Iterative Compilation in a Non-Linear Optimisation Space*

F. Bodin[1] T. Kisuki[2] P.M.W. Knijnenburg[2] M.F.P. O'Boyle[3] E. Rohou[1]

[1] IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, France
[2] Dept. of Computer Science, Leiden University, The Netherlands
[3] Division of Informatics, The University, Edinburgh

Abstract

This paper investigates the applicability of iterative search techniques in program optimisation. Iterative compilation is usually considered too expensive for general purpose computing but is applicable to embedded applications where the cost is easily amortised over the number of embedded systems produced. This paper presents a case study, where an iterative search algorithm is used to investigate a non-linear transformation space and find the fastest execution time within a fixed number of evaluations. By using execution time as feedback, it searches a large but restricted transformation space and shows performance improvement over existing approaches. We show that in the case of large transformation spaces, we can achieve within 0.3% of the best possible time by visiting less then 0.25% of the space using a simple algorithm and find the minimum after visiting less than 1% of the space.

1. Introduction

The use of transformations to improve program performance has been extensively studied for over 30 years. Such work is based primarily on static analysis, possibly with some profile information to determine the significant regions of the code [8] and runtime dependent control-flow. Each technique is characterised by trying (i) to determine how a program would perform on a particular processor and (ii) developing a program transformation such that the code is likely to execute more efficiently. Such an approach relies on modeling those features of the architecture and the program that are considered important. Although there has been improvement, much work remains because it is extremely difficult to accurately model program/machine interaction. The problem is made worse in that the number of transformations to apply is potentially infinite.

This paper examines another approach to this problem by first describing the problem as that of searching a non-linear optimisation space in order to find the minimum. Analytic techniques typically consider one parameter in the optimisation space at a time, e.g., tile size, data layout etc. In practice, however, program transformations are not independent in their effect on performance. Furthermore, the optimisation space is highly non-linear (see figure 1) with many local minima. In order to search this space, we propose a technique based on iterative compilation: Different transformations are applied, corresponding to points in the transformation space, and evaluated by executing the program. In order that such an approach is feasible, we need to minimise the number of points evaluated. We show that in the case of large transformation spaces, we can achieve within 0.3% of the best possible time by visiting less then 0.25% of the space using a simple algorithm and find the minimum after visiting less than 1% of the space.

Although such an approach requires very long compilation time since it now includes several runs of the program, it is applicable in those instances where the same application is to be executed many times. Embedded systems are an good example of this and in the following section we briefly describe a compiler framework developed to optimise multimedia codes for embedded systems.

This paper describes a case study to see if iterative compilation can be worthwhile. A simple and well studied problem, matrix multiplication, is selected and executed on four different processors for two different data sizes. This is followed by the description of a simple search algorithm that tries to find the best performance within the fewest number of evaluations. This is repeated for a larger transformation space. The algorithm is applied to the TriMedia TM1000 simulator (a VLIW processor produced by Philips aimed at the embedded processor market [2]) and its behaviour evaluated. The paper finishes with a brief survey of related work and some concluding remarks.

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

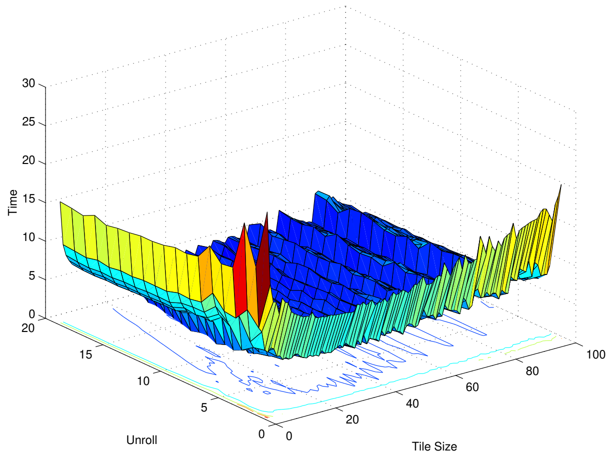


Figure 1. Transformation Space for UltraSparc

2. Iterative Compilation

The reason that program optimisation is difficult is that we are trying to minimise a function (execution time), which is undecidable at compile time, over the infinite space of transformations. One solution is to actually evaluate the program at certain points in this space iteratively. Iterative compilation is generally not considered a viable proposition due the size of the space to be searched and hence the excessive compilation time. This is not the case for embedded systems where there is just one application to be optimised and long compilation times are acceptable if they increase performance. As a guide executing MPEG-2 for a reasonable number of frames takes less than 30 seconds on a TriMedia TM1000. It typically takes an application programmer 3 months to produce an efficient implementation in which time a quarter of a million versions could be evaluated and even more if there is more than one platform.

Analytic techniques have produced good results but are limited to the number of parameters they can consider. The size of the transformation space must be limited and assumptions about the low-level compiler made. Furthermore, if a part of the system changes that is not implicitly or explicit modeled, such as the register allocation policy of the local compiler, then analytic approaches are unable to adapt. Other factors generally ignored, include the introduction of spill code and instruction cache misses. Iterative compilation by definition consider all parts of the system when deciding on the best optimisation.

