Evaluating Iterative Compilation

G.G. Fursin[1], M.F.P. O'Boyle[1], and P.M.W. Knijnenburg[2]

[1] ICSA, School of Informatics,University of Edinburgh, United Kingdom.
[2] LIACS,Leiden University, the Netherlands.

Abstract. This paper describes a platform independent optimisation approach based on feedback-directed program restructuring. We have developed two strategies that search the optimisation space by means of profiling to find the best possible program variant. These strategies have no *a priori* knowledge of the target machine and can be run on any platform. In this paper our approach is evaluated on three full SPEC benchmarks, rather than the kernels evaluated in earlier studies where the optimisation space is relatively small. This approach was evaluated on six different platforms, where it is shown that we obtain on average a 20.5% reduction in execution time compared to the native compiler with full optimisation. By using training data instead of reference data for the search procedure, we can reduce compilation time and still give on average a 16.5% reduction in time when running on reference data. We show that our approach is able to give similar significant reductions in execution time over a state of the art high level restructurer based on static analysis and a platform specific profile feedback directed compiler that employs the same transformations as our iterative system.

1 Introduction

Traditional approaches to compiler optimisations are based on static analysis and a hardwired compiler strategy which can no longer be used in a computing environment where the platform is rapidly changing. Modern architectures have very complex internal organisations: high issue widths, out-of-order execution, deep memory hierarchies, etc. However, compiler machine models are necessarily simplified to be tractable and only take into account a small part of the actual system. Such models provide very rough performance estimates which, in practice, are too simplistic to statically select the best optimisations. What is required is an approach which evolves and adapts to architectural change without sacrificing performance.

This paper examines a feedback assisted approach based on traversing an optimisation space. Early results suggest that such an approach can give significant reductions in execution time over purely static approaches with, on average, a 20.5% improvement over the highest optimisation levels provided by the native compiler. Although such an approach is usually ruled out in terms of excessive compilation time, it is precisely the approach used by expert programmers when the application is to be executed many times. Embedded systems are an extreme example of this, allowing the cost of compilation to be amortised over many shipped products.

In previously published work [2, 7], we have shown the use of *iterative compilation* in optimising program performance. Different transformations are applied, corresponding to points in the transformation space, and their worth evaluated by executing the program. Several evaluations, based on a compiler search strategy, are performed to a certain pre-defined maximum number, with the compiler selecting the best one. Related work in the area of linear algebraic libraries has also shown good performance [13].

However, the main drawback of previous work is that it has focused solely on tuning compute-intensive kernels where the optimisation spaces being searched are relatively trivial. Clearly, for iterative compilation to be considered a realistic optimisation approach, it must be shown to be able to find good results on the large spaces that arise for realistic applications with a relatively few number of evaluations.

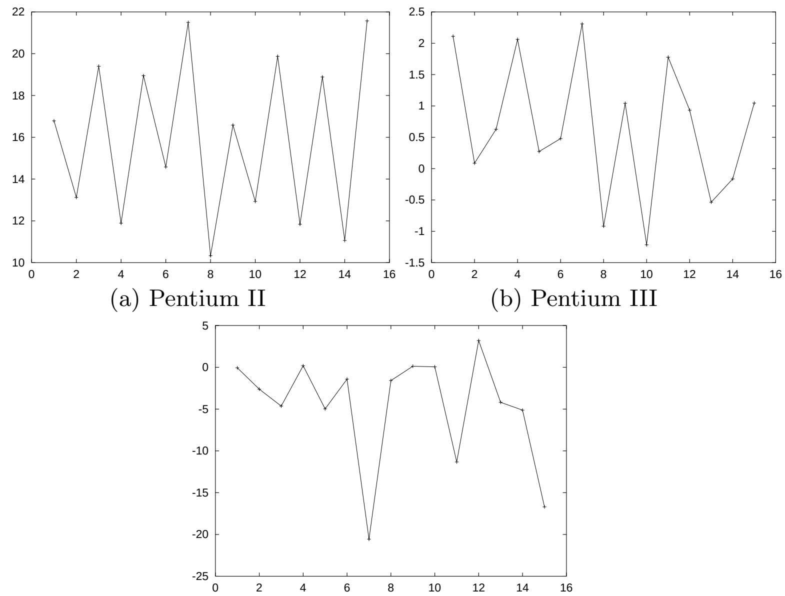
Although iterative approaches can find good results, they may be inappropriate if the data size, for instance, is different from that actually encountered at runtime. In order to investigate this phenomenom, we applied our approach to training data before applying the selected transformation to distinct reference data. In all cases our approach outperforms the native optimising compiler.

