OCEANS: Optimising Compilers for Embedded ApplicatioNS*

Michel Barreteau[1], François Bodin[2], Peter Brinkhaus[3], Zbigniew Chamski[4], Henri-Pierre Charles[1], Christine Eisenbeis[5], John Gurd[6], Jan Hoogerbrugge[4], Ping Hu[5], William Jalby[1], Peter M. W. Knijnenburg[3], Michael O'Boyle[7], Erven Rohou[2], Rizos Sakellariou[6], 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] Department of Computer Science, Leiden University, P.O. Box 9512, 2300 RA 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, University of Manchester, Oxford Road, Manchester M13 9PL, U.K.
[7] Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, U.K.

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 optimisation approaches within an iterative framework for compilation.

1 Introduction

Embedded applications have become increasingly complex during the last few years. Although the appearance of sophisticated hardware solutions, such as those exploiting instruction-level parallelism, aims to provide improved performance, it also creates a burden for application developers. The traditional task of optimising assembly code by hand becomes unrealistic due to the high complexity of hardware/software and 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 integrates source-level restructuring transformations with low-level, machine dependent, optimisations [1, 16, 18]. A major objective of the project is to provide

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

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 and *vice versa*. Although the back-end target can be any VLIW or superscalar architecture, the Philips TriMedia (TM1000) VLIW processor [10] is currently used for the validation of the system.

In this paper, we present the work that has been carried out during the first 15 months since the project started (September 1996); this has largely concentrated on the development of the necessary compiler infrastructure. An overall description of the system is given in Section 2. Sections 3 and 4 present the high-level and the low-level subsystems respectively, while the steps that have been taken towards their integration are highlighted in Section 5. Finally, some results from the initial validation of the system are shown in Section 6, and the paper is concluded with Section 7.

2 An Overview of the OCEANS Compiler System

The OCEANS compiler is centred around two major components: a high-level restructuring system, MT1, and a low-level system for supporting assembly language transformations and optimisations, SALTO, which is coupled with SEA, a set of classes that provide 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, syntax and semantic analysis of a source FORTRAN program (`File.f`) and constructs an internal data representation on which data dependence analysis is performed. Then, a sequence of source program transformations specified in a *Strategy Specification Language* (SSL) can be applied. Each transformation is also specified using a *Transformation Definition Language* (TDL).
- The restructured source program is then fed into the code generator which generates sequential assembly code. This code makes use of virtual registers and is annotated with instruction identifiers that are used to identify common objects in MT1 and SALTO. Along with the assembly code (`File.s`) comes a file written in an *Interface Language* (`File.IL`) that provides information on data dependences and control flow graphs.
- Finally, SALTO (coupled with SEA) is in charge of producing the final code after performing code scheduling and register allocation. At this step guarded instructions are also created (so that instructions can move across branches) and resource constraints are taken into account.

The above process is repeated iteratively until a certain level of performance is reached; thus, different optimisations, both at the source-level and the low-level, can be checked and evaluated. An important feature of the system is also the existence of a client-server protocol that has been implemented in order to provide easy access to the compiler over the Internet, for all the members of the

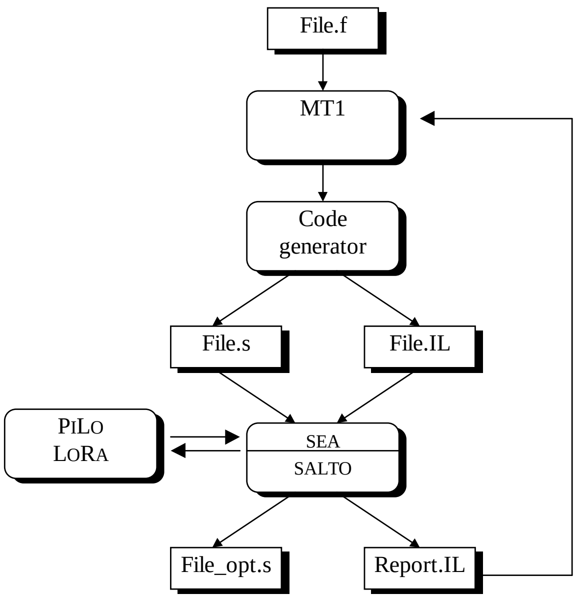

Fig. 1. The Compilation Process.

consortium. Thus, on each participating site, a client driver has been installed that can establish contact with the server at Leiden through sockets. MT1 and the code generator are located at Leiden, and SALTO, SEA, PILO and LORA are located at Rennes. In this way, all partners can have access to the latest version of the compiler, while all development is kept local.

