OCEANS

Optimising Compilers for Embedded Applications*

Michel Barreteau[1], François Bodin[2], Zbigniew Chamski[4], Henri-Pierre Charles[1], Christine Eisenbeis[5], John Gurd[6], Jan Hoogerbrugge[4], Ping Hu[5], William Jalby[1], Toru Kisuki[3], Peter M.W. Knijnenburg[3], Paul van der Mark[2,3], Andy Nisbet[6], Michael F.P. O'Boyle[7], Erven Rohou[2], André Seznec[2], Elena A. Stöhr[6], Menno Treffers[4], and Harry A.G. Wijshoff[3]

[1] Laboratoire PRiSM, Université de Versailles, 78035 Versailles, France.
[2] IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, France.
[3] LIACS, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands.
[4] Philips Research, Information and Software Technology, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands.
[5] INRIA, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France.
[6] Department of Computer Science, The University, Manchester M13 9PL, U.K.
[7] Division of Informatics, The University, Edinburgh EH9 3JZ, U.K.

Topic 14
Emerging Topics in Advanced Computing in Europe
Keywords
Optimisation, Iterative compilation, Loop transformations, Embedded systems

Abstract. This paper presents an overview of the activities carried out within the second year of the ESPRIT project OCEANS whose objective is to investigate and develop advanced compiler infrastructure for embedded VLIW processors. This combines high and low-level optimisation approaches within an iterative framework for compilation. In this paper we discuss the approach to iterative compilation adopted in the OCEANS project.

1 Introduction

Embedded applications have become increasingly complex during the last few years. Although sophisticated hardware solutions, such as those exploiting instruction level parallelism, aim to provide improved performance, they also create a burden for application developers. The traditional task of optimising assembly code by hand becomes unrealistic due to the high complexity of hardware/software. Thus the need for sophisticated compiler technology is evident.

Within the OCEANS project, the consortium intends to design and implement an optimising compiler that utilises aggressive analysis techniques and that integrates source-level transformations with low-level, machine dependent

* This research is supported by the ESPRIT IV reactive LTR project OCEANS, under contract No. 22729.

optimisations [7, 8]. A major objective is to provide a prototype framework for iterative compilation, where feedback from the low-level is used to guide the selection of a suitable sequence of source-level transformations. Currently, the Philips TriMedia (TM-1000) VLIW processor [4] is used for validation of the system.

In general, compiler optimizations rely on static analysis, simplified processor and cache models and sometimes profiling information. Static analysis is necessarily a pessimistic approximation of runtime behaviour, and processor/memory hierarchy models only approximately the behaviour of a part of the system. Profile based analysis produces averages of the observed behaviour of the system for a limited number of benchmarks/input sets. Compiler analysis determines the best parameters for each compiler optimisation separately (e.g., tile size). However, optimizations are not independent in their effect. Finally, in the present market hardware is changing rapidly. Therefore, the compiler and its optimisation sequence/stategy need to adapt quickly to hardware changes in order to remain competitive. We conclude that the traditional approach to optimization only gives suboptimal results.

In order to cope with the problems described above, an iterative approach to optimization has been proposed in the OCEANS project. It consists of searching for a good transformation sequence. This means that we need to optimize, compile and execute the program many times. However, for the case of embedded applications, this can be afforded. In [3, 11] we have presented two studies into the characteristics of transformation spaces and the feasibility of searching these spaces. Based on these studies we have concluded that searching for an optimization sequence may be a viable solution to the optimisation problem.

In this paper, we present an overview of the work that has been carried out during the second year of the project. An overall description of the system is given in Section 2. In section 3 we present a study into the effects of different transformations on execution time. In section 4 we discuss approaches to iterative compilation. In section 5 we discuss related work. Finally, in section 6 we present some conclusions and directions of future work.