Finally, we compared our approach to a state-of-the-art profile driven optimiser that is present in the Compaq compiler for the Alpha processors. There are many optimisations used in this optimiser, including all of the high level source to source transformations that are used by our system, plus many others. This optimiser collects runtime data to steer its optimisation process, like our approach. However, unlike our approach, it uses this data, by certain fixed heuristics, in a fixed strategy. We show that our searching techniques outperform this static approach significantly, even though the static profile driven optimiser has access to additional transformations not considered by our scheme that can dramatically improve execution time, such as software pipelining.

The paper makes the following contributions:

– For the first, time it demonstrates that iterative compilation outperforms static approaches on realistic non-kernel benchmarks.
– It demonstrates that good optimisations can be found with variable runtime data.
– It demonstrates significant reductions in execution time compared to a state-of-the art native high level restructurer that employs statically (among others) the same transformations as our system with few evaluations.
– It demonstrates significant reductions in execution time over an existing platform specific feedback directed optimiser that employs (among others) the same optimisations as our system.

This paper is organised as follows. Section 2 describes the benchmarks and platforms investigated. Section 3 shows comparatively how performance is affected by different transformations. Section 4 describes the overall compiler infrastructure and the iterative compilation strategies implemented. This is followed in Section 5 by an evaluation of this approach. Section 6 provides a brief review of related work and Section 7 provides some concluding remarks.



(a) Pentium II (b) Pentium III

(c) HP-PA

Fig. 1. Percentage reduction in execution time for varying pad sizes: Swim

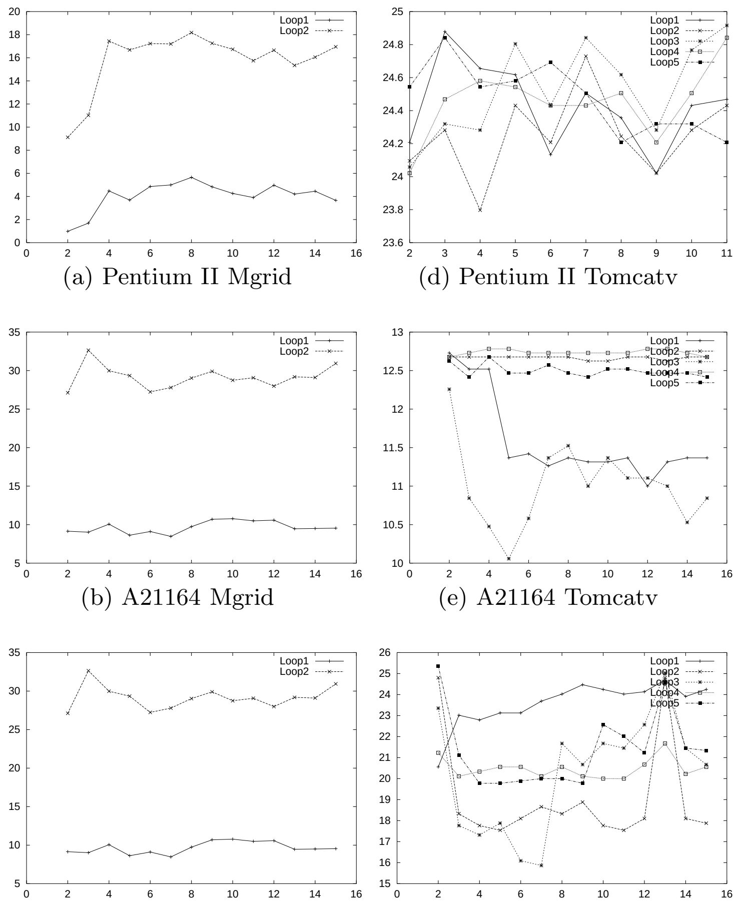
2 Benchmarks and Platforms

We consider the following SPEC95 FP benchmarks: `Tomcatv`, `Swim` and `Mgrid` with the reference data input sets. The following platforms are used:

Alpha 21164 500MHz. In-order. Digital UNIX V4.0D. F77 V5.0. 8K L1.
Alpha 21264 500MHz. Out-of-order. Digital UNIX V4.0E. F77 V5.2. 64K L1.
Pentium II 350 MHz. Windows 2000 Professional. Compaq F77 V6.1 16K L1.
Pentium III 600MHz. Red Hat Linux 6.1. g77 2.95.1. 16 k L1.
HP-PA 9000/712 80 MHz. OS A.09.07 F77.9.0. 128K L1.
Ultrasparc 300 MHz. SunOS 5.7, g77 2.95.1. 16K L1.