The Oceans project is an ESPRIT funded project, concerned with developing an iterative compiler for embedded systems [1]. In particular, we are targeting general purpose VLIW processors of which the Philips TriMedia TM1000 [2] is typical. Economies of scale allow the production of cheaper and faster general purpose processors over custom embedded processors. However, such processors rely on efficient software implementations of the embedded applications but can afford long compilation times, hence our interest in iterative compilation. It is the long term goal of this project to successfully integrate static analysis and feedback information for embedded application performance.

3. Problem Description

In this paper, matrix multiplication is selected as the case study to optimise because (i) it is well known, allowing independent comparison with other techniques; (ii) it forms the core of the fdct in MPEG-2, and (iii) there are several legal transformations that can be applied. Furthermore, if we can show improvement for this extremely well-studied problem, then it is likely that further improvement will be possible in those programs that have received less attention.

The parameters of our experiment are as follows:

- Problem Size: $N = 400$ and $N = 512$
- Processor Type: UltraSparc, R10000, Pentium Pro, Alpha and TriMedia TM1000
- Transformation Space: Loop Unrolling 1 to 20, Tiling 1 to 100, Padding 1 to 10

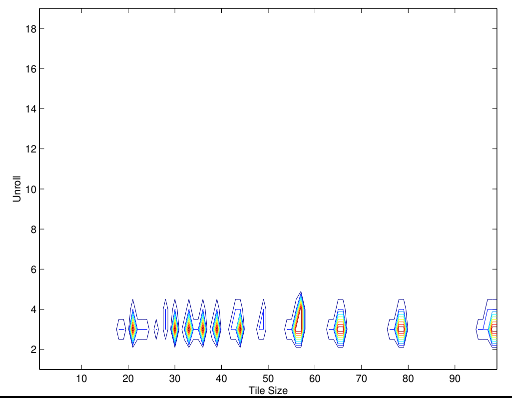
We are interested in the correct combination of optimisations that minimises execution time on each processor for each data size. In this work we only consider high level optimisations and the only feedback information used is execution time.

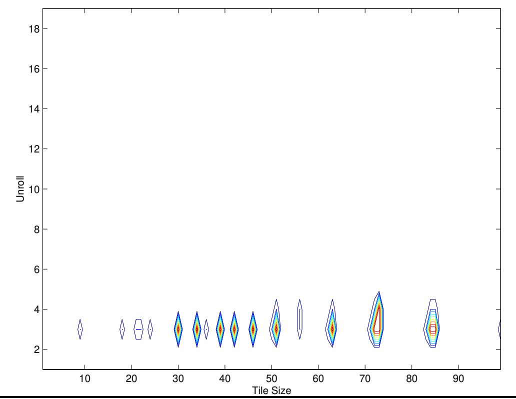
In the following sections, the properties of the optimisation space are described and the impact of our algorithm in finding minima. Initially, we consider just tiling and unrolling for existing commodity processors. We then extend the experiments to consider padding and then apply the algorithm to the TriMedia TM1000 simulator.

4. Tiling and Unrolling

Figure 1 shows the execution time of matrix multiplication on the UltraSparc for problem size $N = 512$ and for varying tile sizes and unroll factors. It is periodic with high frequency oscillation and many local minima. Within such a space it is difficult to find the absolute minimum. The graphs in figures 2 to 5 show the areas of the transformation space such that that transformed program has an execution time within 20% of the absolute minimum for the four different

Ultra Sparc $N = 400$	Ultra Sparc $N = 512$





min = 0.563 max = 10.498 orig = 4.99216	min = 1.541 max = 22.114 orig = 11.5321
avg = 1.142 Near Min: 2.6%	avg = 4.072 Near Min: 1.5%
Minimum at: Unroll = 3, Tile = 57	Minimum at: Unroll = 3, Tile = 73

Figure 2. Transformation Space Characteristics of UltraSparc

commodity processors and two different data sizes. The x-axis denotes tile size and the y-axis denotes unroll factor. The original[1], minimum, maximum and average execution time for the space is also shown as is the number of points near 20% of the minimum execution time. What is immediately apparent is that the best transformation depends largely on the processor and to a lesser extent on the data size. The UltraSparc and Alpha perform best with a small unroll factor but the tile size varies. The Pentium has a more dispersed range of minima while the R10000 has the largest percentage of points closest to the minimum. Such a characteristic should increase the probability of finding a good result within a reasonable number of samples. By way of contrast, the number of minimal points is much smaller for the Alpha and there is a much wider range of performance values. Across the various examples, optimisation gives an improvement of between a factor of 1.8 and 10 over unoptimised code.

5. Iterative Strategy

As can be seen from the the graphs in figures 2 to 5, there are several local minima per application and the minima varies from one data size/processor to the next. Clearly any algorithm that wishes to search the space must be robust enough so as not to be trapped in a local minimum. Therefore, techniques based on gradient approaches are not applicable. This must be balanced against searching too much of the transformation space, especially those regions where no suitable candidates can be found. This problem is compounded by the occurrence of minimal points surrounded by large values. It is not the purpose of this paper to propose the best search algorithm for such spaces, instead we are interested in the applicability of such algorithms.

Our search algorithm visits a number of points at spaced intervals, applying the appropriate transformation, executing the transformed program and evaluating its worth by measuring the execution time. Those points lying between the current global minimum and the average are added to an ordered queue. Iteratively, such points are removed from the queue and points within the neighbouring region are investigated, again at spaced intervals. This process is continued until a specific number of points has been evaluated whereupon the point with minimal value, i.e., the fastest transformed program, is reported.

