A Feasibility Study in Iterative Compilation*

Toru Kisuki[1], Peter M.W. Knijnenburg[1], Mike F.P. O'Boyle[2], François Bodin[3], and Harry A.G. Wijshoff[1]

[1] Leiden Institute of Advanced Computer Science, Leiden University
Leiden, the Netherlands

[2] Division of Informatics, the University of Edinburgh
Edinburgh, United Kingdom

[3] IRISA-INRIA, Rennes, France

Keywords:
Compiler Optimisations, Iterative Compilation,
Loop Tiling, Loop Unrolling, Embedded Systems

Abstract. In this paper we investigate the feasibility of iterative compilation in program optimisation. This technique enables compilers to deliver efficient code by searching for the best sequence of optimisations. In embedded systems, long compilation time can be afforded since the application is an integral part of the shipped product. However, in practice search spaces may be extremely large. Our experimental results show that in the case of large transformation spaces, near optimal transformations can be found by visiting only a small fraction of the entire search space by using a simple search algorithm.

1 Introduction

Modern compilers make extensive use of optimisation to improve program performance on current micro-processors. The use of a particular optimisation largely depends on static program analysis based on a simplified machine model. Such an optimisation approach has been followed for over 20 years and has produced, in many cases, good results. Due to the unsolvability of the *Halting Problem* [7], however, static analysis is necessarily incomplete and cannot determine the best optimisation for a particular processor/program pair. Furthermore, while the processor and memory hierarchy is typically modelled by static analysis, this does not account for the behaviour of the entire system. For instance, the register allocation policy and strategy for introducing spill code in the back-end of the compiler may have a significant impact on performance. Thus static analysis can improve program performance but is limited by compile-time decidability.

* This research was partially supported by the ESPRIT IV reactive LTR project OCEANS, under contract number 22729.

This paper investigates a different approach to compilation, *iterative compilation*, where successive transformations are applied to a program and their worth determined by actual execution of the resulting code. A large number of different versions of the program are generated and executed, with the fastest version selected. Such an approach is decidable and, given sufficient time, will find the best program. The obvious drawback is that compilation time dramatically increases. For general purpose computing this is not considered feasible. In the case of embedded applications, however, only one program is to be executed and the cost of compilation will be amortised over the number of systems shipped and the lifetime of the application. In such applications, performance is critical and has typically relied on hand coded assembly implementations of the entire application. With the advent of general-purpose processors being used in embedded applications, due to the economies of scale, compiler technology, previously used for general-purpose computing, is now being targeted at embedded processors.

Although iterative compilation is a natural approach for embedded systems, it may become increasingly viable for general-purpose computing as PC system architectures can change almost every three moths. Clearly such a rate of change of compiler technology is hard to sustain and iterative compilation may provide a natural method for programs to adapt to changing hardware.

This paper examines the applicability of iterative compilation to program optimisation. This is achieved by first examining the optimisation space of a set of small programs and then developing an algorithm to search this space efficiently. We show that, although the optimisation space is highly non-linear, iterative compilation can find the best program transformation by examining only a very small fraction of the optimisation space.

The paper is organised as follows. In section 2 we discuss the impact of two compiler optimisations, unrolling and tiling, on program execution time across three different platforms and three benchmark programs with two data sizes. This is followed by section 3 which evaluates an iterative compilation algorithm that attempts to efficiently find good program optimisations. In section 4 we discuss ways in which to reduce the potential complexity of the search. In section 5 we discuss some related work. Finally, in section 6, we draw some conclusions and discuss future research directions.

2 Transformation Space Characteristics

This paper is concerned with studying the viability of iterative compilation by means of 18 case studies. We selected three important and extensively studied kernels and examined their behaviour across three separate platforms and two different data sizes This section is primarily concerned with examining the characteristics of the transformation space and is followed in section 3 by an evaluation of an iterative approach to finding the global minima.

The three kernels and data sizes considered are:

1. Matrix-Matrix Multiplication (MxM) for data sizes $N = 256$ and 300,

2. Matrix-Vector Multiplication (MxV) for $N = 1024$ and 1200,
3. Successive Over Relaxation (SOR) for $N = 512$ and 600.

Each of these programs was executed on three different architectures: MIPS R4000, Pentium II and HP-PA.

For the purpose of this feasibility study, we restrict our attention to a small area of the transformation space formed by applying loop unrolling (with unroll factors from 1 to 20) and loop tiling (with tile sizes from 1 to 100). Each of these 2000 different programs was executed on each of the three platforms and their execution times plotted.

