables high level restructuring to be legally applied, are described and it is shown that static Java compilation can approach the performance of Fortran without sacrificing behaviour in the presence of exceptions. Our work is similar in that we also use high level transformations as an optimisation approach. We differ in that we search an optimisation space based on runtime feedback.

The Quicksilver project [10] based upon the Jalapeno [5] project has considerable overlap with our approach. They too wish to exploit the performance of compiled binaries over interpreted byte-code in an adaptive compiler [1]. In [10] they focus on developing a system that allows the use of pre-compiled code within a dynamic compiler. This allows the maintenance of Java's strict semantics but allows the use of pre-compiled optimised code where legal. Thus, their framework is similar to ours but they focus on the integration of pre-compiled code within an adaptive compiler while this paper investigates the opportunities for using high level transformations within such a system.

3 Adaptive Java Optimisation Framework(AJOF)

3.1 System Architecture

The Adaptive Java Optimisation Framework (AJOF) shown in figure 1, aims to provide an environment that promises continuous performance enhancement to Java applications, especially those that run persistently.

It reads Java source as input and outputs either bytecode or native code. The Analyser optionally performs analysis, applies certain static optimisations, and is responsible for inserting Profiler Stubs into potential hotspots in the Java code. The Transformer is responsible for applying dynamic optimisations, based on profile information from the Profiler and is the subject of evaluation in section 4. Only high-order transformations are currently implemented in AJOF and thus we rely on existing bytecode and/or native code generator(s). Other components include the Profiler Stub which collects information with minimal system interference and the Dispatcher which is responsible for dispatching the output of the compiler whether it be bytecode or native code.

As we are developing a feedback orientated system profile gathering and storage are central. Finally, the Simulator is an optional module of AJOF aimed at providing an alternative way to profile Java applications in a more detailed manner.

3.2 Current Status

AJOF is an ongoing project and here we summarise the current status. A prototype AJOF Optimising Compiler exists which includes the Analyser and Transformer using *javac* as the bytecode generator. Currently we do not

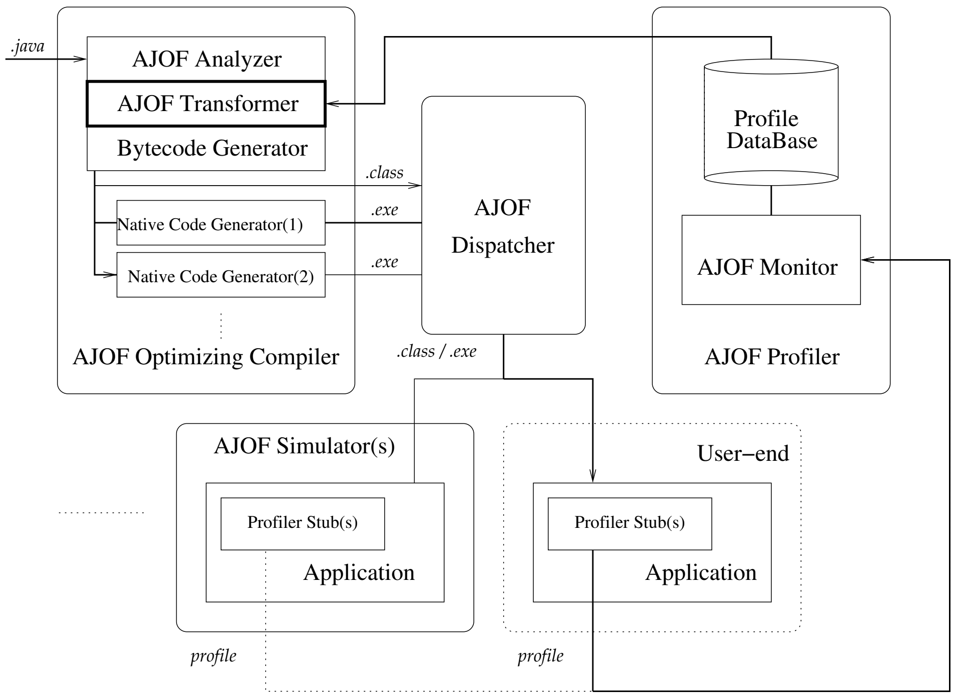

Figure 1. Architecture of AJOF System

use a native code generator. and the Dispatcher and Simulator are still under development.