5.1. Step Size vs. Iteration Count

The step size within each of the transformation dimensions is a key component of the search algorithm. The step size that gives the biggest improvement depends not only on the program/data size/processor but also on the total number of evaluations to be undertaken. For instance, a particular step size may be best if we are considering just 20 evaluations, but if we increase this to 200, another step size may be preferable. Despite this relationship, it was found that an initial step size of five samples in any one dimension gave a reasonable performance regardless of the number of evaluations undertaken or processor/data size/transformations considered.

[1] All programs including transformed ones were compiled with -O2 optimisation

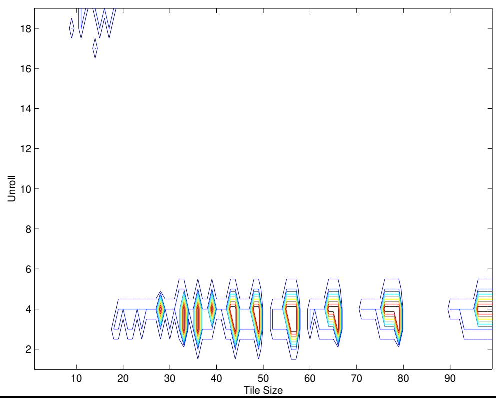
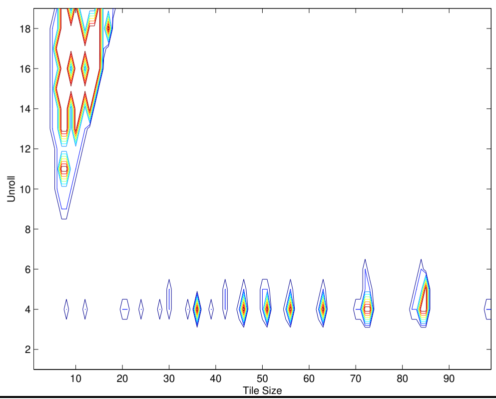
R10000 $N = 400$	R10000 $N = 512$
	
min = 0.408 max = 3.403 orig = 0.712106	min = 1.088 max = 7.602 orig = 2.792315
avg = 0.778 Near Min: 7.8%	avg = 2.019 Near Min: 7.2%
Minimum at: Unroll = 4, Tile = 57	Minimum at: Unroll = 4, Tile = 85

Figure 3. Transformation Space Characteristics of R10000

5.2. Performance

Figures 6 and 7 gives the performance results of 4 of the 8 examples. The remaining 4 have similar performance. The x-axis of each graph is the number of evaluations carried out for each application of the iterative algorithm. The y-axis shows the percentage difference from the absolute minimum. For instance, in the first graph after just one evaluation, the best performance found is 165% more than the actual minimum. That is, it is 2.65 times slower than the actual minimum. In the case of the Alpha it is 5.25 times slower. In each case the search algorithm finds the minimum in less than 200 steps or 10% of the search space. More importantly, within 20 steps or 1% of the space, it is within 23.75% of the minimum. There is a rapid improvement in performance up to about 20 steps, with a more gradual improvement later. Despite the difficult characteristics of the transformation space, a relatively straightforward optimisation algorithm can find a very good implementation with a relatively few number of evaluations.

6. Padding

In order to further investigate the worth of iterative compilation, we extend the search space to include array padding in the first dimension of each array up to a pad size of 10, increasing the possible number of transformations to consider to 20 000. Clearly, in practice, it is unrealistic to evaluate exhaustively the transformation space, but in order to evaluate the performance of an iterative algorithm, we generated all possible transformed programs for two processors, UltraSparc and Alpha, for data size $N = 512$. The evaluation space is 4 dimensional and cannot be easily be presented graphically. Instead, certain slices of the space are shown in figures 8 and 9. In these figures, the areas of the transformation space that yield programs with an execution time within 20% of the minimum execution time are depicted.

Without padding, the best transformations for the UltraSparc are for the case when unroll factor is 3, as can be seen in figure 2. When padding is also considered, however, none of the points within 20% of the minimum have an unroll factor of 3. This demonstrates the close interaction of transformations and the error introduced when considering them separately. In fact, the majority of minimal points occur for an unroll factor of 4, as shown in figure 8. Without padding, the best tile size was found to be 73 (see figure 2) and in figure 8 the points near minimum for this tile size are shown when padding is also considered. In fact, the best performance found is when the tile size is 51, unrolling factor is 4 and padding size is 8. The execution time for this compound transformation is now 1.2269, a 20% improvement in execution time when compared to just unrolling and tiling. Solving one or more transformations independently will not give the best overall combination of transformations. In the case of the Alpha processor, shown in figure 9, unrolling by a factor of 4 still gives the best improvement, but now with a tile size of 73 instead of 85 (see figure 5) and padding of size 8. As in the case of the UltraSparc, the improvement is an approximate 20% reduction in execution time compared to the case where no padding was employed.