2 An Overview of the OCEANS Compiler System

The OCEANS [7, 8] compiler is centered around two major components: a high-level restructuring system, MT1, and a low-level system for supporting assembly language transformations and optimisations, SALTO. SALTO is coupled with SEA, a set of classes that provides an abstract view of the assembly code, and tools for software pipelining (PILO) and register allocation (LORA). Their interaction is illustrated in figure 1 which shows the overall organisation of the OCEANS compilation process. In particular, a program is compiled in three main steps:

- First, MT1 performs lexical, syntactical and semantic analysis of a source FORTRAN program (`File.f`). Also, a sequence of source program transformations can be applied. These transformations are written in the Transformation Definition Language and the order of their application is specified using the Strategy Specification Language [8].

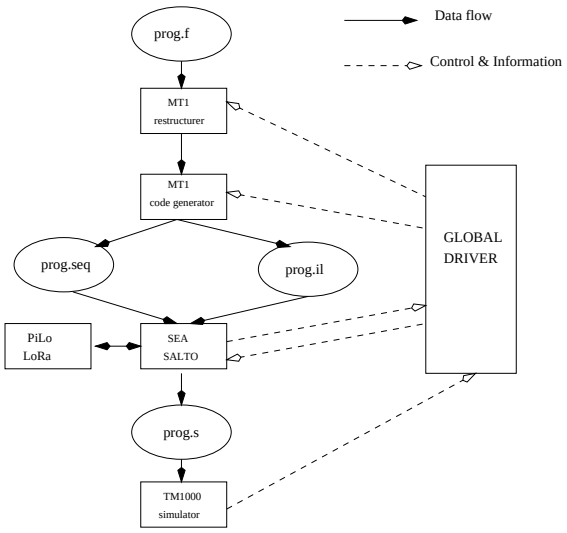


Fig. 1. The Compilation Process.

- The restructured source program is then fed into the code generator which generates sequential assembly code that is annotated with instruction identifiers used to identify common objects in MT1 and SALTO, and a file written in an *Interface Language* (`File.IL`) that provides information on data dependences and program structure.
- Finally, SALTO (coupled with SEA) performs code scheduling and register allocation. At this step guarded instructions are created and resource constraints are taken into account.

The above process is driven by a global driver which select optimisations at the source-level and the low-level iteratively until a certain level of performance is reached. This paper is primarily concerned with the structure of such a driver which, for the purposes of this study, focuses on high-level transformations. In the next section the optimisation space considered is described and a candidate search algorithm for the global driver is evaluated. This is followed by a short description of alternative search strategies currently under evaluation within the OCEANS project.

3 Transformation Space Characteristics

This section is primarily concerned with examining the characteristics of transformation spaces. We initially selected three important and extensively studied kernels and examined their behaviour across seven separate commodity processors and different data sizes. This is followed by a more restricted evaluation of the TriMedia processor.

The initial three kernels and data sizes considered are: *matrix-matrix multiplication (MxM)* for $N = 256$, 300, 400 and 512, *matrix-vector multiplication (MxV)* for $N = 1024$ and 1200, and *Successive Over Relaxation (SOR)* for $N = 512$ and 600. These programs were executed on seven different architectures: MIPS R4000, MIPS R10000, Pentium II, Pentium Pro, Alpha, UltraSparc and HP-PA. We also used the Philips TM-1000 simulator as an example of an embedded processor. In this feasibility study, we restrict our attention to loop unrolling (with unroll factors from 1 to 20) and loop tiling (with tile sizes from 1 to 100). We generated all versions of the programs and executed them on several of the platforms. Although the application of iterative compilation to commodity processors is interesting, we are particularly concerned with applying such techniques to embedded processors such as the TriMedia-1000. In this case, only matrix-multiplication was considered with smaller data sizes of $N = 64$ and $N = 128$ due to the overhead of using a simulator.

3.1 Transformation Spaces

Figure 2 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. The z-axis shows the resulting execution time. The goal of an optimising compiler is to find the minimum 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. Hence if an optimising compiler were to use an inaccurate heuristic, the resulting transformed program may run less efficient than the original program.