The AJOF Profiler is still in a state of development and currently provides semi-automatic, off-line profile information. This will soon move to automatic on-line profiling. In detail:

- a simple Profiler Stub has been implemented based upon [6]. It uses a timer to evaluate the execution time and gives the results in milliseconds. A more comprehensive mechanism to support multi-threading will be adopted.
- all the Profiler Stubs are manually inserted into the input source before the latter is read by the AJOF Analyser. Currently all For-loops are considered to be potential hot spots. Automatic insertion will be available shortly
- all the profiles are collected and written to a data file with a *.profile* suffix, which is then be currently evaluated by an off-line profile analyser. This will shortly become an on-line analyser.

This paper focuses on the use of high level transformations (AJOF Transformer) on program performance. Such transformations are platform independent, thus naturally allowing architectural adaptivity. In the next section we evaluate the impact of such transformations on program performance. This is followed by the evaluation of a feedback directed optimiser that uses information from the AJOF profiler (simply execution time in this case) to determine the best high level transformation.

4 Transformation Evaluation

In this paper we are interested in the impact of high level source transformations on Java performance. We initially focus on numerical array based programs and consider suitable loop and array based optimisations. We assume that previous phases have determined those regions where program transformations may safely be applied (see [2]). We focus on uni-processor optimisation and initially examine two well known optimisations tiling and

Program	Tile	Unroll	Tile+Unroll	Scalar	Hoisting	Linear	Dist.	ALL
A	1.00	1.22	1.19	1.28	-	1.22	0.69	1.47
B	1.41	1.00	1.41	1.33	-	1.16	1.08	1.85
C	1.40	1.23	1.76	-	1.41	0.92	-	1.86
D	1.00	1.05	1.05	1.21	-	1.01	-	1.31

Figure 2. Speedup achieved by applying different transformations.

unrolling which are known to be beneficial to uni-processor performance for imperative programs [9]. As shown in [4] the impact of such transformation on performance is highly non-linear and thus we exhaustively explore a restricted space to see the behaviour in the case of Java [1].

Due to the overhead of Java arrays we consider two specific techniques. The first replaces multiple references by accesses to a pre-loaded scalar and exploits simple invariant access by hoisting. The second applies linearisation to true rectangular multi-dimensional arrays based on analysis described in [7]. Loop Distribution can often enable other optimisations and thus its impact is first considered in isolation and then in combination with all other transformations. This is followed by an evaluation of an iterative technique to determine good optimisations based on feedback information and an evaluation of them in an adaptive on-line context.

4.1 Setup and Benchmarks

The experiments were conducted with Java version 1.2.2 Classic VM (built Linux_JDK_1.2.2_FCS, native threads, sunwjit) installed on Linux RedHat 6.2 (kernel 2.2.16). The hardware platform used was a single Intel Celeron (533MHz) processor with 128M RAM.

The transformations were applied to 4 code segments from Java Grande Forum Benchmark Suite [6], http://www.epcc.ed.ac.uk/javagrande/:

Program A: doIteration(), which contains a fairly long loop body;

Program B: CalculationR() which contains a two-dimensional nested loop (300x300);

Program C: a 300x300 matrix multiplication

Program D: iterative routine from Euler benchmark

As JIT execution improves the execution time over interpretation, all results below are with JIT compilation enabled.

4.2 Results

Figure 2 summarises the impact of program transformations on each of the programs. Each entry gives the speedup found when applying a particular transformation to a particular program. The sections below provide a more detailed evaluation.

Tiling Loop tiling was applied to each of the nested loop within each of the code segments. All tile sizes from 1 to 100 were applied and the resulting execution time recorded. Figure 3 shows how performance varies as a function of tile size for the 4 programs and as can be seen from figure 2, the performance improvement ranged from no improvement at all (speedup 1.00) up to a 41% improvement. In the case of A and D the best tile size was 1 i.e. no tiling and hence a speedup of 1. In the case of B and C we see much greater improvements of 41 % and 40 % respectively due to the relatively large data sizes accessed in both programs. The best tile size for B was 14 while in the case of C it was 12.

