Overview of the Esprit IV Reactive LTR Project OCEANS Optimizing Compilers for Embedded Applications

Peter M.W. Knijnenburg

Leiden Institute of Advanced Computer Science, Leiden University,
Niels Bohrweg 1, 2333 CA Leiden
The Netherlands
peterk@liacs.nl

Abstract

This paper presents an overview of the activities carried out within the ESPRIT project OCEANS whose objective is to investigate and develop advanced compiler infrastructure for embedded VLIW processors. This combines high and low-level optimization approaches within an iterative framework for compilation.

1 Introduction

Increasingly, general-purpose processors are used for embedded applications rather than customised hardware. As processor cost drops, it becomes more attractive to use one processor for several applications rather than designing specific hardware. Multimedia based applications are typical of the growing uses of embedded systems, requiring cost-effective implementation and high performance. *Very Long Instruction Word* (VLIW) processors are an attractive solution for such applications as they provide potentially high performance, due to multiple parallel functional units, and are relatively cheap to manufacture due to the simple processor architecture. However, sophisticated optimizing compiler technology is necessary to exploit the fine-grain parallelism as assembly programming of complex applications is not feasible.

With current compiler technology, the average number of operations per cycle in VLIW processors is only 2 to 2.5 [2].

Moreover, 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 optimizing assembly code by hand becomes unrealistic due to the high complexity of hardware/software. Thus the need for sophisticated compiler technology is evident.

The goal of the OCEANS project [1, 4, 5] is to investigate and develop state-of-the-art compilation techniques to allow high performance implementations of embedded applications. In such applications, long compilation times can be afforded as each embedded processor will usually execute a limited number of applications throughout its lifetime. This makes it feasible to use more aggressive compilation techniques than previously considered. Within the OCEANS project, the compiler utilises aggressive analysis techniques and integrates source-level transformations with low-level, machine dependent optimizations. 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 [11] 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 optimization 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 optimization 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 [10, 15] 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 optimization problem.

In this paper, we present an overview of the OCEANS project. First, the objectives of the

project are stated in section 2. An overall description of the system is given in Section 3. In section 4 we present a study into the effects of different transformations on execution time. In section 5 we discuss approaches to iterative compilation. In section 6 we discuss some related work. Finally, in section 7 we present some conclusions and directions of future work.

2 Project Objectives

The initial objectives of the OCEANS project are the following.

High-Level Optimizations Objectives: We aim to develop high-level restructuring transformations for the exploitation of VLIW processors. These transformations are primarily designed to enable successful later low-level exploitation of fine-grain parallelism. The strategy, or sequence of transformations, employed is guided by feedback from other stages.

Low-Level Optimizations Objectives: We intend to develop low-level restructuring techniques, concentrating on a highly retargetable object code scheduler that includes optimizing techniques suited for embedded applications and VLIW architectures. This is achieved through a multifunction testbed tool which can manipulate assembler code in order to implement low-level code restructuring as well as to provide the high-level code restructurer with information collected from the assembler code and from instruction profiling.

Integration Objectives: We intend to integrate the above into a prototype system based on iterative compilation, where a close interaction between the high and low-level exists, allowing better exploitation of available information. The main back-end target for this project is the Philips TriMedia (TM-1) processor [11]. The objective is to show that this approach yields more efficient code for this particular processor, eventually as optimal as hand-optimized code, while cutting down the code development time considerably.

An overall aim of this project is to achieve high retargetability of the code optimization process. The reason for this is that the cost of the development of compilers for embedded architectures must be amortized across variations of hardware implementations using the same instruction set architecture.

In this document we discuss how these objectives have been implemented during the project.