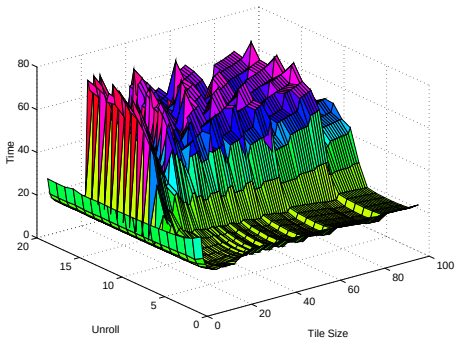


Fig. 2. Performance R4000 for $N = 256$ on MxM

Distribution of Minima for Commodity Processors To gain more insight in the characteristics of the transformation space, we focus on those areas of the space that are close to the absolute minimum. For example, in figures 3 through 8, the areas that are within 3% of the minimum are depicted for matrix-vector

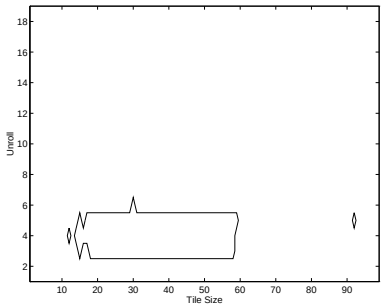


Fig. 3. HP-PA: MxV $N = 1024$

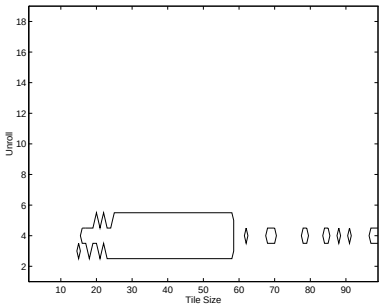


Fig. 4. HP-PA: MxV $N = 1200$

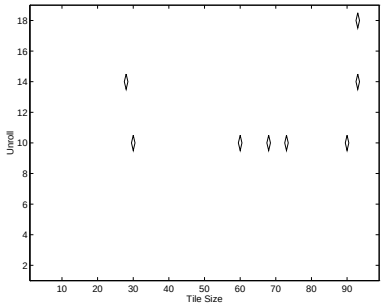


Fig. 5. Pentium II: MxV $N = 1024$

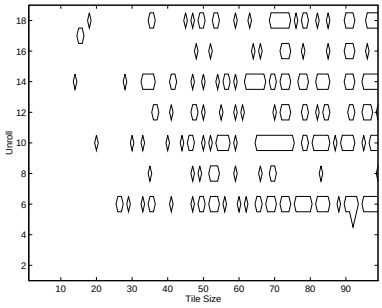


Fig. 6. Pentium II: MxV $N = 1200$

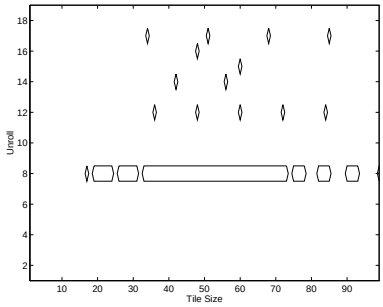


Fig. 7. R4000: MxV $N = 1024$

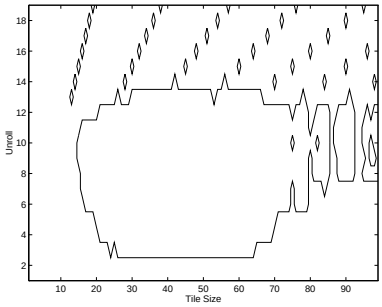


Fig. 8. R4000: MxV $N = 1200$

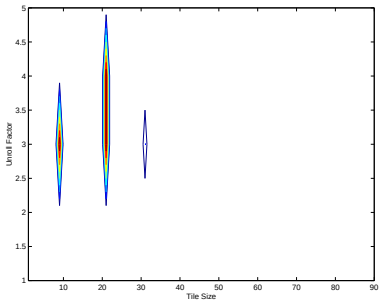


Fig. 9. TM-1000: Minimal points for MxM $N = 64$

multiplication on three commodity processors. For a full discussion consult [3, 11].

Across figures 3 through 8 we observe a wide variety of behaviour. We see that the minima on the HP cluster around a small unroll factor 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.