Although the number of points in the space has increased by a factor of 10 to 20 000 points, the number of points needed to be evaluated has not grown proportionally. In fact, within

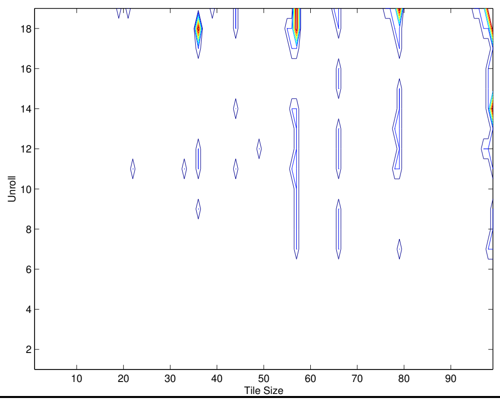
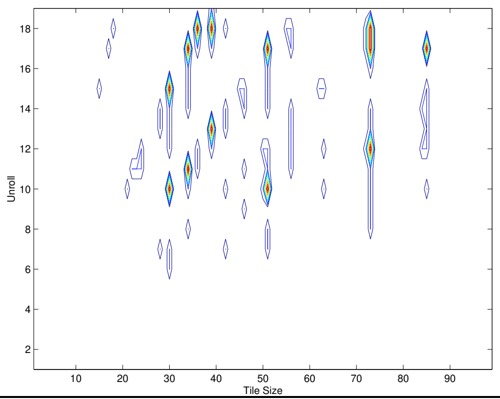
Pentium Pro $N = 400$	Pentium Pro $N = 512$
	
min = 1.433 max = 12.719 orig = 4.88	min = 3.340 max = 26.770 orig = 30.93
avg = 2.425 Near Min: 4.3%	avg = 5.875 Near Min: 4.6%
Minimum at: Unroll = 19, Tile = 57	Minimum at: Unroll = 17, Tile = 51

Figure 4. Transformation Space Characteristics of Pentium Pro

50 evaluations we have found a transformed program while performance is within 0.3% of the global minimum on the UltraSparc. This can be seen in figure 10 where there is a rapid improvement in performance from over 7 times the minimal execution time to within 0.3% of the minimum in 42 steps. Also shown in figure 10 is the same plot with the first 20 iterations removed so as to give more detail on the algorithms performance. A similar behaviour is also seen in figure 11 for the Alpha. It, however, needs just over 82 steps to approach within 2% of the minimal possible execution time, finding the actual minimum within 136 evaluations.

7. TriMedia TM1000

The previous sections have described the transformation space and the performance of a search algorithm for several commodity processors. We are particularly interested in applying such techniques to embedded processors, such as the TriMedia TM1000. In this section we evaluate our iterative approach to optimization by running the transformed programs on a cycle accurate simulator. As we are using a simulator rather than an actual processor, cycle counts are given as the performance measure and smaller data sizes are considered, namely $N = 64$ and $N = 128$.

7.1. Performance

Figure 12 shows those points within 20% of the minimum and their distribution for the case $N = 64$. This minimum was found for unroll factor of 3 and a tile size of 21. The performance of the search algorithm is also given. What is immediately noticeable, is that the number of points needed to sample does not scale down linearly with the size of the transformation space, just as it did not scale up when considering padding (see section 6.).

Although this paper has evaluated all points in the transformation space to give an absolute measure on the performance improvement of the iterative algorithm, in practice no absolute minimum will be available. Rather, the scheme will evaluate points returning the best available as long as sufficient time remains. This is the case with the final experiment where $N = 128$ on the TriMedia TM1000. Due to the excessive simulation time it is not feasible to exhaustively search the space, so no absolute measure of performance is available. Nevertheless, the results in figure 13 show that the iterative algorithm makes steady improvement, reducing the execution time by a factor of 7 over the original program in less than 60 evaluations.

8. Related Work and Discussion

There is a large body of working considering program transformations to improve uniprocessor performance. In [3], 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. For example, in [3] a tile size of 170×2 was shown to give the best performance on the Al-

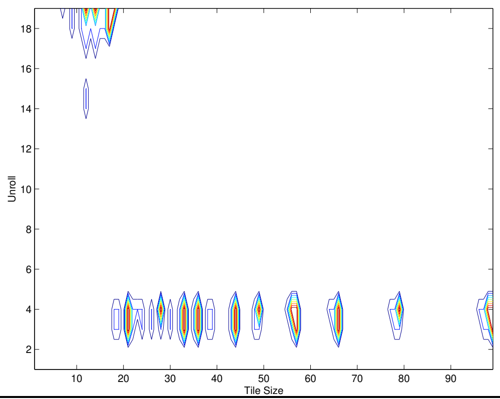
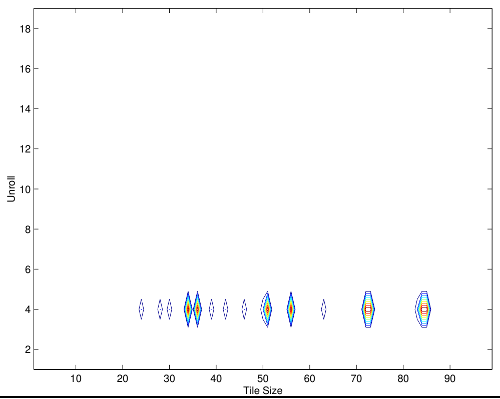
Alpha $N = 400$	Alpha $N = 512$
	
min = 1.237 max = 38.186 orig = 12.041	min = 3.347 max = 81.403 orig = 31.726
avg = 2.854 Near Min: 4.3%	avg = 7.644 Near Min: 0.9%
Minimum at: Unroll = 4, Tile = 44	Minimum at: Unroll = 4, Tile = 85

Figure 5. Transformation Space Characteristics of Alpha

pha for matrix multiplication when $N = 256$. Applying this transformation gives an execution time of 0.794460 seconds. On applying our iterative algorithm, however, we reduce this time to 0.33863 with a tile size of 17, unrolling factor of 18 and array padding of 8.