2.1 Example

Figure 1 shows the transformation space of matrix multiplication on the R4000 for $N = 256$ when applying loop unrolling and tiling. The x-axis and y-axis give the tile size and unroll factor respectively, while the z-axis shows the resulting execution time. The goal of an optimising compiler is to find the minimal point in such a space. One immediate observation is that there is approximately a factor of 4 between the maximal and minimal points: selecting the wrong tile size and unroll factor can be critical. The space is highly non-linear, containing many local minima and some discontinuities. Any approach that tries to rely on static analysis to find the global minima would require an analytic model of the space which, due to the non-linearity, would be infeasible. Even if such a space could be accurately modelled it would be extremely difficult to find its minimum due to its complexity.

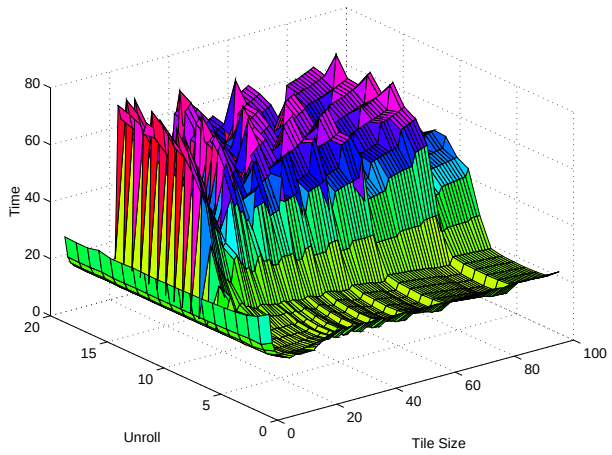
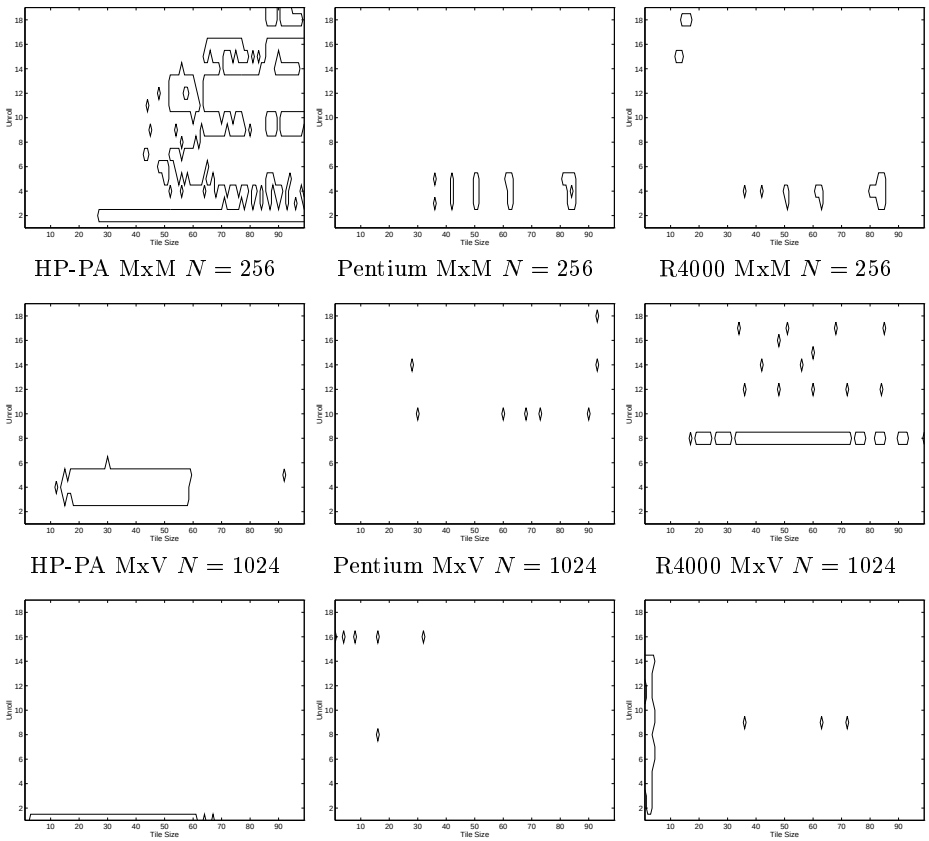


Fig. 1. Performance R4000 for $N = 256$ on MxM

To gain more insight in the characteristics of the transformation space, we focus on those areas of the space that are nearest the minima. In the figures 2 and

3 those areas of the space within 3% of the absolute minimum are highlighted, allowing observations to be made about their distribution. The x and y-axis in each figure corresponds to the unroll factor and the tile-size, respectively.