Distribution of Minima for an Embedded Processor Figure 9 shows those points within 20% of the found minimum and their distribution for the case program matrix-matrix multiplication on the TriMedia for the data size $N = 64$. The original untransformed code takes 6.750×10^6 cycles to execute as compared to an average of 5.094×10^6 cycles across the transformation space. In other words, the original program is 32% slower than the application of a random transformation on average. However, there is great variance across the space and the maximum execution time is 12.801×10^6 cycles. Thus, a wrong transformation selection can more than double the execution time of the original program. The actual minimum execution time is 1.584×10^6 cycles, which is over 4 times faster than the original. However, only 1.7% of the entire space is within 20% of this value.

3.2 Conclusion

From the results in the above sections we can conclude that the best transformations for a particular program are highly dependent on the underlying architecture, data sizes and program structure. If static techniques are to find the local minima, they need to model program/processor interaction extremely closely. Such a model would be very close to a cycle level accurate simulator. Given the difficulty of statically finding the minima, the next section considers the use of iterative compilation to search through the transformation space in order to find the best combination of transformations.

4 Iterative Compilation

To deal with the problems discussed in the previous section, the OCEANS project is concerned with searching the transformation space for the best optimization. In this section we discuss how search techniques are applied at the high level and the low level. We also discuss a genetic algorithm approach.

4.1 High level searching

Our compiler algorithm, presented below, searches for the best transformation, by sampling the transformation space and measuring execution times. 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. The search algorithm is simple but suffices for studying the feasibility of searching: without exploiting any system or application characteristics we can already produce good results.

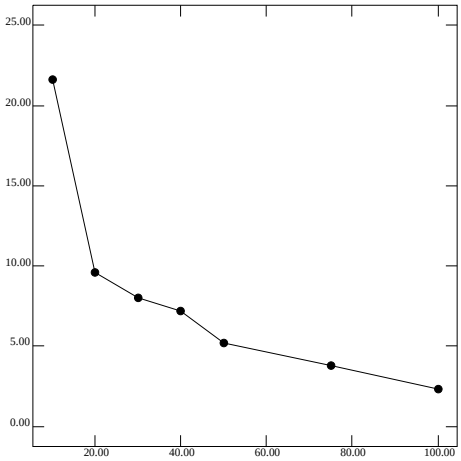
Search Algorithm The algorithm used in the feasibility study is grid based. It can be briefly described as follows.

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 (10%, say). Order these points in a priority queue.
4. 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.

Results In [3, 11] it is shown that across all platforms and benchmarks, the iterative algorithm approaches the absolute minimum rapidly. See figure 11 for an example graph for the TriMedia, in which the number of evaluations is plotted against the relative distance to the absolute minimum. For the commodity processors analogous graphs have been obtained [3, 11]. In figure 10, we have given the average percentage of how close to the absolute minimum the search algorithm comes across all platforms, benchmarks and data sizes. The average is taken over 26 measurements. The x-axis shows the number of evaluations and the y-axis shows the distance to the minimum. The figure shows a monotonic decreasing graph that reaches high levels of optimization rapidly. In the table below the graph, we have also shown between brackets the standard deviation, that is, the average distance to the mean. This standard deviation shows that there is some variance across all measurements, but this variance is low enough to conclude that we reach good levels of optimization rapidly.

TriMedia-1000 As we are particularly interested in applying such techniques to embedded processors, the performance of the search algorithm on the TM-1000 is shown in figures 11 and 12. In the case of $N = 64$, we have the values of all points in the transformation space and can determine the relative difference between the best transformation found so far against the known minimum. The original point selected is more than 4 times slower, but after 20 steps is about 1.5 times slower than the absolute minimum. Within 70 evaluations the best possible transformation sequence is found.

Although this paper has evaluated all points in the transformation space for the commodity processors and the case of $N = 64$ for the TM-1000, in practice 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



Number of evaluations						
10	20	30	40	50	75	100
21.6 (16.5)	9.6 (11.2)	8.0 (9.6)	7.2 (7.5)	5.2 (5.2)	3.8 (4.4)	2.3 (2.5)

Fig. 10. Average percentage difference minimum

TriMedia-1000. Due to long simulation times, it was not feasible to exhaustively search the space, so no absolute measure of performance is available. Nevertheless, the results in figure 12 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.