[1]If none of the transformations evaluated under an exhaustive search provide significant performance improvement, there is little reason to develop an optimisation algorithm to find them!

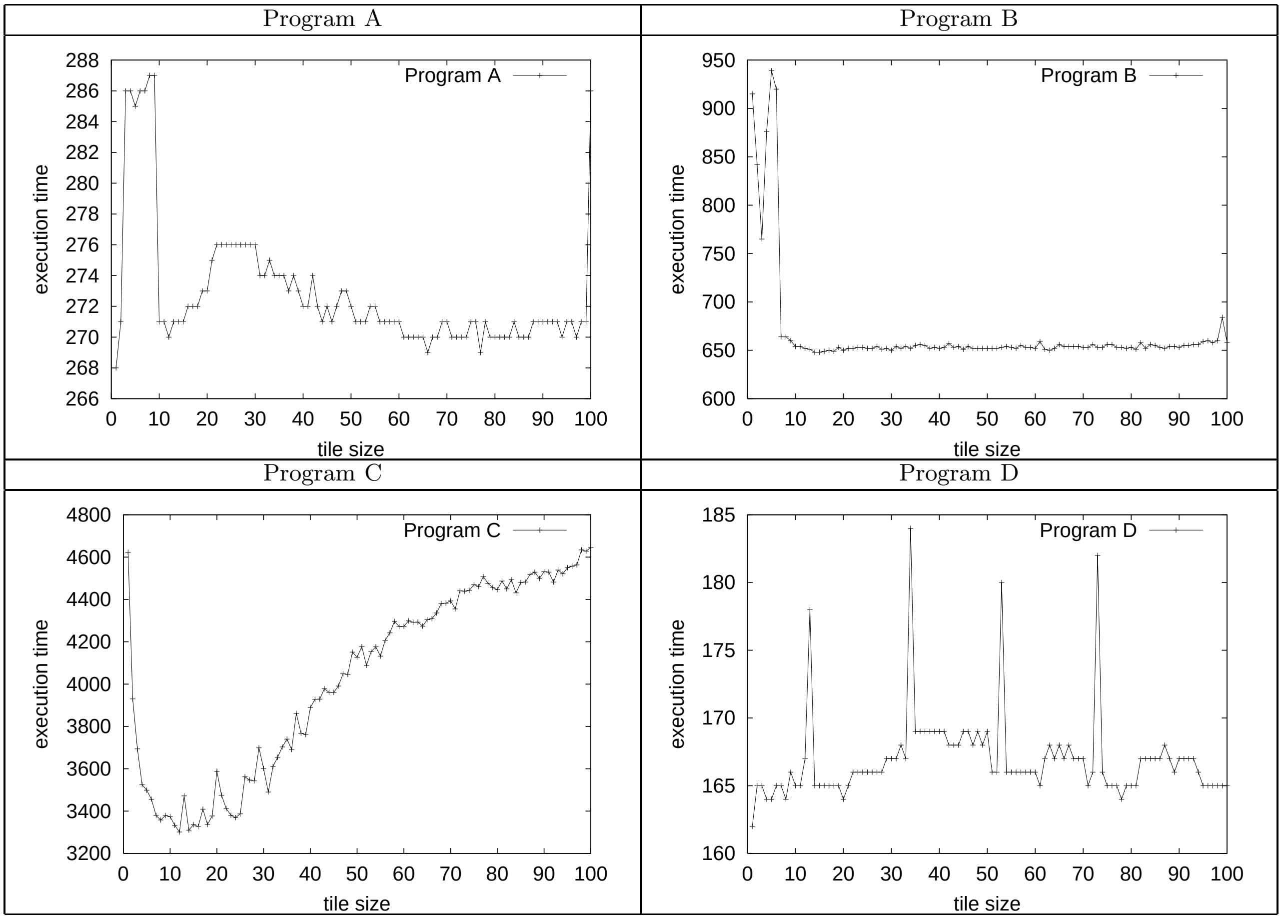

Figure 3. Execution time vs tiling

Unrolling Loop unrolling was applied to each of the innermost loops of the 4 programs and the resulting execution times recorded [2] and plotted against unroll factor (see figure 4). Again we see two programs where the transformation has relatively little impact but this time they are B and D. Due to the relatively large loop body of B, unrolling does not improve performance and hence the best result is for not unrolling. In the case of A,C and D, the best performance occurs with an unroll factor of 2,19 and 4, respectively with performance improvements reaching up to 23%.