HP-PA MxM $N = 256$ | Pentium MxM $N = 256$ | R4000 MxM $N = 256$

HP-PA MxV $N = 1024$ | Pentium MxV $N = 1024$ | R4000 MxV $N = 1024$

HP-PA SOR $N = 512$ | Pentium SOR $N = 512$ | R4000 SOR $N = 512$

Fig. 2. Space within 3% of minimum

2.2 Observations

Across figures 2 and 3 there is a wide variety of behaviour. In the case of matrix multiplication on the HP, there is a large part of the space near the minimum for tile sizes greater than 20. The Pentium II and R4000, however, have a similar behaviour quite distinct from that of the HP. Here the minima occur around a small unroll factor with a clustering of minima depending on tile size occuring on the Pentium II. In the case of matrix vector multiplication, we see a very different behaviour. This time the minima on the HP cluster around a small unroll factor

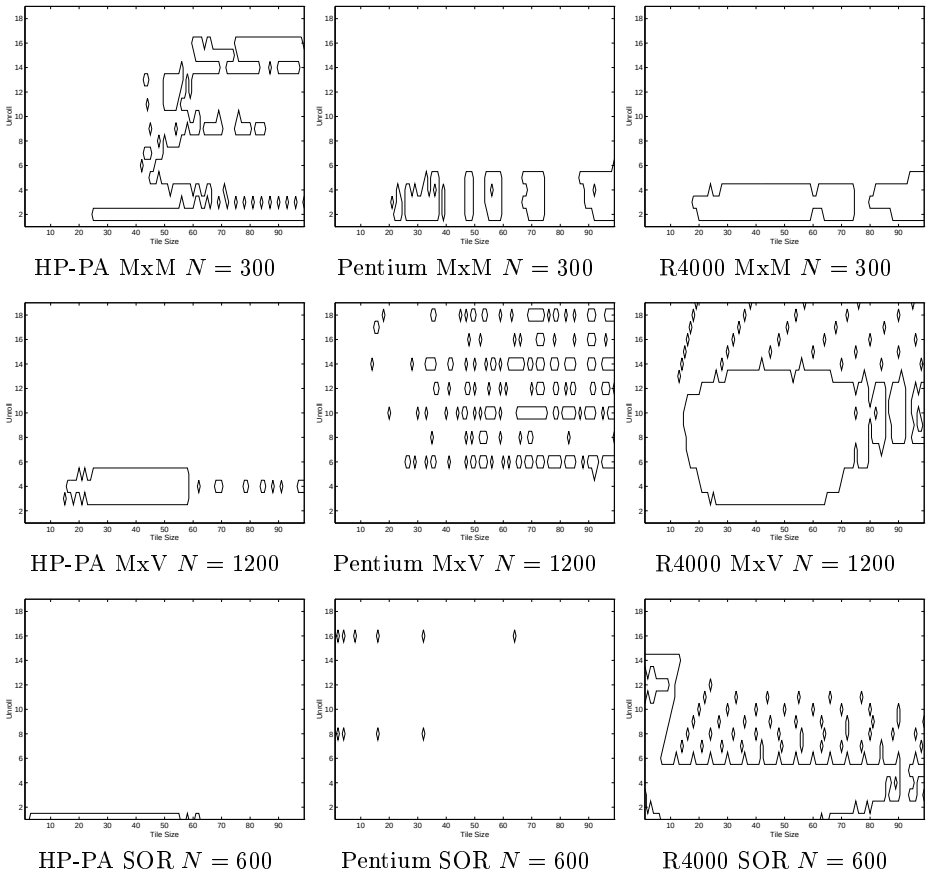


Fig. 3. Space within 3% of minimum

while the Pentium II has a very scattered set of minima whose number is highly dependent on data size. The behaviour of the R4000 is also highly dependent on data size, with the majority of near minimal points occurring around a unroll factor of 8 for small data sizes and with a large area of minima occurring for the larger data size. We can also observe lines of minima spreading out, converging at the origin. For the third program, SOR, we again see wide variation, with the best points on the HP occurring for a very small unroll factor and independent of tile size up to 50. The Pentium II, however, has no points in this region, with 16 being the preferred unroll factor in the sparsely populated graph. Finally, the R4000 has two completely distinct behaviours depending on the data size.