All comparative experimental data is with respect to the native compilers at their highest optimisation level. We later compare our approach against the Compaq high level restructurer which is only available on the Pentium and the 2 Alphas. The Compaq compiler with the optimisation level set to -O5 becomes a high level restructurer which applies all of the transformations of our system. This compiler, moreover, applies other loop transformations as well, including software pipelining that is well known to boost program performance.

Furthermore, on the Alpha platforms this compiler allows profile driven optimisation where it uses runtime data to drive these loop transformations. We compare our approach against this option also.



(a) Pentium II Mgrid (d) Pentium II Tomcatv

(b) A21164 Mgrid (e) A21164 Tomcatv

(c) A21264 Mgrid (f) A21264 Tomcatv

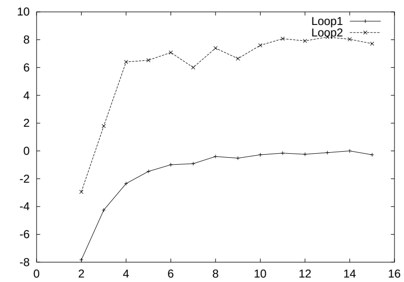
Fig. 2. Percentage reduction in time for varying unroll factors: Mgrid + Tomcatv

3 Impact of Program Transformations

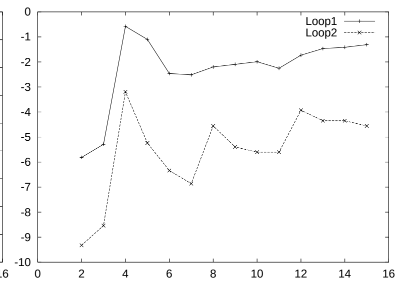
It is well-known that program transformations have a variable impact on program performance and that finding the best transformation sequence is a difficult task. In this section we wish to empirically demonstrate not only the non-linear impact of program transformations, but how this varies across machines, demonstrating the challenge in developing generic compilers that can adapt to different platforms.

3.1 Transformations

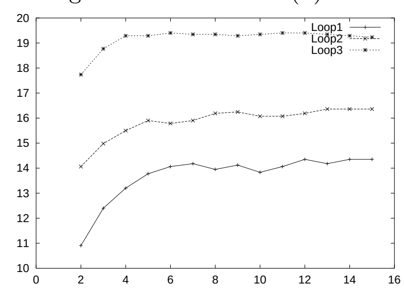
Here we examine the impact of 3 well known transformations, array padding, loop unrolling and tiling on selected benchmarks and platforms.



(a) A21164 Mgrid



(b) A21164 Tomcatv



(c) A21164 Swim

Fig. 3. Percentage reduction in execution time for varying tile sizes: A21164

Padding Array padding is used to reduce conflict misses in cache based architectures [10]. Figure 1 shows the reduction in execution time due to padding with respect to the original code on Swim across three of the platforms. This oscillatory behaviour is not particularly surprising and is well studied [8], however, it does highlight the difficulty for an optimising compiler in determining whether array padding should be considered and finding the best factor, particularly when moving from one platform to another. For instance, on the Pentium II, array padding gives, on average, a clear improvement, even if small changes in parameter values give wide variation in behaviour. In the case of the Pentium III, however, it has little impact on performance while for the HP-PA it should generally be avoided.

Loop Unrolling Loop unrolling is a well known optimisation used to expose more instruction level parallelism (ILP) to the back end scheduler and reduce the relative overhead of memory access. Figure 2 show the impact of loop unrolling on 21162, 21264 and the Pentium II when applied to Mgrid and Tomcatv. We have highlighted the impact on the most time consuming loops: two in the case of Mgrid and five in the case of Tomcatv. In absolute terms there is much less variability than in the case of padding, though clearly in the case of Mgrid, unrolling loop 1 gives a much greater reduction in execution time than loop 2. Similarly the best unroll factor varies from platform to platform. In the case of Tomcatv, all 5 loops benefit from unrolling and there is generally no large absolute differ-

	PII	PIII	HP-PA	US	21164	21264	Avg.
Tomcatv	31.4	25.3	38.6	22.6	13.5	25.4	26.1
Swim	21.7	2.31	8.35	17.73	22.6	40.0	18.8
Mgrid	18.1	1.29	17.38	15.1	32.6	15.4	16.6
Avg.	23.7	9.63	21.4	18.5	22.9	26.9	20.5

Table 1. Strategy 1: Percentage reduction in execution time

ence between different unroll factors. However, unrolling by a factor of 5 on loop 3 on the A21164 gives particularly poor performance, and an unroll factor of 13 on the A21264 seems surprisingly beneficial to all loops.