3 High-Level Transformations

Optimizing and restructuring compilers incorporate a number of program transformations that replace program fragments by semantically equivalent fragments [20, 21]. The aim is to obtain more efficient code for a given target architecture. However, the order in which transformations are applied cannot be defined *ad hoc*. The problem of finding an optimum order (or strategy), commonly known as the *phase ordering problem*, is traditionally approached by compilers in a static way: transformations and their application order are hard-coded. Within the MT1 compilation system [3, 6, 9] this problem is solved by providing a Transformation Definition Language [4] and a Strategy Specification Language [2]. Transformations and strategies can be specified in these languages and then loaded dynamically into the compiler and executed.

3.1 Transformation Definition Language

The TDL is based on pattern matching. The general form of a transformation is the following:

```
transform
    input pattern
into
    output pattern
condition
    condition
;
```

The user can specify an *input pattern*, a transformed *output pattern* and a *condition* where the transformation can be legally and/or beneficially applied. Patterns may contain *expression* and *statement variables.* When a pattern is matched against the code these variables are bound to actual expressions and code fragments, respectively. The expression and statement variables can be used in turn in the specification of the output pattern and the condition. This mechanism allows one to specify a large number of transformations, such as loop interchange, loop distribution or loop fusion. However, it is not powerful enough to express other important transformations, such as loop unrolling where the loop body needs to be duplicated, and each occurrence of the loop index `I` needs to be replaced by `I+1` in the copy of the loop body. Therefore, the TDL also allows for *user-defined functions* in the output pattern. User-defined functions are the interface to the internal data structures of the compiler. In this way, any algorithm for transforming the code can be implemented and made accessible to the TDL. Similarly, various tests on the structure and properties of the code can be implemented.

For example, the following TDL description implements loop unrolling. In this, `!e`*n* are expression variables and `!s`*n* are statement variables. The user-defined function `tdl_replace` is used to replace in the program fragment designated by `!s1` the expression `!e1` by the third argument.

```
transform
    list(do(!e1, !e2, !e3, 1, !s1), !s2)
into
    list(do(!e1, !e2, (!e2 - 1) + ((!e3 - !e2 + 1) / 2) * 2, 2,
            merge(!s1, tdl_replace(!s1, !e1, !e1 + 1))),
    list(if(((!e3 - !e2 + 1) / 2) * 2 .neq. (!e3 - !e2 + 1),
         tdl_replace(!s1, !e1, !e3)), !s2))
condition
    tdl_isint(!e2) and tdl_isint(!e3)
;
```

3.2 Strategy Specification Language

The capability of specifying transformations is only one part of the general problem of obtaining optimal code by means of program transformations. The order in which transformations have to be applied needs to be considered also. Therefore, a Strategy Specification Language (SSL) has been implemented. This language contains sequential composition of transformations, a choice construct and two repetitive constructs and it allows the specification of an optimising strategy at a more abstract level than the source code level.

An IF statement consists of a transformation that acts as a condition, a THEN part and an optional ELSE part. The transformation in the condition can be applied successfully or not. If it is successful, the transformations in the THEN part are to be executed. Optionally, in the ELSE part a list of transformations can be given which should be executed in case the transformation matched but was not applied successfully due to failing conditions.

The two repetitive constructs consist of a transformation to be checked and a statement list to be executed if the condition is true or false, respectively. They consist of a WHILE-ENDWHILE and a REPEAT-UNTIL construct.

The language contains a mechanism for applying sequences of transformations only if they can all be applied, by means of a roll back statement. Any number of transformations can be grouped together in the roll back construct. If any of the transformations fails, the entire construct fails and no changes are made to the program under consideration. If all transformations in the group are successfully applied, the result is rolled forward. This means that the fragment that matched the first transformation is replaced in the program by the result of all transformations in this fragment.

Examples of how to specify strategies in SSL can be found in [2].

4 Low-Level Optimisations

Low-level optimisations are built on the top of SALTO, a retargetable system for assembly language transformation and optimisation. Its objective is to provide the user with a single environment for implementing algorithms needed for performance tuning of low-level codes [17]. To facilitate the implementation of optimisations, a set of classes has been designed, SEA (SALTO Enhanced Abstraction), that provides an abstract view of the assembly code which is more pertinent to the code scheduling and register allocation problems. The most important features of SEA are that it allows the evaluation of various code transformations before producing the final code, and that it separates the implementation of the global low-level optimisation strategy from the implementation of individual optimisation sequences.