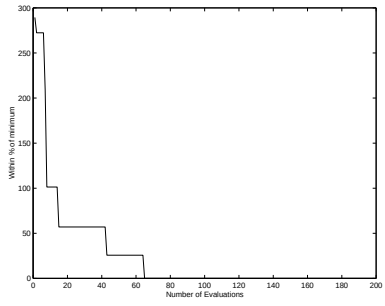


Fig. 11. TM-1000: MxM $N = 64$

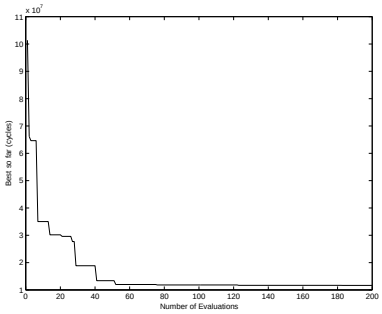


Fig. 12. TM-1000: MxM $N = 128$

4.2 Alternative Search Techniques

Although we have focused on one search base technique for iterative compilation, other approaches are also being investigated within the project.

Code size–performance trade-off Another search algorithm examined by the OCEANS project is a depth-first tree search, where sets of transformations are recursively built up and examined. The global driver decides if a set is worthwhile for further examination by enlarging the set, or it backtracks by shrinking the set. Main focus with this search is on

- Feedback between the different modules. We exploit two kinds of feedback:
  Static Feedback: consisting of the static number of cycles for an iteration,
  Dynamic Feedback: consisting of the dynamic number of cycles and data cache behaviour obtained by executing the code.
- Trade-off between code size and code performance.

One of the key issues for iterative compilation is to provide a useful feedback from the different components of the compiler, so decisions can be made by choosing between various sets of transformations. In this section we consider loop unrolling. Since unrolling has mostly an impact on scheduling, static feedback is sufficient. However, in some cases cache behaviour needs to be known and dynamic feedback is used. To illustrate the use of the feedback information by the global optimising driver (shown in figure 1), we briefly describe the various steps of the compilation.

1. Extract source code characteristics.
2. Get a reference point: the original code is compiled without any transformations to get the basic static performance of the loop.
3. Start transformation space exploration using the information stored.
4. Use dynamic and static feedback for decisions about a next point in the optimisation space.

An additional constraint of compilers for embedded applications compared to traditional compilers is that code size is important. Larger code usually means a larger die size and thus increased production costs. We therefore not only search for the best optimisation concerning code performance but try to find a trade-off between these two aspects, using a cost model which takes dynamic and static feedback into account. Traditional optimising compilers are based on a fixed set of heuristics and only optimise for speed or code size, but don't search for a trade-off. Figure 13 shows how complex this issue can be. This figure is obtained by our implementation of the search algorithm. It shows a range of unrolling factors applied to a loop. It should be noted that when a software pipelining algorithm is applied to the unrolled loop, the code size can grow with a factor of 5. Relative gain gives the relative gain between to successive unroll factors.

Genetic algorithms As an alternative to traditional search techniques, we are also investigating the application of genetic algorithms (GA) as a means of determining the best transformation sequence. This has the potential benefit of investigating transformation spaces which cannot easily be described as a cartesian domain and is extremely robust in the presence of local minima.