Static analysis could be used to "seed" the transformation space, that is, give initial points to investigate. As static analysis generally considers just one or two transformations, these points will form search hyper planes with potentially many points to investigate. Further analysis techniques could be used to reduce this number. In this paper we have considered execution time as the metric for evaluating goodness of a transformation. As our system [1] 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.

Several researchers have considered using runtime information to select the best implementation. They, however, define one or more options statically which are then considered at runtime. For example, in [5], whether or not a portion of the iteration space should be tiled depends on runtime characteristics and in [7], different synchronisation algorithms are called depending on runtime behaviour. The work in this paper, however, considers a much larger space of optimisations at compile time without incurring runtime overhead. Later work could combine the approaches by including dynamic monitoring to select, at runtime, one of a number of optimisations programs that were determined (at compile time) to perform well under certain circumstances.

In [6], genetic algorithms are used to create and select transformations for parallel optimisation. This work is similar in spirit to the work presented in this paper but at present generates many illegal programs which must be discarded and hence examines a much larger set of programs before finding any improved solutions.

In this paper we have used a very simple search algorithm as a basis for iterative compilation There is in fact a large literature on non-linear optimisation [4], though it is based on a continuous underlying optimisation function rather than the discrete space we consider. Techniques such as polynomial fitting could be applied to help improve the performance of the search algorithm. Although the best transformations to select are interdependent, knowledge of the processor and application domain could help bias the search space to first consider the areas where the minima is most likely to exist. For instance, in several of the examples, unrolling near a factor of 4 often leads to good results. However, this is not always the case (see Pentium figure 4).

Although the search spaces considered are large, the domain is extremely limited. Only 3 transformations were applied to a simple loop nest. For general programs, the space to consider will be much greater. Future work needs to consider the application of a iterative compilation to large programs. This paper has also focussed on parameterised transformations. However, many transformations are not parameterised: they are either applied or not. For such transformations, search trees might be more appropriate.

While it may be feasible to efficiently search large transformation spaces, it is not necessarily always easy to generate the transformed program. Applying a sequence of program transformations corresponding to the position in the space is