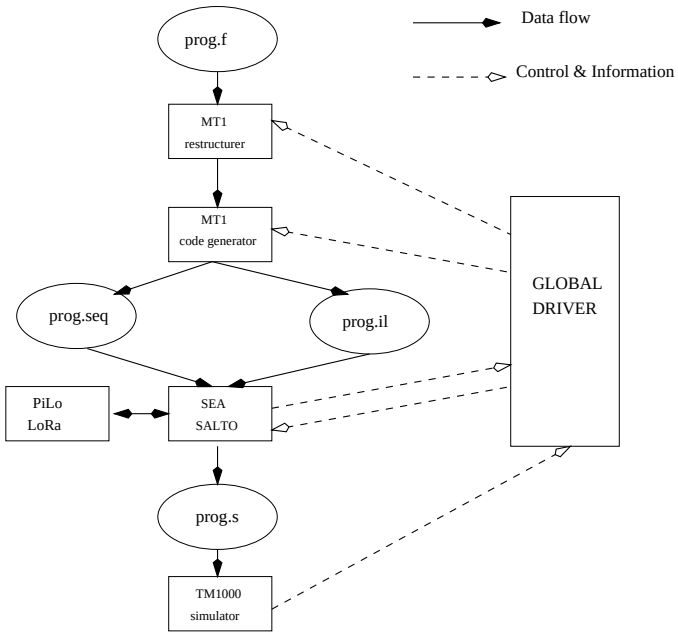


Figure 1: The Compilation Process.

3 The OCEANS Compiler System

The OCEANS [1, 4, 5] compiler is centered around two major components: a high-level restructuring system, MT1, and a low-level system for supporting assembly language transformations and optimizations, 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 [7] 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 (TDL) [6] and the order of their application is specified using the Strategy Specification Language (SSL) [3].
- The restructured source program is then fed into the code generator which generates sequential assembly code that is annotated with instruction identifiers used to iden-

tify 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. SALTO connects to the tools PILO and LORA to perform software pipelining and register allocation.

The above process is driven by a global driver which select optimizations at the source-level and the low-level iteratively until a certain level of performance is reached.

MT1 is developed and maintained at Leiden University, whereas SALTO is developed and maintained in INRIA, Rennes. The two components are connected together over the Internet via a socket interface. Next we discuss the software components in more detail.

The MT1 Restructuring Compiler: Over the past few years, a full Fortran 77 compiler, called MT1, has been developed at Leiden University [7]. An important aspect of the MT1 compiler is that it provides a facility for specifying program transformations that can be applied interactively by means of a Transfromation Definition Language (TDL) [6]. These are defined by an input pattern, an output pattern and a condition under which the transformation can be applied. Input patterns may contain meta-variables that are bound to program expressions or statements. These meta-variables may be used in specifying the output pattern and the condition. Moreover, functions that act directly on the internal representation may be defined and may be used in the output pattern and the condition. Conditions typically check for the existence of dependences between certain parts of the input pattern. Furhtermore, MT1 contains a Strategy Specification Language (SSL) [3] that allows the user to specify the order in which transformations are to be applied.

SALTO: A Retargetable System for Assembly Language Transformation and Optimization: Salto [17] is a retargetable framework for developing a whole spectrum of tools that manipulate assembly language programs. The objective of the system is to provide the user with a single environment that facilitates the implementation of performance tuning tools for low-level codes. This set of tools includes assembly code schedulers, profiling, and tracing tools. Salto is retargetable with respect to instruction sets and hardware details.

Salto consists of three parts; a kernel, a machine description file and an optimization or instrumentation algorithm. The kernel performs the parsing of the assembly code and of the machine description file, and the construction of the internal representation. The internal

representation is then available via the user interface. The machine description file provides a model of hardware configuration and the complete description of the instruction set, including per-instruction resource reservation tables. The optimization or instrumentation algorithm is supplied by the user, via a user-supplied function `Salto_hook`.

The user interface of Salto is object-oriented and provides classes to represent a complete description of the control-flow graph of the program and a model of the target architecture.

PiLo and LoRa: PiLo and LoRa are packages for software pipelining and loop register allocation developed at INRIA Rocquencourt. PiLo has one heuristic mode based on the decomposed software pipelining algorithm [18], as well as one exact mode for code scheduling under register constraints based on an integer programming formulation. LoRa is a package that optimally allocates the loop variables into registers while controlling loop unrolling when necessary [13]. PiLo and Lora are connected to Salto via an interface describing architectural and dependency constraints among instructions.

4 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.

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.