Loop Tiling Loop tiling [8] is used to improve cache utilisation by exploiting temporal and spatial locality. Figure 3 show the impact of loop tiling on the three benchmarks on the A21164. Here we again highlight two of the main loops in Mgrid, this time we just focus on two loops for Tomcatv and three for Swim. In the case of Mgrid, tiling is beneficial for tile sizes greater than 4 for loop 1 but should be avoided for loop 2. It is beneficial for all tile sizes in the case of Swim, with those greater than 14 giving the greatest reduction in execution time. However, tiling always gives poor performance on Tomcatv, due to the lack of intra-loop locality within this program [8].

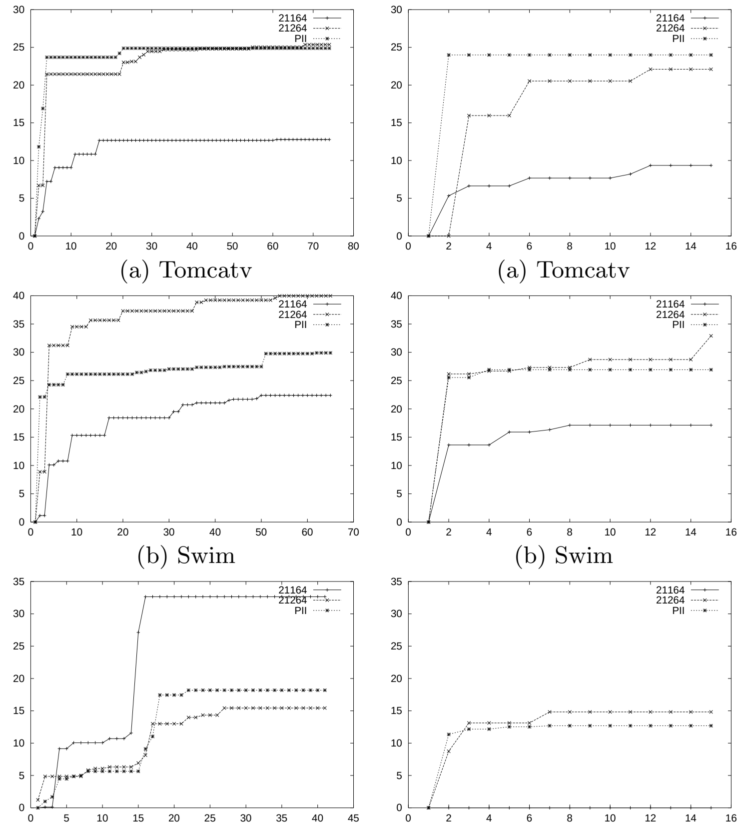
Although the impact of program transformations has been well studied, this section has shown that high-level transformations can have a significant impact on performance even when compared to modern high performance native compilers. It is not the intention of this paper to *explain* processor behaviour in the presence of transformations, rather that, as the figures shown in this section suggest, such behaviour will be difficult to accurately predict. Thus, designing an optimising strategy that works well across such platforms is highly non-trivial. In the next section we develop two compiler strategies that are intended to be suitably generic.

4 Compiler Strategies

The main objective of a compiler strategy is to decide which transformations to apply, guided by information in the form of static analysis, execution time, or heuristics which are meant to reduce the transformation space to consider. While the majority of research in optimisation via high level restructuring has relied on static information, here we are primarily concerned with developing techniques that have no architectural knowledge and are solely based on dynamic information.

4.1 Strategy 1

This strategy uses data and loop transformations in a cost-conscious manner. Rather than search through a large space of all possible loop and data transfor-



(a) Tomcatv (a) Tomcatv

(b) Swim (b) Swim

(c) Mgrid (c) Mgrid

Fig. 4. Strategy 1 and 2 : The reduction in execution time of the best transformation found so far by strategy 1 wrt the number of evaluations. Performed on 3 platforms

mations, it targets those sections of the program that dominate program execution and considers restricted loop and data transformations in separate phases reducing the number of combinatorial options by imposing a phase order.