From the above results, we can conclude that the best transformations for a particular program are highly dependent on the underlying architectures. Thus, whenever a system is upgraded, any compiler optimisation technique based on static analysis may have to be completely rewritten. Furthermore, the behaviour of a particular processor across programs is highly dependent on the program

structure. Each of the three kernels considered consist of 2 or 3 perfectly nested loops with access to 2-d arrays. Although similar in structure, the prefered transformations differ widely.

If static techniques are to find the local minima, they need to model program/processor interaction. Such a processor model would be very close to a cycle level accurate simulator. Given the difficulty of statically finding the minima, the next sections considers the use of iterative compilation to search through the transformation space in order to find the best combination of transformations.

3 Iterative Compilation

This section evaluates an iterative approach to compiler optimisation. A compiler algorithm is presented which searches for the best transformation, by sampling the transformation space, and measuring execution times. This is equivalent to searching for the minima in non-linear spaces described in the previous section. Although this approach could potentially be prohibitively expensive, we show that good performance can be achieved by evaluating only a very small percentage of the transformation space.

3.1 Search Algorithm

After prototyping several search algorithms it was quickly determined that a grid based approach consistently outperformed other approaches. It defines a coarse grain grid on the transformation space and refines this grid around good candidate points using a priority based queue.

The algorithm to search the space can be briefly described as follows.

1. First, define a coarse grid on the search space. Initial experimentation [5] indicates that five points in each dimension gives a good initial grid.
2. Evaluate all points on this grid by generating the transformed programs and executing them.
3. Find the point with the current minimum execution time and all current points that are within an allowable distance from this minimum (10%, say).
4. Order these points in a priority queue.
5. For each point in the queue
   - if the execution time associated with this point is within an allowable distance from the minimum found so far, refine the grid around this point by forming a new grid with half the spacing in each dimension.
   - If new points are found that are close to the minimum found so far, enqueue them in the priority queue.

Using this algorithm the search space is traversed to find the best optimisation.