Tiling and Unrolling Given the individual impact of unrolling and tiling, we investigated the combined impact of both transformation on program performance evaluating all 2000 different versions of each program[2]. In three cases, A,B and D the combined transformation gives no improvement over just unrolling in A and D's cases and just tiling in B's case. This is not surprising as each of the other transformations for these programs had no impact. In the case of C, which benefited from both tiling and unrolling individually, the combined application gave increased performance improvement. Interestingly the best tile and unroll factors of 16 and 7 were different from the ideal values if determined independently (12 and 19). This shows the typical non-linear coupled behaviour of combined transformations.

Scalar + Hoisting This is a simple transformation where multiple reference to an array element are replaced by first loading the element into a scalar and then referring to the scalar when subsequent accesses are made. For this technique to be applicable there must be a least two array references to the same array element and hence

[2] due to excessive execution times, Program B was only evaluated with 10 unroll factors

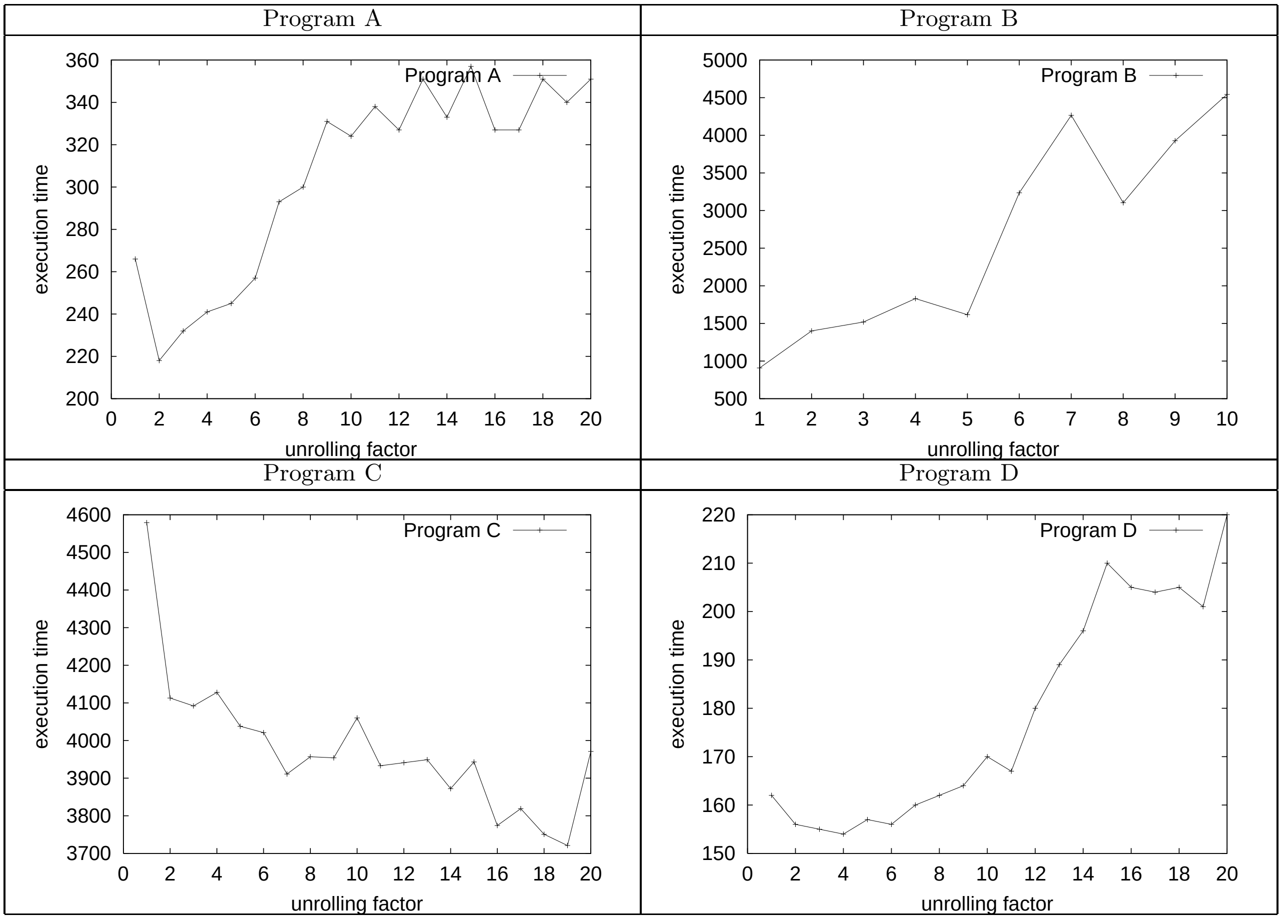

Figure 4. Execution time vs unrolling

this transformation is not applicable to Program C. This transformation had a significant impact on execution time across all the programs where it was applicable, giving a speedup of between 1.21 and 1.33. Array hoisting is the placing of invariant accesses outside of a loop and loading the value into a scalar. It can be considered to be a generalisation of scalar replacement. In the case it was applicable C, it gave a 41% improvement in execution time. Thus reducing access to array structures improves execution time across all programs.

Linearisation Each of the multi-dimensional rectangular static arrays was linearised to a one dimensional form and the array accesses updates accordingly. In one case, C, this actually caused a slow down, possibly due to the introduced complex subscript expressions. In the case of D, it had negligible impact and in the cases of A and B it gave up to a 22% improvement in execution time. This transformation therefore has variable impact and should be applied cautiously.

Loop Distribution Loop distribution could only be applied to two of the 4 cases, A, B either because there was only one statement in the loop or distribution was illegal. Although it is primarily an enabling transformation, it is interesting to see its impact. In the case of A it has a severe negative effect while for B it gives only a marginal improvement.Thus as a stand-alone transformation it needs to be used with caution

All transformation While transformations individually can improves performance, we are ultimately interested in the combination that gives the best performance. Therefore we evaluated all legal combinations of tiling,