Initially, the program is profiled and those subroutines that dominate execution time are marked. Within each marked routine, those loop nests that dominate execution are also marked as are the arrays referenced within them. After this initially marking phase, we consider data transformations on those arrays marked as significant. As data transformations are global in effect, they are considered first on the assumption that local loop transformations can later compensate for some adverse effects that can be caused locally by the global data transformations. In this strategy, the only data transformation considered is array padding and this is applied to the first dimension of the marked arrays inter-procedurally. If there are p padding factors to consider and a arrays, then

	PII	21164	21264	Avg.
Tomcatv	24.6	9.9	22.0	18.8
Swim	14.5	19.0	33.0	22.6
Mgrid	14.5	30.5	14.8	19.9
Avg.	17.8	19.8	23.2	20.2

Table 2. Strategy 2: Percentage reduction in execution time

the number of different padding combinations is p^a. To reduce this complexity, we pad each array the same amount, reducing the complexity to p. For this new padded program, we now consider loop transformations.

Loop tiling (with tile sizes ranging from 2 to the range of the loop bounds) is considered for all those loop nests marked initially as significant. Each loop nest is considered in turn and tiled. When the best tile size is determined, it is recorded before moving on to the next loop nest. To avoid combinatorial explosion, each loop is optimised in isolation, ignoring the effect of transforming one loop on the rest of the program. Once the tile factors for each significant loop have been determined, they are all applied to give a new program. Finally loop unrolling is applied in a similar manner.

4.2 Strategy 2

This strategy again focuses on the three transformations considered before: array padding, loop tiling, and loop unrolling. Once again, profiling is used to determine those arrays and loop nests of interest. This time, however, rather than combine the best padding, tiling and unroll factors, we randomly search for the best combination. One or more loops and arrays are randomly selected and random tile, pad and unroll factors applied. This avoids the coupled behaviour of transformations (where the best form of one transformation plus the best form of another gives a sub-optimal value when combined), without having to exhaustively search a large space.

Both strategies retain the best version found so far at each evaluation, so that after evaluating a fixed number of transformed programs, the best transformed program is returned as the final selected program. As is immediately apparent, neither of these strategies contain any platform or program specific information. The next section evaluates to what extent they may improve performance.

5 Experimental Results

In this section we evaluate the two iterative search strategies. This is followed by an evaluation of the use of smaller training data as a mechanism to reduce overall compilation time.

	PII	PIII	HP	US	A21164	A21264	Avg.
Tomcatv	32.5	25.3	38.6	22.6	11.9	19.4	25.05
Swim	21.5	0.09	3.2	14.0	23.1	38.6	16.7
Mgrid	12.1	0	0	0	31.8	5.1	8.16
Avg.	22	8.5	13.9	12.21	22.3	21	16.63

Table 3. Strategy 1: Training Data: Percentage reduction in execution time

Finally, we evaluate our iterative approach against an existing high level restructurer and a feedback directed optimiser that employ (among others) the same transformations as our iterative system.

5.1 Evaluating iterative search strategies

The first search strategy was allowed to run for 200 evaluations[3] and Table 1 shows the reduction in execution time found across the platforms and benchmarks. In all cases we improve on the best obtainable performance of the native compiler and give on average a 20.5% reduction in execution time. Tomcatv is most improved by program optimisations considered in this paper and Swim the least, though on the 21264 a 40% improvement is found. Comparing different platforms, the 21264 is most improved by the program optimisations considered in this paper and the PIII the least.

In Figure 4 we show how the first search strategy performs with respect to the number of evaluations. The reduction in execution time of the current best program version is shown for three of the six different platforms across the three benchmarks. At each evaluation a new program version is selected by the strategy. If the new program selected is an improvement on the best version so far, we see an improvement in execution time reduction and the new program becomes the current best version. Otherwise the current best version is retained and we see no change in execution time reduction. In the case of Tomcatv, the most significant performance gains are made within 40 evaluations. In the case of Swim, higher performance gains are made for these three platforms, taking approximately 40 evaluations to find the majority of the available performance gains. Finally, in the case of Mgrid, after just 18 iterations the search strategy finds good program optimisations across the three platforms.

Despite the complex behaviour of transformations across platforms and their interaction with each other, our first iterative strategy has shown that it can perform well across all platforms. Interestingly, the rate at which it finds good candidate optimisations is broadly similar for each target machine.

Although the first strategy finds good performance with relatively few evaluations, this may still be too time consuming in practice. Therefore, we now

[3] An evaluation consists of 3 parts: (*i*) transform the program, (*ii*) compile it with the native compiler, and (*iii*) execute the program.