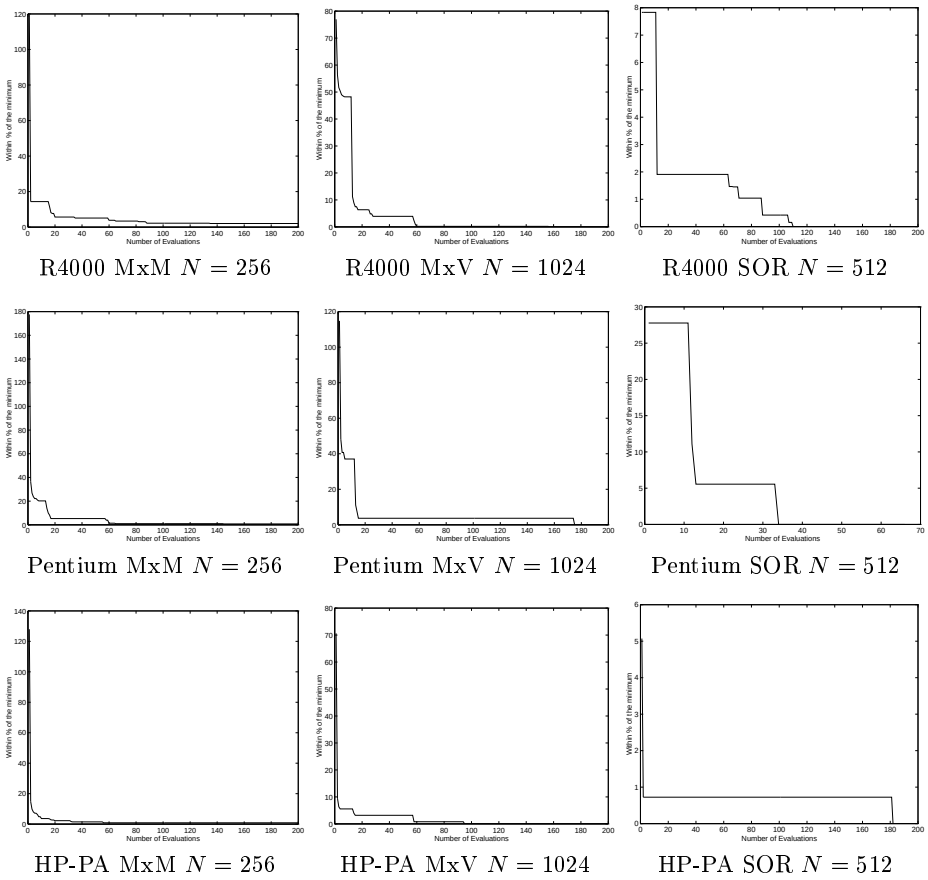


Fig. 4. Performance search algorithm

3.2 Number of Evaluations

The iterative algorithm described above was applied to each program for two different sizes on each of the three processors. We are interested in how quickly such an approach can find a good optimisation. We therefore measured how close the execution time of the transformed program comes to the real minimum, for different number of evaluations. The results are shown in figures 4 and 5. In these figures, the x-axis corresponds to the number of evaluations carried out by our search algorithm, and the y-axis corresponds to how close to the absolute minimum we get. Since our search algorithm keeps track of the best version found so far, these figures show monotonously decreasing graphs.

We see that for more than half of the experiments, the search algorithm does indeed find the absolute minimum within 200 evaluations (10% of the entire search space). In the other cases, the search algorithm comes close to this minimum but does not reach it.

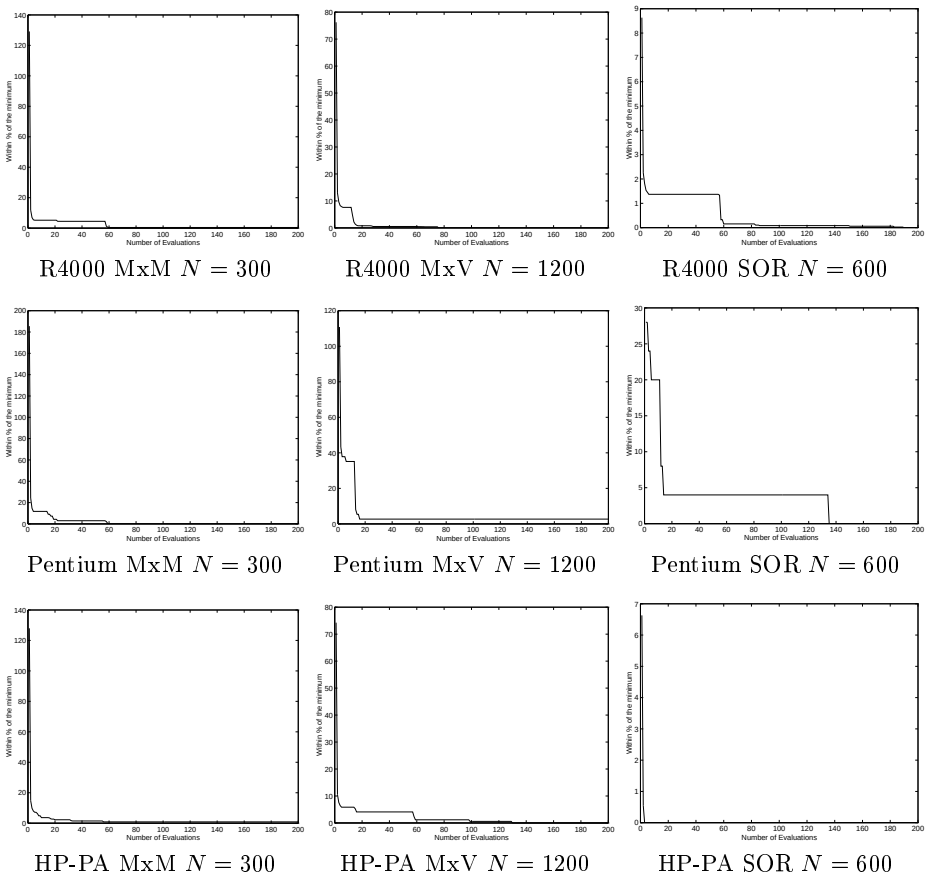


Fig. 5. Performance search algorithm

3.3 Results

First of all, we see that across all three platforms and benchmarks, the iterative algorithm approaches the absolute minimum rapidly. This is illustrated in figure 6 where the average over all the graphs shown in figures 4 and 5 is shown as a function of the number of evaluations carried out by the search algorithm. We see that for less than 15 evaluations, the improvement tends to level out somewhat, but drops again sharply after that. The standard deviation in this average has about the same magnitude. This indicates that there is some variance in the improvement. However, it also indicates that the standard deviation becomes smaller rapidly and that for every benchmark we approach the minimum quite fast. This is very encouraging, demonstrating that at least for the examples selected, iterative compilation is not prohibitively expensive.