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 optimizing 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 optimizing compiler were to use

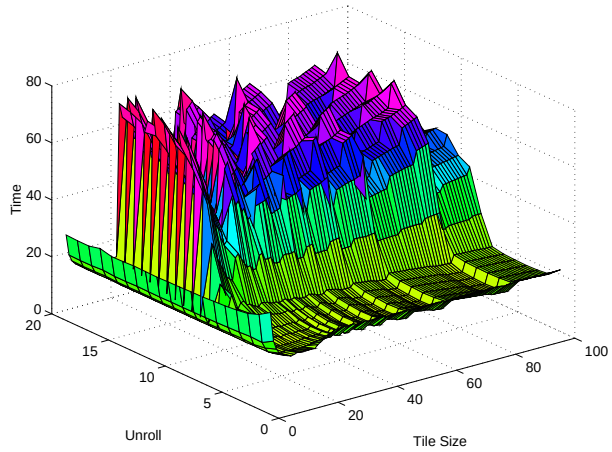


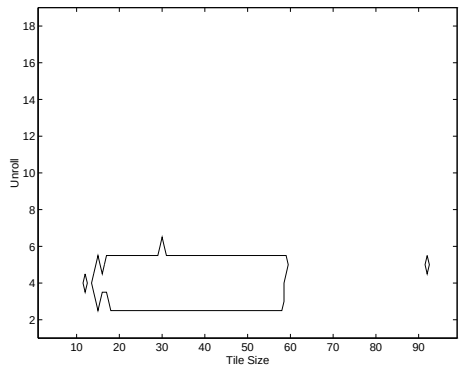
Figure 2: Performance R4000 for $N = 256$ on MxM

an inaccurate heuristic, the resulting transformed program may run less efficient than the original program.

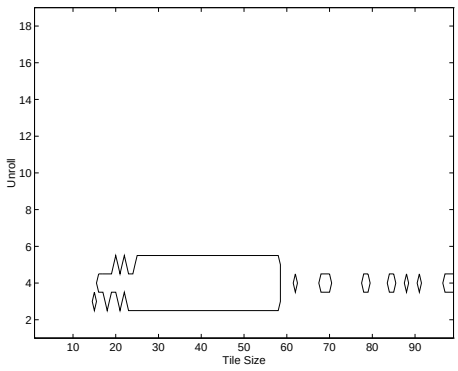
To gain more insight in the characteristics of the transformation space, we focus on those areas of the space that are close to the absolute minimum. For example, in figure 3, the areas that are within 3% of the minimum are depicted for matrix-vector multiplication on three commodity processors. For a full discussion consult [10, 15].

Across the figures 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.

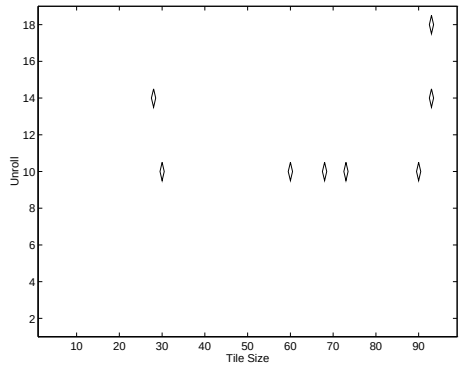
From the results discussed in [10, 15] we can conclude that the best transformations for a particular program are highly dependent on the underlying architecture, data sizes and program structure. If static techniques are to find the local minima, they need to model program/processor interaction extremely 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.