	PII	A21164	A21264	Avg.
Tomcatv	25.4	11.2	22.3	19.6
Swim	28.4	23.9	35.7	29.3
Mgrid	11.5	32.7	4.3	16.16
Avg.	21.7	22.6	20.76	21.68

Table 4. Strategy 2: Training Data: Percentage reduction in execution time

evaluate Strategy 2 with a maximum of 15 evaluations and compare its performance against the native compiler, restricting our attention to 3 of the 6 platforms. What is immediately apparent from the results in Table 2, is that the second strategy is able to find considerable reductions in execution time despite the small number of evaluations. On the Pentium, it achieves 75% of the performance found using Strategy 1 and over 85% on the two Alphas.

If we examine in detail how fast the strategy finds good results as shown in Figure 5, we find that within just 5 evaluations, significant reductions in execution time are found. Considering the size of the optimisation search space considered, this is a significant result.

5.2 Evaluating the use of training data to determine transformation

Although we have shown that our approach outperforms native compilers in every case with relatively few evaluations, this still may be too expensive. In this experiment we therefore use the smaller training data (and hence shorten evaluation time) from the SPEC benchmark suite in order to find a good optimisation and then apply the resulting best optimisation to the actual reference data. Use of the training data will also give an insight into how iterative compilation performs in the presence of different data sets and sizes.

As can be seen in Table 4, the first iterative strategy using training data never performs worse than the native optimiser and in the majority of case gives significant reduction in execution time. On average there is a 16.63% improvement which compares favourably with the 20.5% average found using solely the reference data (Table 1). Using training data we reduce the evaluation time and obtain over 80% of the execution time reduction when using the actual reference data. In the case of Mgrid, performance gain was found on only 3 of the 6 platforms, showing that its performance is more closely related to the actual runtime data.

If we apply the second strategy with just 15 evaluations to three of the six platform, we find the results shown in Table 5, where we have on average a 21.68% improvement. If we compare the execution time reduction of Strategy 2 against Strategy 1 on each machine, we see that their performance is almost identical when using training data. Furthermore, if we compare the execution time reduction of Strategy 2 using training data with the performance obtained using reference data (Table 2), the training data actually gives slightly better results due to the random nature of the search strategy.

5.3 Comparison against an existing Static High Level Restructurer

The previous sections have shown that iterative compilation can give good performance improvements over the native compiler in relatively few iterations and in the presence of smaller training data.

In order to further evaluate the use of iterative compilation in high level restructuring, this section compares our approach to an industrial high level restructurer. The Compaq compiler has an option (-O5) which enables high level restructuring and is integrated within the entire compiler chain. This restructurer uses an elaborate phase ordered strategy based on sophisticated static analysis and considerable architectural knowledge. Loop transformation optimisations that are used by the Compaq compiler include loop blocking, loop distribution, loop fusion, loop interchange, loop scalar replacement, and outer loop unrolling. It moreover employs array padding and software pipelining. Hence this compiler uses the same transformations as our iterative system, and several that are not implemented by us.

We applied Strategies 1 and 2 to the three platforms (where the Compaq compiler is available) with high level restructuring enabled. Thus, we are applying high level transformations which are then fed into a native compiler which may, in turn, apply further high level transformations. The results are given in Tables 6 and 7. Overall Strategy 1 is able to reduce execution time by 19.6% and Strategy 2 by 14.9%. Thus a techniques that evaluates just 15 program transformations is able to give significant execution time reduction when compared to a state of the art optimiser.

In only one case does Strategy 2 fail to make an improvement and in this case simply achieves the same performance as the native high level restructurer as we are using the native high level restructuring as our backend compiler. This ability to make use of the best available vendor supplied compiler technology is a useful feature of our approach. However, for a strictly fair comparison, we should compare our approach using the native low level optimiser (-O4) as our backend compiler, to Compaq's high level restructurer (-O5) which also makes use of the native low level compiler as its backend compiler. In such a case, strategies 1 and 2 give 9.92% and 8.6%, respectively, reduction in execution time when compared to the Compaq high level restructurer, using the same native low level optimiser.

Thus we are able outperform an existing high level restructurer, and furthermore can use that same restructurer as a backend to further improve performance. The ability to adapt to improvements in vendor supplied system software is a useful feature of our approach.