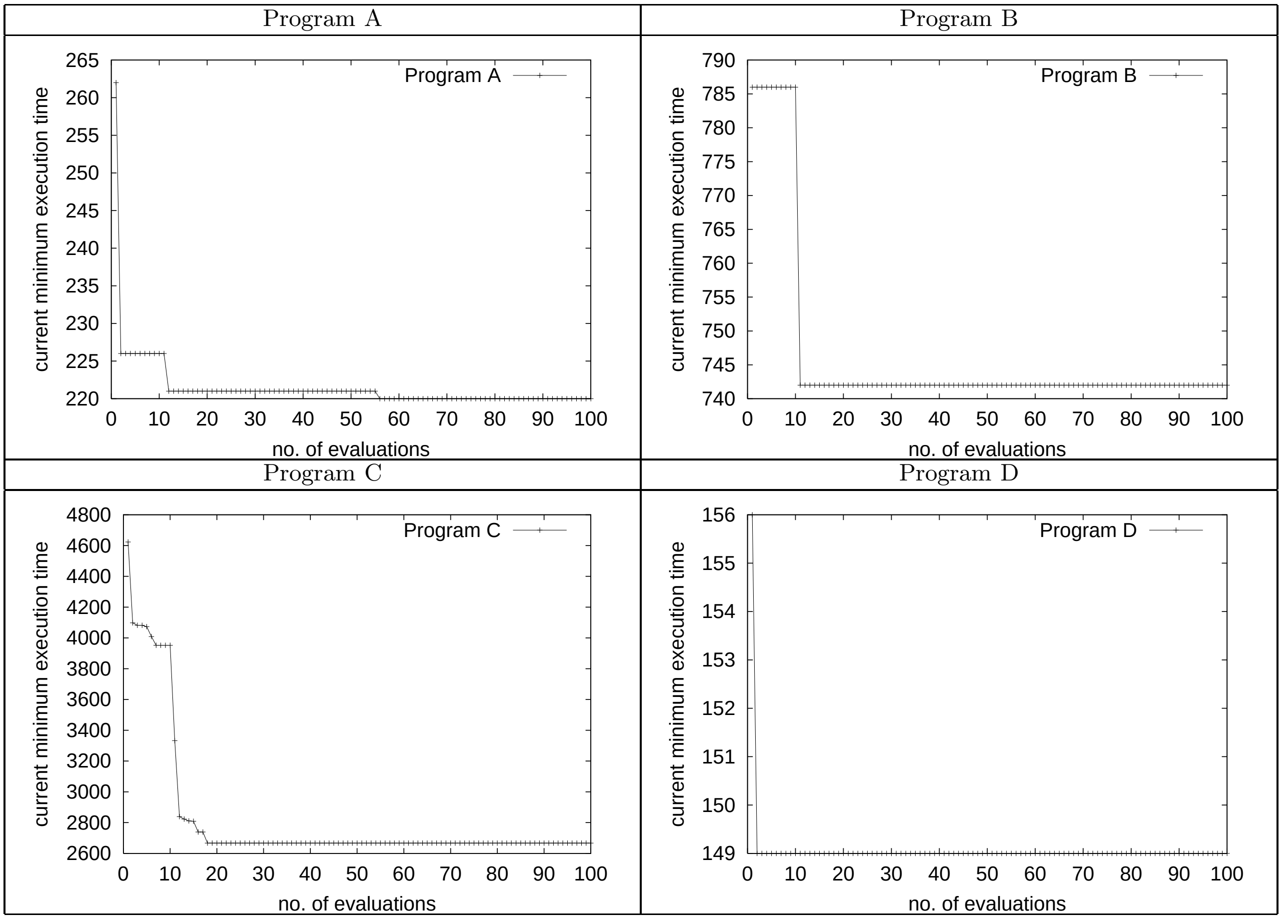

Figure 5. Current best execution time vs number of evaluations

unrolling, scalar replacement, hoisting, linearisation and distribution and found that in all cases the effect of combined transformations outperformed any one transformation in isolation (see figure 2).

It is obvious that the transformation spaces shown in these above figures are highly non-linear, containing many local minima and discontinuities. This mirrors the behaviour found on other systems [4] and is largely due to the interaction between the underlying memory architecture and the JVM. It shows that although static models and analysis may improve program performance, actual dynamic behaviour is compile-time intractable, supporting the need for feedback directed approaches.

The above results show that combined loop transformations can significantly improve program performance. However, we have not considered how easy it is to develop a compiler strategy that finds such optimisations in an efficient manner; indeed developing such a strategy is an extremely difficult problem. The next section investigates the use of feedback directed optimisation as a means of determining good combined program transformations.

5 Feedback Directed Optimisation (FDO)

Given the above variability of program performance with respect to program transformation we would like a technique that automatically selects good program transformations. Rather than using static models that are inherently architecture specific, we use a feedback directed optimisation (FDO) or iterative compilation approach to determine a good optimisation in a short time. We applied the grid based algorithm described in [4] and determined how many evaluations or versions of the program that need to be considered before a good optimisation is selected.

5.1 Grid-Based Search Algorithm

The grid-based search algorithm given in [4] can be briefly described as follows.

1. First define a coarse grid on the search space,
2. Evaluate all points on this grid by generating the transformed programs and executing them,
3. Find the point with minimum execution time and all points that are within an allowable distance from this minimum,
4. Order these points in a priority queue,
5. For each points in the queue, if the execution time is within an allowable distance from the current minimum, then refine the grid around this point, otherwise neglect it. If new points are found close enough, enqueue them in the priority queue.

5.2 Results

We applied the FDO search to the problem of finding the best tile and unroll factor for each of the 4 programs. Tiling and unrolling was selected as there are a relatively large number of points - 2000, the transformations are applicable to all the programs and it has been extensively evaluated allowing comparison with other work [9] .