In table 1 we report the speedup over the original program found by our search algorithm after 25, 50, 75 and 100 evaluations of a transformed program.

Optimisation seems to have the greatest effect on the Pentium II processor across all programs and on matrix multiplication across all processors. It seems to have the smallest impact on SOR. Such a conclusion could have been drawn from a static analysis of the programs: tiling is more likely to exploit temporal locality within matrix-multiplication than in the case of SOR. We also see that the maximum speedup to be gained for SOR is small: 1.05 on average. If we can deduce by static analysis that a program fragment is likely not to gain much by the transformations under consideration, this may used by an iterative algorithm in order that it concentrates its efforts elsewhere.

Finally, we see that after just 25 evaluations, which corresponds to 1.25% of the entire transformation space, the improvement is close to the maximum. This holds for all benchmarks, data sizes and platforms. Hence we conclude that it may be possible to find good optimisations in a relatively small number of steps.

		25 ev.	50 ev.	75 ev.	100 ev.	max
HP-PA	MxM (256)	1.93	1.96	1.96	1.96	1.98
	MxV (1024)	1.44	1.44	1.47	1.49	1.49
	SOR (512)	1.05	1.05	1.05	1.05	1.06
HP-PA	MxM (300)	1.89	1.89	1.89	1.89	1.92
	MxV (1200)	1.47	1.47	1.51	1.52	1.53
	SOR (600)	1.05	1.05	1.05	1.05	1.05
Pentium	MxM (256)	2.22	2.22	2.31	2.31	2.34
	MxV (1024)	2.21	2.21	2.21	2.21	2.30
	SOR (512)	1.63	1.73	1.73	1.73	1.73
Pentium	MxM (300)	2.01	2.01	2.07	2.07	2.07
	MxV (1200)	1.67	1.67	1.67	1.67	1.67
	SOR (600)	1.23	1.23	1.23	1.23	1.28
R4000	MxM (256)	2.01	2.02	2.06	2.07	2.13
	MxV (1024)	2.05	2.10	2.18	2.18	2.18
	SOR (512)	1.35	1.35	1.36	1.37	1.37
R4000	MxM (300)	1.82	1.82	1.90	1.90	1.90
	MxV (1200)	2.23	2.23	2.24	2.24	2.24
	SOR (600)	1.44	1.44	1.46	1.46	1.46

Table 1. Speedup found

4 Managing Search Space Complexity

Although our results provide some evidence for the feasibility of searching for the best program optimisation, in general the search space under consideration is huge. We have shown that a simple bound on the number of evaluations needed can produce encouraging results by visiting a relatively small fraction of the

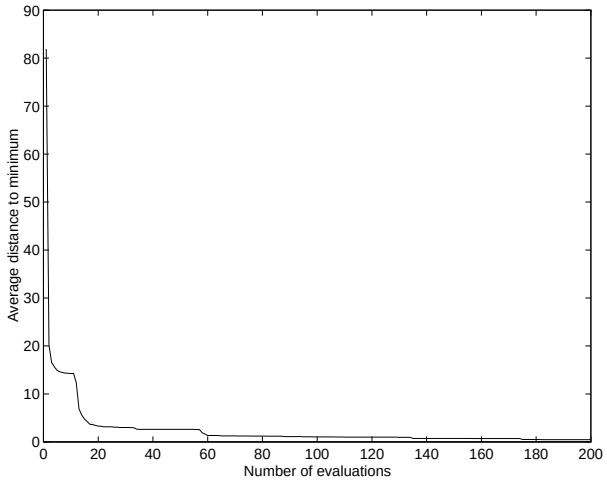


Fig. 6. Average distance to minimum (%)

entire space. However, even this low percentage may be too large for real applications. Therefore, future work will focus on reducing the number of points to consider. Rather than considering static analysis and iterative compilation as distinct approaches, we could consider them as two extreme points on a continuum. A possibly fruitful approach would be to use a simple cost model as a means to generate potentially interesting transformation points.