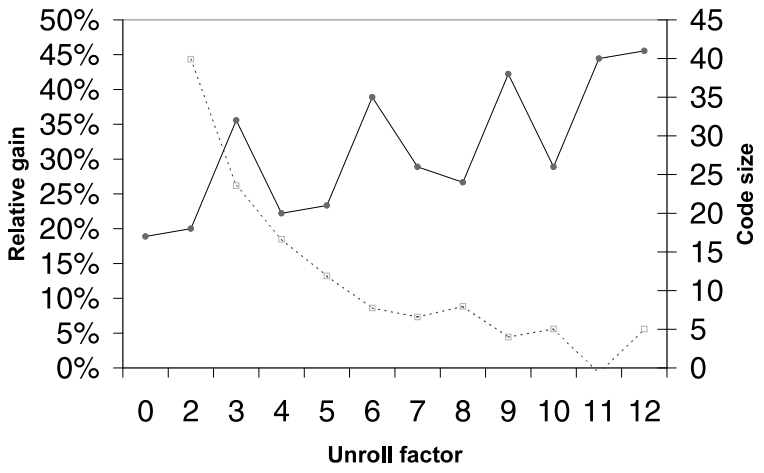


Fig. 13. Code size and performance

The OCEANS GA search is implemented as part of the GAPS compiler framework described in [12] which uses GA based optimisation for an auto-parallelising compiler. In the OCEANS GA, traditional restructuring transformation sequences such as tiling, loop-permutation, loop-distribution, loop-fusion, loop-skewing and statement reordering are represented as multi-dimensional mappings [10]. GA optimisation initialises a population of mappings using a combination of randomised methods and conventional compiler techniques. Thus, a population represents a subset of the transformation space for a program. Mappings representing transformed programs having good performance (i.e., low execution time/predicted overheads) are given high reproduction selection probabilities. Mutation and recombination based reproduction operators generate new *child* mappings from randomly selected *parent* mappings currently in the population. Steady-state reproduction with an elitist replacement strategy ensures that child mappings only replace mappings associated with programs having low performance. Reproduction is iteratively applied until a maximum number of mappings have been created or until a real-time performance constraint is satisfied. The use of elitism in conjunction with conventional compiler techniques ensures that the performance of the best solution produced by GA optimisation will be equal to or greater than that produced by the conventional techniques.

5 Related Work

There is a large body of work considering program transformations to improve uniprocessor performance. In [5], an analytic algorithm to give a good tile size to minimise interference and exploit locality is presented. 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 [13], and Bilmes et al. [1] describe a system for generation highly optimised versions of BLAS routines by probing the underlying hardware to find optimal values for blocking factors, unroll factors etc. Experimentation [1, 13] 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 [14] have described a compiler that also searches for the best optimisation. This compiler also considers the entire optimisation space. 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. The present approach, however, is based on actual execution times instead of static cost models.

Bodin et al. [2] 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.

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 [9], whether or not a portion of the iteration space should be tiled depends on runtime characteristics and in [6], 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.

6 Conclusions and Open Problems

In this paper we have described the activities within the second year of the Esprit project OCEANS. 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 also 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 focused 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 required for these systems. Compilation time can be amortised over a large number of shipped products.

References

1. 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.
2. 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.
3. 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.
4. B. Case. Philips' hope to displace DSP with VLIW. *Microprocessor Report*, 8(16):12–15, 1995. See also http://www.trimedia-philips.com/.
5. S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *Proc. Programming Language Design and Implementation*, 1995.
6. Pedro Diniz and Martin Rinard. Dynamic Feedback: An Effective Technique for Adaptive Computing. *Programming Languages Design and Implementation, ACM Press*, pages 71–84, 1997.
7. B. Aarts et al. OCEANS: Optimizing compilers for embedded applications. In *Proc. Euro-Par 97, LNCS 1300*, pages 1351–1356, 1997.
8. M. Barreteau et al. OCEANS: Optimizing compilers for embedded applications. In *Proc. Euro-Par 98, LNCS 1470*, pages 1123–1130, 1998.
9. S.F. Hummel, I. Banicesu, C.-T. Wang, and J. Wein. Load balancing and data locality via fractiling: An experimental study. In *Proc. 3th Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 85–98. Kluwer Academic Publishers, 1996.
10. W. A. Kelly. *Optimization within a Unified Transformation Framework.* PhD thesis, Univ. of Maryland, 1996.
11. T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, F. Bodin, and H.A.G. Wijshoff. A feasibility study in iterative compilation. In *Proc. ISHPC'99*, 1999.
12. A. Nisbet. GAPS: Genetic algorithm optimised parallelization. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998. Workshop organised in conjunction with PACT'98.
13. R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of Alliance 98*, Illinois, US, April 1998. Available through `http://www.netlib.org/atlas/`.
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.