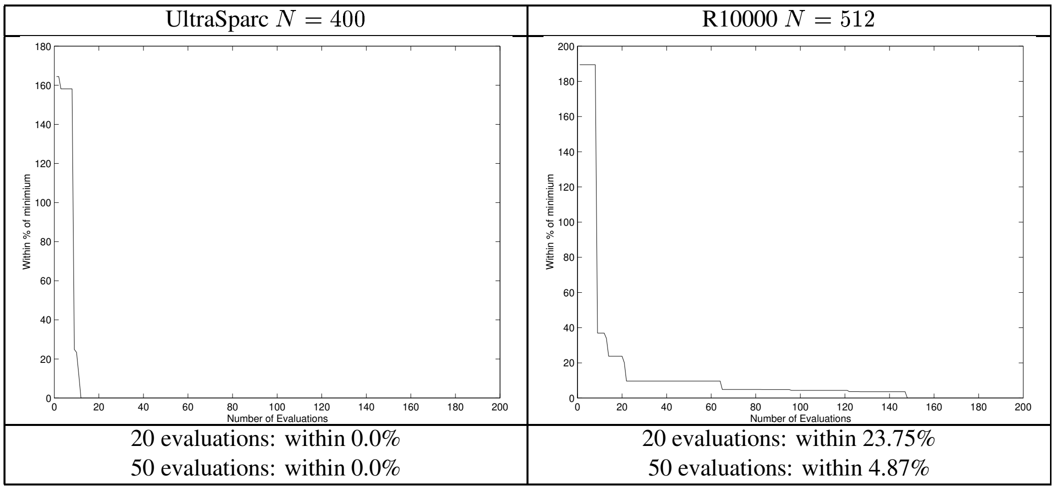


Figure 6. Search Algorithm Performance

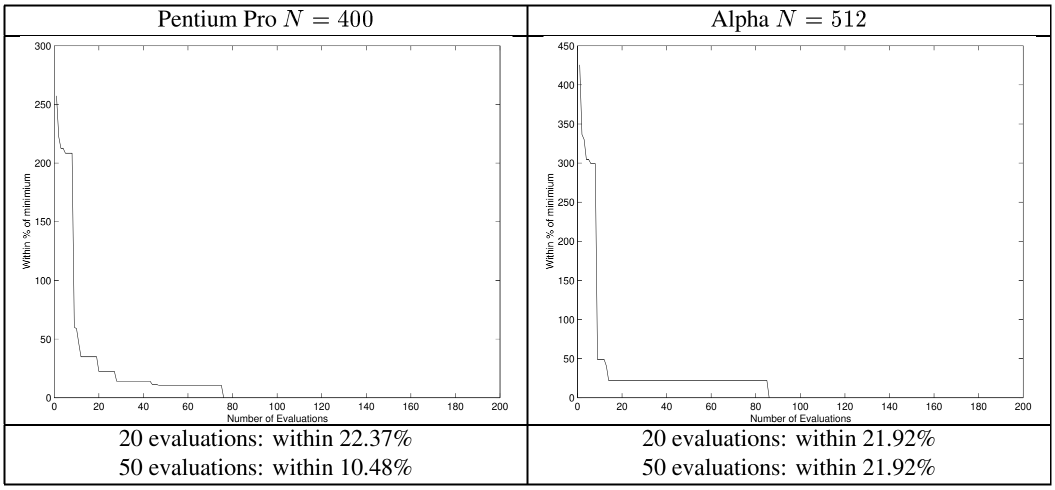


Figure 7. Search Algorithm Performance

limited by the form of code produced by any previous transformation. In future, it will therefore necessary to develop program transformation techniques to allow general iterative compilation consisting of may compound transformations.

This paper has concentrated on the effect of temporal performance. However, in embedded systems, code size is also important as it determines the amount of ROM required [9]. If we have a cost metric which is a function of execution time and code size, the techniques described in this paper can be immediately applied.

Finally, the processor concerned strongly affects the likelyhood of finding a good solution. Those processors which have less non-linear optimisation spaces are likely to be easier to optimise for.

9. Conclusion

This paper has investigated the use of iterative compilation as an program optimisation technique. By considering program optimisation as searching a transformation space for minima, we have shown that a simple search algorithm can achieve good results. We have shown that iterative compilation is a viable program optimisation approach, particularly for embedded systems. Future work will consider larger application programs and transformation spaces. The combination of static and iterative information to guide optimisation will also be considered. Finally, improved search algorithms will be investigated.

Acknowledgment We would like to thank Kyle Gallivan for his for his ideas and initial input to this work.

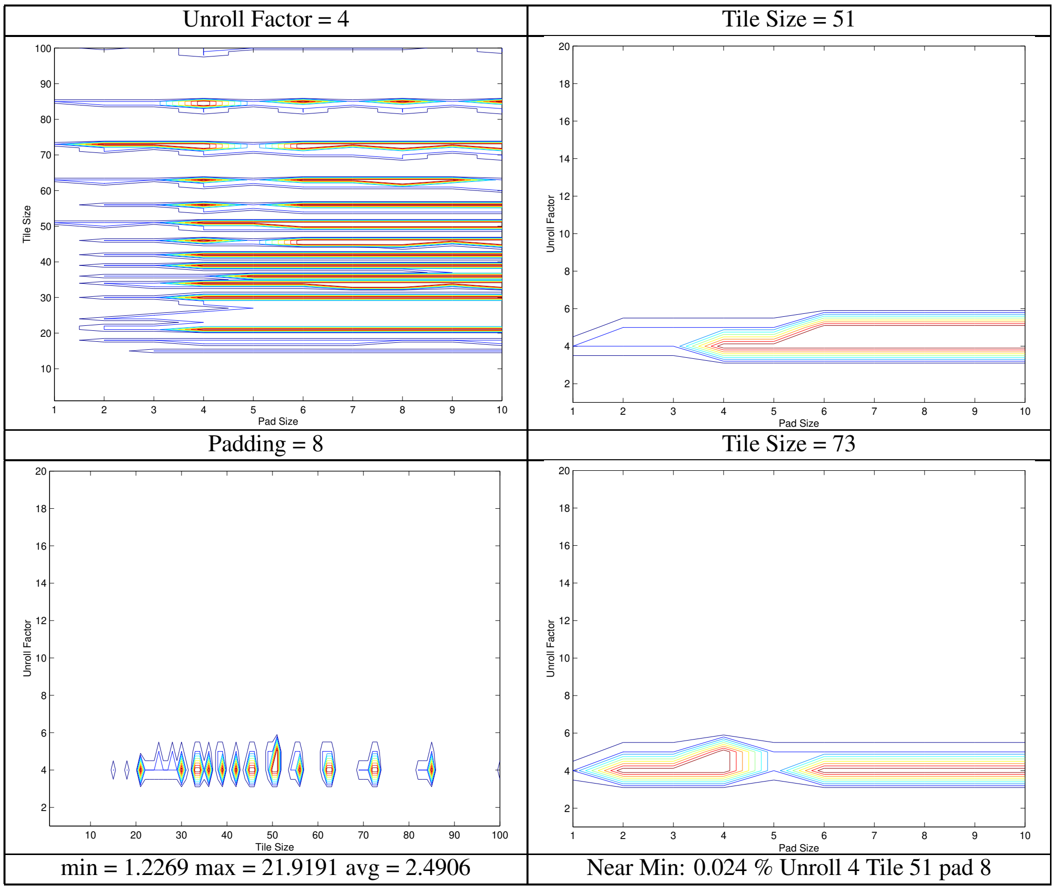


Figure 8. Transformation Space: UltraSparc $N = 512$

References

[1] B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J.R. 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. Stohr, M. Verhoeven, H.A.G. Wijshoff, **OCEANS: Optimizing Compilers for Embedded Applications**. Proc. EuroPar97, 1997.