5.4 Comparison against an existing Profile-directed Compiler

We have shown that our technique outperforms native compilers, with full optimisation enabled, and an existing static high level restructurer. Here we show

	A21164	A21264	PII	Avg.
Tomcatv	12.3	25.4	22.3	20
Swim	27.9	38.2	20.3	28.8
Mgrid	4.3	10.5	18.0	10.9
Avg.	14.8	24.7	20.2	19.6

	A21164	A21264	PII	Avg.
Tomcatv	8.3	23.7	14.6	15.5
Swim	20.1	31.8	11.9	21.2
Mgrid	0	8.3	16.1	8.13
Mgrid	12.23	21.26	14.16	14.9

Table 5. Strategy 1 and 2: Percentage reduction in execution time wrt a high level restructurer -O5

our generic approach also outperforms an existing profile-directed compiler. The Compaq compiler on the two Alphas has accesses to low level profile tools that allows it to gather information during one execution in order to improve code generated for the next run [5]. This can be done under two modes: with full low level optimisation on (-profile -O4) and with full low level optimisation on *plus* high level restructuring (-profile -O5). Hence on the Alphas the Compaq compiler can drive the same loop transformations as our system using profile data. The difference between the Compaq compiler and our system lies in the fact that the Compaq compiler uses predefined heuristics in a predefined order to select transformations whereas our system performs a search procedure. In this section we show that searching outperforms highly tuned static heuristics significantly.

We plotted the speedups of these two different modes against the native compiler in Figures 6, 7, and 8 (-profile -O4, -profile -O5). We also plotted the speedups of all other approaches described in this paper. Namely, the original native (-O4) execution time, the native high level restructurer time (-O5). For further comparison, we also plotted the results of both strategies using the native compiler as the backend compiler (-it st 1/2 -O4), and the native compiler with high level restructuring enabled as the backend (-it st 1/2 -O5).

As can be immediately seen, the iterative approaches outperform the Alpha's profile directed approach in all cases. Furthermore, iterative compilation with a simple native compiler (-it st 1/2 -O4) even outperforms the profile directed approach using high level restructuring (-profile -O5) in most cases. In the majority of cases the A21264 benefits more from optimisation than the A21164, except in the case of mgrid where (-O5) optimisation dramatically improves performance for both the profile directed and iterative approaches on the A21164. Interestingly, the Alpha's profile directed compilation actually performs better without the use of high level restructuring on the A21164. It is not immediately apparent why this is the case, possibly high level restructuring may interfere with the profiler.

Overall, Strategy 1 reduces the execution time on average by 16.52% when compared to the profile directed compiler, while Strategy 2 reduces the execution time by 12.48%.

Once again these performance gains are made with the Compaq high level restructurer (-O5) as our backend compiler. However, for a strictly fair compar-

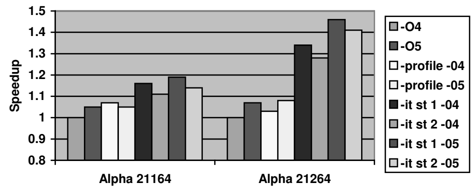


Fig. 5. Tomcatv: Speedup

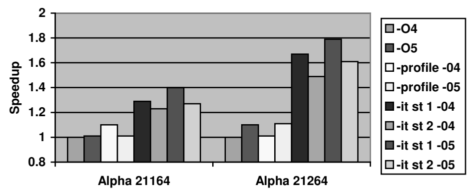


Fig. 6. Swim: Speedup

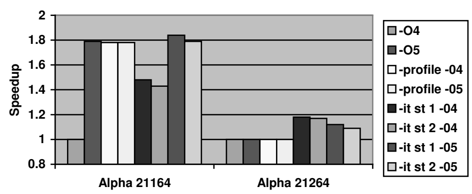


Fig. 7. Mgrid: Speedup

ison, we should compare our approach using the native low level optimiser (it st 1/2 -O4) as our backend compiler against Compaq's profile directed approach using a high level restructurer (-profile -O5) which also makes use of the native low level compiler as its backend compiler. In such a case, strategies 1 and 2 give 9.8% and 8.5%, respectively, reduction in execution time when compared to the Compaq's profile directed, high level restructurer; both using the same native low level optimiser.

Thus, our generic approach outperforms even highly optimised platform specific, feedback directed approaches.

6 Related Work and Discussion

Feedback directed optimisation [11] is a basic technique used in computer architecture where hardware resources are dedicated to tracing and predicting program behaviour. Similarly, in low-level compilers profile guided compilation is widely used to determine execution path, allowing improved program optimisation.

Due to the limits of static analysis, systems for generation highly optimised versions of BLAS routines are proposed which probe the underlying hardware for platform specific parameters [13, 1]. In the SPIRAL project a feedback directed search approach is applied to DSP algorithms that can be expressed as tensor products Within this domain, excellent versions of DSP algorithms can be found in a relatively short number of executions [9].