We plotted the execution time of the best version currently found by the FDO algorithm against the number of evaluations required to find it, the results of which are shown in figure 5. In each case the lowest point on the y-axis represents the best execution time possible or absolute minima in the transformation space under consideration.

As can be seen, in all cases the algorithm quickly found good transformations in a relatively short number of evaluations. In all cases, an improved transformation is found within 10 evaluations and 95% of the possible speedup available is found within 15 evaluations in every case. Thus relatively few transformations have to be tried to find good optimisations.

6 Discussion, Conclusions and Future work

The previous sections have shown that (i) high level program transformations can significantly improve Java performance and (ii) it is possible to determine good optimisations with relatively few runs of the program. Although this paper has considered searching for the best optimisation based on an off-line algorithm, it is our intention to make this process on-line shortly. This will have the added benefit of providing runtime parameters such as loop trip count which are critical for performance but unavailable using a solely off-line technique.

The key question is how applicable is feedback directed high level optimisation for an adaptive Java compiler? Clearly, using the decoupled code-generation model of ADAPT [11], the sections of code above would have to be executed a number of times before a good transformations (found remotely) could be applied to the program and the running binary updated. If a separate processor were unavailable, then the AJOF system would intermittently investigate different versions of the program only if the program section were evaluated many times. In cases where the code under investigation is executed many thousands of times, then a execution-time only driven feedback mechanism will give excellent results [9]. However, when the number of evaluations is expected to be much shorter, then static analysis can be used to help steer the search for good optimisations [9].

Feedback directed optimisation is inherently adaptable to both program and architectural change and thus is eminently suitable for the environment we wish to consider. Program - construct and architecture specific static analysis can also be used, where available, to improve the generic approach. Thus we have a flexible optimisation system that is also able to make use of analysis where available.

This paper has described a framework for adaptively optimising Java programs. High level source to source transformations are inherently portable across platforms and this paper has primarily focused on the applicability and impact of such transformations on Java performance. We have shown that they can have a significant impact, giving up to an 86% improvement in performance. However, determining the best transformation sequence has historically been shown to be very difficult. Rather than using architecture specific analysis we have applied a generic feedback directed optimisation approach which finds 95% of the the speedup available within 15 evaluations. Future work will investigate the use of optimisation at the bytecode and binary level and the use of long-term profile data in determining good optimisations within the AJOF framework. We also intend to empirically validate our results over a much broader range of benchmarks.

References

[1] M. Arnold et.al, Adaptive Optimization in the Jalapeno JVM, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000), Minneapolis, Minnesota, October 2000.

[2] P.V. Artigas M. Gupta S. Midkiff and J. Moreira, Automatic Transformations and Parallelization for Java, ICS, Santa February 2000.

[3] A.Bik, J. Villacis, and D. Gannon, Javar - a prototype Java reconstructing compiler, Concurrency, Practice and Experience 9(11), November 1997.

[4] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou, Iterative Compilation in a Non-Linear Optimisation Space, Proceedings of the Workshop on Profile Directed Feedback-Compilation, PACT'98, October 1998.

[5] M.Burke et.al, The Jalapeno Dynamic Optimizing Compiler for Java, Proc. of ACM'99 Java Grande Conference, June 1999.

[6] M.Bull et.al, A Benchmark Suite for High Performance Java, Concurrent: Practice and Experience, 12, 2000.

[7] M. Ciernack and W. Li, Just in time optimization for high performance Java programs, Concurrency Practice and Experience, 9(11) 1063-73 November 1997.

[8] K. Ebcioglu and E.R. Altman, "DAISY: Dynamic Compilation for 100 % Architecture Compatibility", TRRC 20538, IBM Research, 1996.

[9] T. Kisuki, P.M.W. Knijnenburg and M.F.P. O'Boyle, Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation, Proc. PACT 2000, Parallel Architectures and Compiler Technology, *IEEE Press*, October 2000.

[10] M. Serrano, R. Bordawekar, S. Midkiff, M. Gupta, Quasi-Static Compilation for Java, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000), Minneapolis, Minnesota, October 2000.

[11] M. Voss and R. Eigenmann, A Framework for Remote Dynamic Program Optimization, Proc. of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00), 2000;