[2] B. Case, **Philips Hope to Displace DSPs with VLIW**. Microprocessor Report 8(16), 5 Dec. 1994, pp. 12–15. See also http://www.trimedia-philips.com/

[3] S. Coleman and K. McKinley, **Tile Size Selection using Cache Organization and Data Layout**. Proc. Programming Language Design and Implementation, ACM Press, 1995.

[4] E. Hansen, **Global Optimization Using Interval Analysis**, Marcel Dekker Inc. New York, 1992

[5] S.F. Hummel, I. Banicesu, C.-T. Wang and J. Wein, **Load Balancing and Data Locality via Fractiling: An Experimental Study**. Kluwer Academic Press, LCR, 1995.

[6] A. Nisbet, **GAPS: Genetic Algorithm Optimised Parallelisation**. Proc. 7th Workshop on Compilers for Parallel Computing, 1998.

[7] P. Diniz and M. Rinard, **Dynamic Feedback: An Effective Technique for Adaptive Computing**. Proc. Programming Languages Design and Implementation, 1997.

[8] M. Fernadez, **Simple and effecient link-time optimizations of modula-2 programs**. Proc. Programming Languages Design and Implementation, 1995.

[9] F. Bodin, Z. Chamski, C. Eisenbeis, E. Rohou and A. Seznec, **GCDS: A Compiler Strategy for Trading Code Size Against Performance in Embedded Apllications**. IRISA Tech Rep. 1153, 1997.