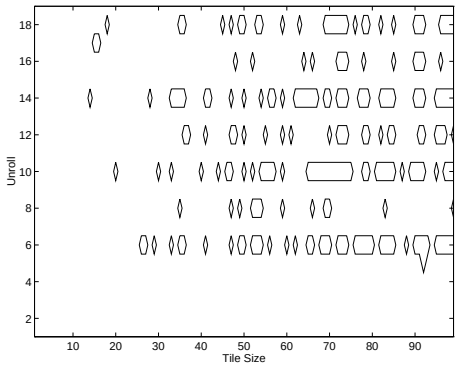
HP-PA: MxV $N = 1024$



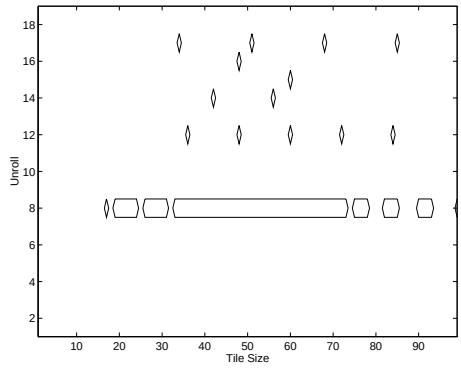
HP-PA: MxV $N = 1200$



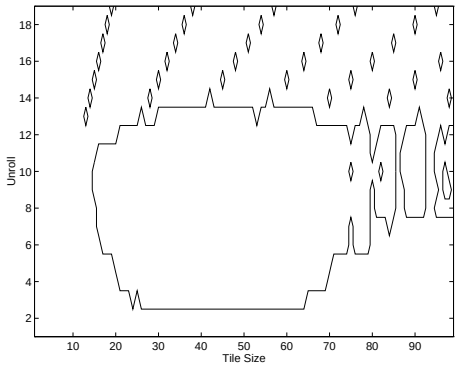
Pentium II: MxV $N = 1024$



Pentium II: MxV $N = 1200$



R4000: MxV $N = 1024$



R4000: MxV $N = 1200$

Figure 3: Areas close to the minimum

5 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.

5.1 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 resulting compilation times are small enough to

5.1.1 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.

5.1.2 Global driver

The global driver is responsible for navigating through the search space. It maintains an internal representation of this space as a multidimensional array where each dimension corresponds to the range of values for the parameters of the individual transformations (e.g., unrolling factors).

One step of the global driver consists of the following steps:

1. Decide the next set of parameters for the transformations using its internal search space and the search algorithm.
2. Construct an SSL file that corresponds to this new sequence.
3. Invoke MT1 that starts the transformation process by reading in the source program, the SSL file and the TDL file.
4. The transformed program is compiled for the target architecture and executed.
5. The execution time is measured and reported back to the global driver.
6. The global driver stores this execution time and starts the next step.

Finally, after a predetermined number of iterations, the global driver stops searching and outputs the transformed program with the shortest execution time.

5.1.3 Results

In this section we discuss the results obtained by the iterative compilation approach. We also discuss the overall compilation time.

We have executed the search algorithm for three fixed input data sizes. In Figure 4 we have shown examples of the performance improvement for three benchmarks. These results where obtained by executing the compiler and the benchmarks on a Pentium II platform running at 233 MHz. The x-axis shows the number of iterations and the y-axis shows the speedup over the original program. We restrict the number of iterations to 400 in each case, since our previous research [10, 15] showed that it is likely that within this number of iterations high levels of optimization are obtained.

In Figure 4, loop tiling, loop unrolling and array padding are applied as transformations. The first observation is that the search algorithm finds good speedups. Another observation is