In this paper we have considered execution time as the metric for evaluating goodness of a transformation. As our system [1, 2] provides additional information, such as code size, register pressure, slot utilisation etc., it is possible to statically evaluate the goodness of a transformation after code generation. Although only approximate, as cache effects etc. cannot be exactly determined, such information may be used to prune transformed programs guaranteed to perform poorly. This paper has also concentrated exclusively on the effect of temporal performance. However, in embedded systems, code size is also important as it determines the amount of ROM required [4]. In order to incorporate code size in the present approach, we would need to have a cost metric which is a function of both execution time and code size. This cost metric can the be used to give a value to the points in the search space. For example, if the code size would be larger than a given maximum, the resulting cost could be set to infinity.

5 Related Work

There is a large body of work considering program transformations to improve uniprocessor performance. In [6], an analytic algorithm to give a good tile size to minimise interference and exploit locality is presented. This work considered

rectangular tiles whose dimensions are a function of the iteration space and the cache organisation. This work gives good performance improvements over existing techniques but does not consider the impact of tiling on unrolling or other transformations.

Whaley and Dongarra [8], and Bilmes et al. [3] describe a system for generation highly optimised versions of BLAS routines. These systems can probe the underlying hardware to find optimal values for blocking factors, unroll factors etc. In contrast to the present approach, these systems are only able to optimise BLAS routines and are not general purpose compilers. Experimentation with these systems [8, 3] has shown that these systems are capable of producing code that is more efficient than the vendor supplied, hand optimised library BLAS routines.

Wolf, Maydan and Chen [9] have described a compiler that also searches for the optimal optimisation. This compiler also considers the entire optimisation space and tries to find the best point in it. In contrast to the present approach, however, their compiler uses a fixed order of the transformations and a static cost model to evaluate the different optimisations. They also use an aggressive but heuristic pruning algorithm to control the complexity of the search. They report good efficiency of the resulting code and short running times of the search. We believe that the present approach that is based on actual execution times instead of static cost models will deliver superior performance.

Bodin et al. [4] describe a method for searching for the best optimisation on the assembly level, taking into consideration both execution times and code size. Their approach also uses a static cost model, in contrast to the present approach, and does not seem to prune the search space.

6 Conclusions

In this paper we have addressed the problem of finding the best optimisation for a given processor, program and data size. We have shown, by describing the actual transformation space, that such an optimal optimisation is hard to find using static analysis. We have shown that an iterative compilation approach based on a simple search algorithm may be able to find a good optimisation by visiting a relatively small fraction of the entire optimisation space. However, for real applications, the search spaces that need to be considered are extremely large. Hence aggressive pruning strategies need to be developed. Future work will be focussed on this issue. In particular, we will investigate whether static analysis and static processor/cost models can be used to guide the search. We will also investigate whether results from mathematical optimisation theory, such as simulated annealing, can be applied in the present case. Nevertheless, the resulting compilation times will be substantial. Therefore, the optimisation approach described in this paper is intended to be used for (kernels of) embedded applications. In this case, long compilation times can be afforded since highly efficient code is essential and compilation time can be amortised over a large number of shipped products.

References

1. B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, P.M.W. Knijnenburg, M.F.P O'Boyle, E. Rohou, R. Sakellariou, H. Schepers, A. Seznec, E.A. Stöhr, M. Verhoeven, and H.A.G. Wijshoff. OCEANS: Optimizing compilers for embedded applications. In *Proc. Euro-Par 97*, volume 1300 of *Lecture Notes in Computer Science*, pages 1351–1356, 1997.
2. M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, P.M.W. Knijnenburg, M.F.P. O'Boyle, E. Rohou, R. Sakellariou, A. Seznec, E.A. Stöhr, M. Treffers, and H.A.G. Wijshoff. OCEANS: Optimizing compilers for embedded applications. In *Proc. Euro-Par 98*, volume 1470 of *Lecture Notes in Computer Science*, pages 1123–1130, 1998.
3. J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proc. ICS'97*, pages 340–347, 1997.
4. F. Bodin, Z. Chamski, C. Eisenbeis, E. Rohou, and A. Seznec. GCDS: A compiler strategy for trading code size against performance in embedded applications. Technical Report 1153, IRISA, Rennes, 1997.
5. F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998. Organised in conjuction with PACT'98.
6. S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *Proc. Programming Language Design and Implementation*, 1995.
7. A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Mathematical Society*, 2(42):230–265, 1936.
8. R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of Alliance 98*, Illinois, US, April 1998.
9. 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.