Wolf, Maydan and Chen [14] have described a compiler that also searches for the optimal optimisation based on a fixed order of the transformations. However, they solely use a static cost model to evaluate the different optimisations which inevitably approximates system behaviour and does not adapt to architectural change. A similar approach has been taken by Han, Rivera and Tseng [6] which uses a model to search for tile and pad sizes; again such an approach is restricted by the use of static models. Finally, Chow and Wu [4] apply “fractional factorial design” to decide on the number of experiments to run for selecting a collection of compiler switches, rather than trying to explore a program optimisation space in a platform independent manner.

Feedback directed high level transformations have also recently become more popular. In [12] a framework is described which allows remote on-line optimisation of a program while it is running, gaining the benefits of actual knowledge of runtime parameters without the overhead of compilation on the critical path. Our approach is similar in spirit in that different optimisations are tried and the best selected, theirs on-line ours off-line. However, the main distinction is that we have developed generic search strategies based on investing a systematic transformation optimisation space. Dynamic online optimisations found in Java just-in-time compilers [3] also make use of runtime behaviour in determining program optimisation. However, such approaches only consider a fixed predetermined number of optimisations.

7 Conclusion

This paper has described an aggressive compiler framework that outperforms static optimisation approaches and that allows optimisers to adapt to new platforms by way of feedback directed iterative compilation. By decoupling strategy from implementation, we have implemented two architecture blind generic optimisation approaches. These rely on our framing the problem of optimisation as that of traversing a transformation space in order to minimise the object function of execution time. We have shown that for three SPEC FP benchmarks, across six platforms, we reduce the execution time by 20.5% on average. When restricting the number of evaluations to just 15, we achieve a reduction of 20.2% across 3 of the platforms. We have also shown that good performance can be achieve when smaller training data is used giving over 80% of the performance achieved using reference data.

For a fair comparison, we compared our approach to that of a native high level restructurer. Using the same native backend compiler we obtain a reduction in execution time of almost 10% on average. Moreover, if we compare our approach to a platform specific profile directed high level optimiser that employs the same transformations as our system plus several more, we also obtain a reduction in execution time of almost 10% on average. Furthermore, we are able to adapt and use the high level restructurer as our backend compiler, where we are able to further improve performance, reducing execution time by 12% on average when compared to the platform specific profile directed high level optimiser.

We have shown, for the first time, that iterative compilation is viable for large optimisation spaces found in general programs and that good performance may be achieved regardless of platform. Future work will investigate both the use of models to further reduce the number of evaluations required and evaluate other search strategies.

References

1. J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. **Optimizing matrix multiply using PHiPAC: A portable, high-performance, C coding methodology**, ICS'97,1997.
2. F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou **Iterative Compilation in a Non-Linear Optimisation Space** , Profile and Feedback Directed Compilation, PACT, 1998.
3. M.Burke et.al, **The Jalapeno Dynamic Optimizing Compiler for Java**, Proc. of ACM'99 Java Grande Conference, June 1999.
4. K. Chow and Y. Wu, **Feedback-Directed Selection and Characterization of Compiler Optimizations**, FDO, 1999.
5. R.Cohn and P.G. Lowney, **Feedback Directed Optimization in Compaq's Compilation Tools for Alpha**, FDO, 1999.
6. H. Han, G. Rivera and C.-W. Tseng, **Software Support for Improving Locality in Scientific Codes**, CPC, 2000.
7. T. Kisuki, P.M.W. Knijnenburg and M.F.P. O'Boyle, **Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation**, PACT, 2000.
8. K. S. McKinley and O. Temam., **A Quantative Analysis of Loop Nest Locality**, ASPLOS, 1996.
9. J. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, M. Puschel, B. Singer, M. Veloso, and J. Xiong. **Generating Platform-Adapted DSP Libraries using SPIRAL**, Proc. HPEC 2001, MIT Lincoln Laboratories.
10. G. Rivera and C.-W. Tseng, **Data Transformations for Eliminating Conflict Misses**, PLDI, 1998.
11. M. Smith, **Overcoming the Challenges to Feedback-Directed Optimizations**, Dynamo'00, 2000.
12. Voss M.J. and Eigenmann R., **A framework for remote dynamic Program Optimization**, Dynamo, 2000.
13. R.C. Whaley and J.J. Dongarra. **Automatically tuned linear algebra software**. Proc. Alliance, 1998.
14. M.E. Wolf, D.E. Maydan, and D.-K. Chen. **Combining loop transformations considering caches and scheduling**. *Int'l. J. of Parallel Programming*, 26(4):479–503, 1998.