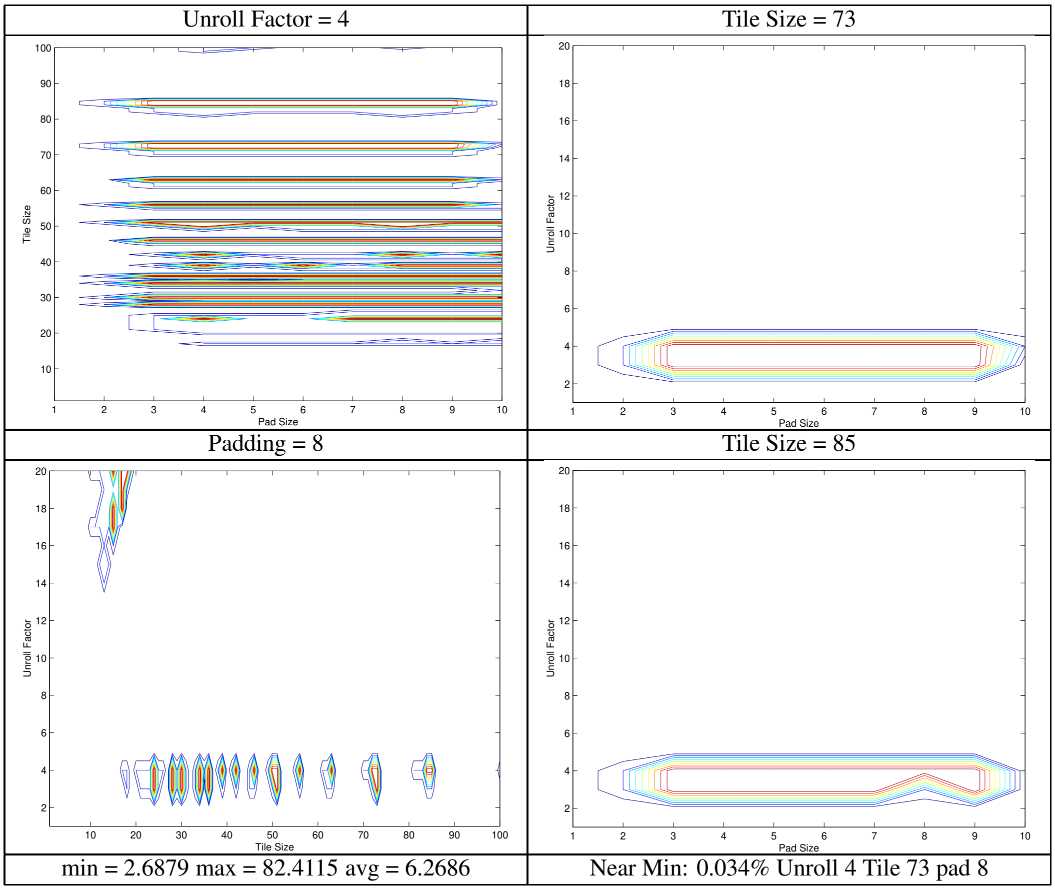


Figure 9. Transformation Space: Alpha $N = 512$

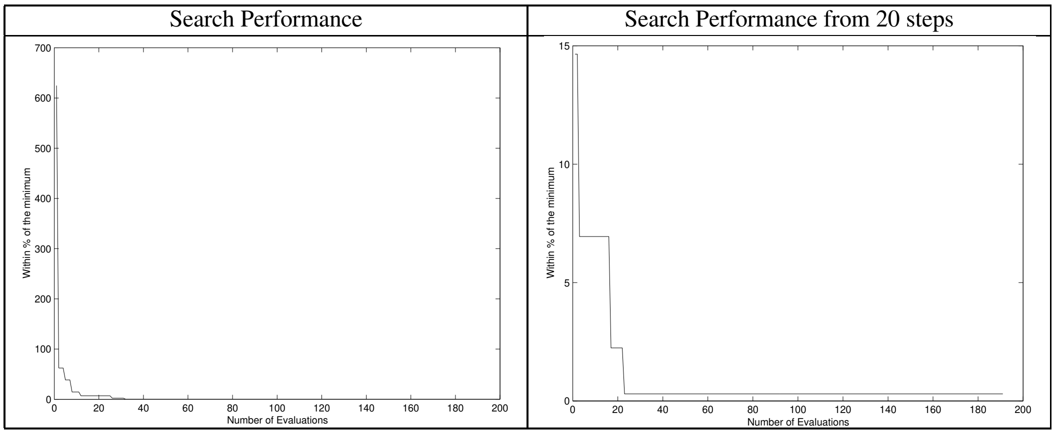
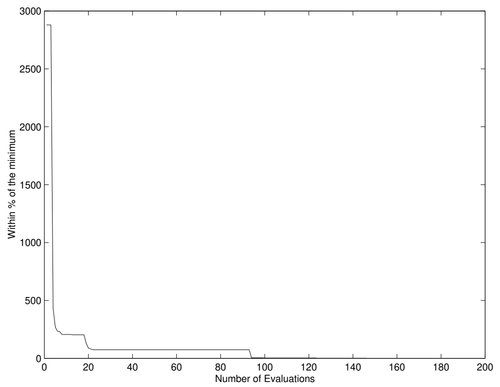


Figure 10. Performance of search algorithm : UltraSparc

Search Performance



Search Performance from 20 steps

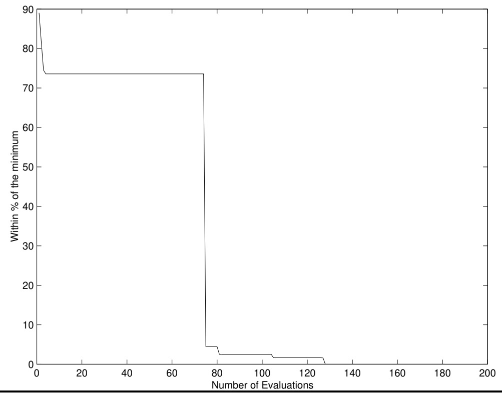
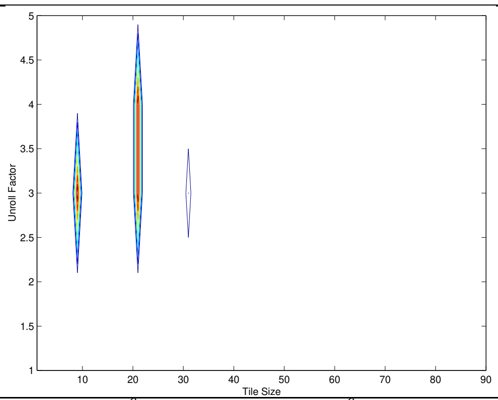


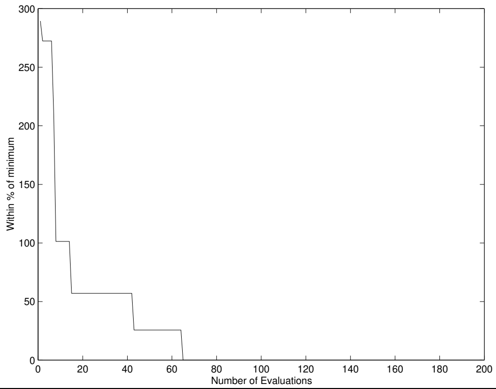
Figure 11. Performance of search algorithm : Alpha

Minima Distribution 64



min = 1.584×10^6 , max = 12.801×10^6,avg = 5.094×10^6

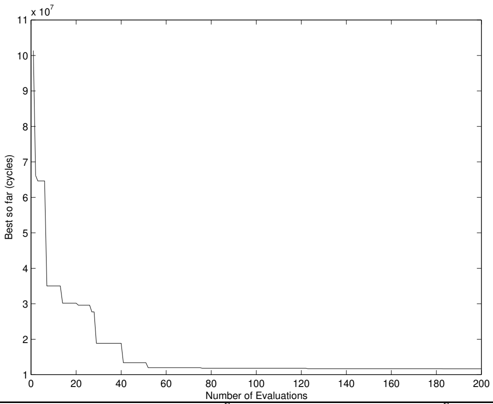
Search Performance 64



orig =6.750×10^6 Near Min: 1.7% Tile 21 Unroll 3

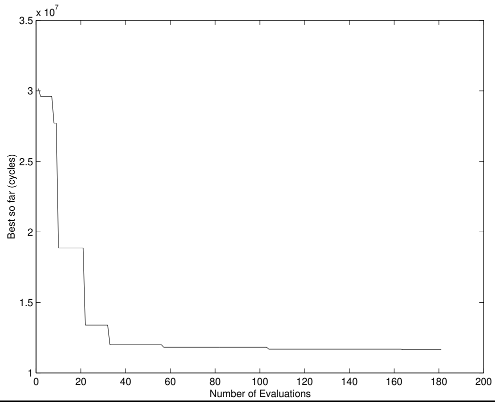
Figure 12. Minimal points and search algorithm performance for TM1000 $N = 64$

Search Performance 128



min =11.664×10^6, max =101.387×10^6

Search Performance from 20 steps



orig = 71.418×10^6, Tile 21 Unroll 4

Figure 13. Minimal points and search algorithm performance for TM1000 $N = 128$