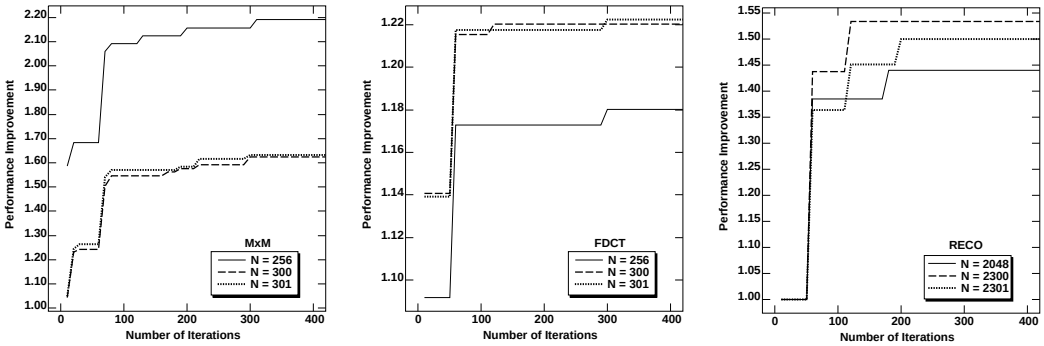


Figure 4: Performance Improvement using Unroll, Tile and Padding

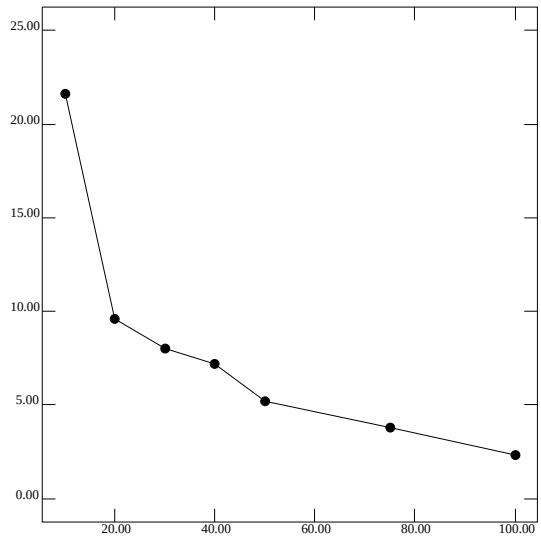


Figure 5: Average percentage difference minimum

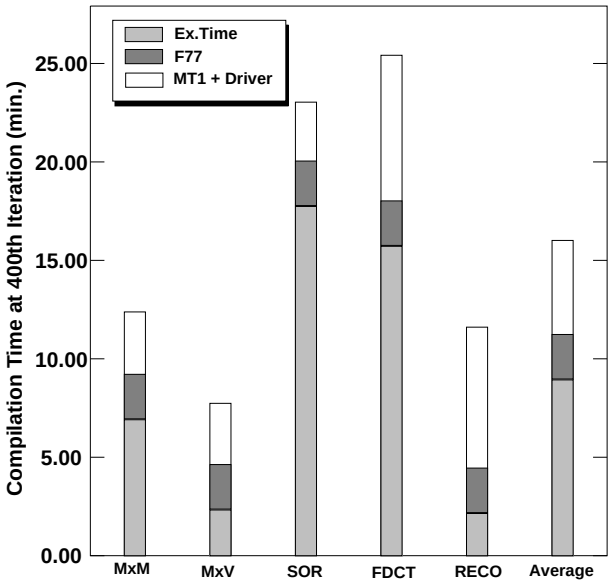


Figure 6: Compilation Time

that the search algorithm finds good parameters quickly. Within 50 evaluations in all cases, except for MxM, the performance improvement is close to maximum. For MxM more than 100 evaluations are required to obtain a good performance improvement. After 350 evaluations, there is no performance improvement observed in all cases. This corresponds to 1.75% of the entire search space.

In figure 5, 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.

5.1.4 Compilation Time

An important consideration for the feasibility of iterative compilation is the running time of the approach. Figure 6 shows the average compilation time for a number of benchmarks and the average of these times.

We observe that compilation time is proportional to the number of iterations. The average

compile time using 400 iterations ranges from 7.7 minutes (MxV) to 25.4 minutes (FDCT). On average we need 16 minutes for 400 iterations. Note that the kernels we used are representative for embedded kernels. In case we are to optimize embedded kernels this amount of time can easily be afforded. In fact, the figure shows that for time-critical kernels many more iterations can be afforded: since compilation time can be seen as an integral component of the total development time of the embedded system, we can afford several hours to heavily optimize the compute intensive routines.

The figure also shows the breakdown of the execution times for 400 iterations. On average, about 50% of the total compilation is spent executing the transformed code. The time needed for native f77 compilation is not large, but significant. Note that in case the target platform needs static instruction scheduling or software pipelining, this time can be much larger. In some cases, the time for MT1 and the global driver is larger than the execution time of the transformed code (RECO).

5.2 Alternative Search Techniques

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

5.2.1 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 the 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. 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 optimization 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 optimizing compilers are based on a fixed set of heuristics and only optimize for speed or code size, but don't search for a trade-off. Figure 7 shows how complex

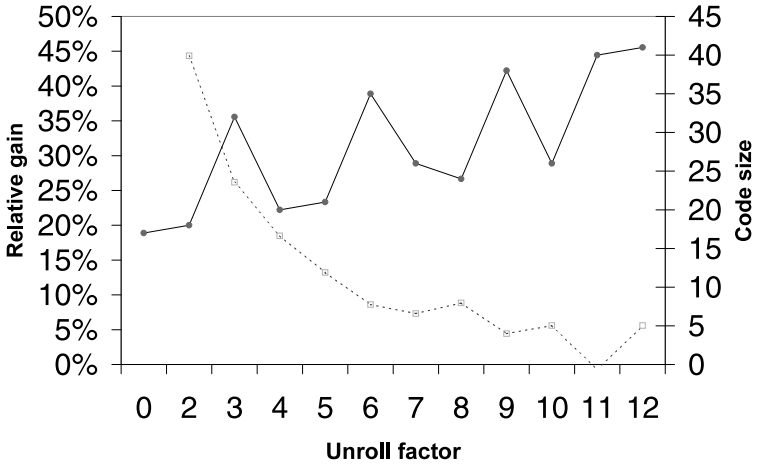


Figure 7: Code size and performance

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.

5.2.2 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.

The OCEANS GA search is implemented as part of the GAPS compiler framework described in [16] which uses GA based optimization 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 [14]. GA optimization 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 optimization will be equal to or greater than that produced by the conventional techniques.

6 Related Work

There is a large body of work considering program transformations to improve uniprocessor performance. In [12], 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 [19], and Bilmes et al. [8] describe systems for generating highly optimized 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 optimize BLAS routines and are not general purpose compilers. Experimentation with these systems [19, 8] has shown that these systems are capable of producing code that is more efficient than the vendor supplied, hand optimized library BLAS routines.

Wolf, Maydan and Chen [20] have described a compiler that also searches for the optimal optimization. This compiler also considers the entire optimization 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 optimizations. 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. [9] describe a method for searching for the best optimization 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.

7 Conclusions

In this document we have described the Esprit Reactive LTR project OCEANS. This project focusses on developing aggressive optimization techniques for embedded systems. We have discussed the compiler infrastructure and the software components from which the compiler has been built. Next we discussed the iterative approach to program optimization adopted in the OCEANS project. This approach consists of searching for the best optimization. The search is conducted along two axes. First, a high level grid-based search is conducted to find optimal parameter values for source to source transformations. Second, a low level trade-off between code size and performance is searched that examines the interplay between loop unrolling and software pipelining. We have shown that this approach to program optimization delivers highly optimized programs in a reasonable amount of compile time. Although this compile time may be too large for general purpose compilers, it can easily be afforded in case of embedded applications where compilation time can be seen as an integral part of the product development time.

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] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang, and A. Wang. Challenges in code generation for embedded processors. In P.Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, chapter 3, pages 48–64. Kluwer Academic Publisher, 1995.

[3] R.A.M. Bakker, F. Breg, P.M.W. Knijnenburg, P. Touber, and H.A.G. Wijshoff. Strategy Specification Language. Oceans Deliverable D2.1.a, 1997. Available through `www.wi.leidenuniv.nl/~peterk`.

[4] 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.

[5] M. Barreteau, F. Bodin, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, T. Kisuki, P.M.W. Knijnenburg, P. van der Mark, A. Nisbet, M.F.P. O'Boyle, E. Rohou, A. Seznec, E.A. Stöhr, M. Treffers, and H.A.G. Wijshoff. OCEANS: Optimizing compilers for embedded applications. In P. Amestoy *et al.*, editor, *Proc. Euro-Par 99*, volume 1685 of *Lecture Notes in Computer Science*, pages 1171–1175, 1999.

[6] A.J.C. Bik, P.J. Brinkhaus, P.M.W. Knijnenburg, P. Touber, and H.A.G. Wijshoff. Transformation Definition Language. Oceans Deliverable D1.1, 1997. Available through `www.wi.leidenuniv.nl/~peterk`.

[7] A.J.C. Bik and H.A.G. Wijshoff. MT1: A prototype restructuring compiler. Technical Report no. 93-32, Department of Computer Science, Leiden University, 1993.

[8] 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.

[9] 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.

[10] 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.

[11] B. Case. Philips' hope to displace DSP with VLIW. *Microprocessor Report*, 8(16):12–15, 1995. See also http://www.trimedia-philips.com/.

[12] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *Proc. Programming Language Design and Implementation*, pages 279–290, 1995.

[13] C. Eisenbeis, S. Lelait, and B. Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proc. PACT'95*, 1995.

[14] W. A. Kelly. *Optimization within a Unified Transformation Framework.* PhD thesis, Univ. of Maryland, 1996.

[15] 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*, volume 1615 of *Lecture Notes in Computer Science*, pages 121–132, 1999.

[16] A. Nisbet. GAPS: Genetic algorithm optimised parallelization. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998. Workshop organised in conjunction with PACT'98.

[17] E. Rohou, F. Bodin, A. Seznec, G. Le Fol, F. Charot, and F. Raimbault. SALTO: System for assembly-language transformation and optimization. Technical Report 1032, IRISA, Rennes, 1996. See also `http://www.irisa.fr/caps/Salto/`.

[18] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su. Decomposed software pipelining: a new perspective and a new approach. *Int'l J. on Parallel Processing*, 22(3):357–379, 1994.

[19] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proc. Alliance 98*, 1998.

[20] 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.