The SEA model contains two kinds of objects:

code fragments, which are parts of the control flow graph. The following objects can be used in SEA: `seaINST`: an instruction object; `seaCF`, an unstructured set of code fragments; `seaSCF`, a structured subset of control flow graph with a unique entry point; `seaSBk`, a superblock; `seaBBk`, a basic block; and finally, `seaLoop`, a structured piece of code that has loop properties. Figure 2 illustrates the corresponding class hierarchy.

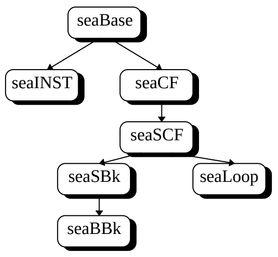

Fig. 2. Sea class hierarchy.

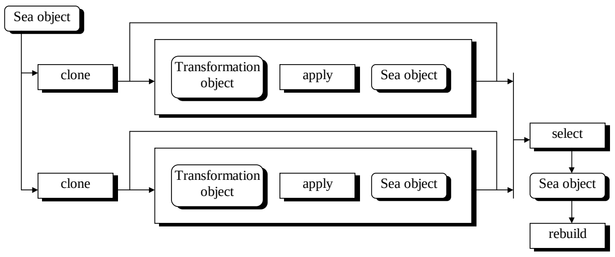

Fig. 3. Typical usage of Sea classes.

transformations to be applied to subgraphs. All transformations are characterized by the following main methods: `preCond()` returns the set of control flow subgraphs that qualifies for the transformation; `apply()` applies the transformation to a given subgraph, and finally, `getStatus()` that returns the status of the transformation after application. The status can be *success* or *failure*, and the reason of the failure can be extracted through a generic diagnostics mechanism.

The usage of the Sea objects is shown in figure 3. A transformation is tried on a cloned piece of code, then according to performance or size criteria one of the solutions found is chosen and propagated to the low-level program representation using the `rebuild()` method.

The following optimisations are currently available within Sea:

– *Register renaming* renames local registers in each basic block.
– *Superblock construction* merges a set of basic block into a superblock [14].
– *Guard insertion* adds guards to instructions to remove jumps and thus allow scheduling across jumps [13].
– *Loop unrolling* (also available at the high-level).
– *Local/superblock scheduling* [14, 15].
– *Software pipeline* generates a modulo scheduling of the loop body. The implementation is based on the tools PiLo [19] and LoRa [11]. PiLo and LoRa are optimisation kernels based on periodic scheduling and graph colouring algorithms. Sea sends data dependencies between instructions, the architectural information and rebuilds the new code according to the produced scheduling and register allocation scheme.
– *Register allocation*; this can be performed either before or after scheduling the instructions.

5 Integration

5.1 The Interface Language

In order to transmit information between the various components of the compiler, an *Interface Language* (IL) was designed. This allows the propagation of information, such as data dependences and loop control data, from MT1 to SALTO, as well as feedback information from the scheduled code, to which a given sequence of transformations has been applied, back to MT1.

An IL description consists of three sections: *a list of keywords* that specifies the list of attributes that can apply to an object; *a default level setting* that indicates the type of code the objects belong to; and, *a list of object references* which specify the nature, contents and attributes of an object. These sections are illustrated in the example given in Figure 4 which is further explained next.

More details on the IL can be found in [8].

5.2 Information Forwarded and Feedback

Data dependences are propagated from MT1 to SALTO and are used for memory disambiguation. Based on the results of high-level data dependence analysis, SALTO may safely assume that all load and store operations refer to different memory locations unless it is specified in the `.IL` file that they refer to the same location.

The feedback from SEA to MT1 (file `Report.IL` in Figure 1) contains information on the code structure, the basic blocks, as well as a record of the transformations that were applied. An example is given in Figure 4. Data related to each basic block include the total number of assembly instructions, the critical path for scheduling the code, the number of cycles of the scheduled code, and a grouping depending on the nature of the instructions. In the example, it can be seen that the compaction rate of the scheduled code (as specified by `.nbCycles`) is close to optimum (as specified by `.CriticalPath`).

MT1 uses the feedback from SALTO in order to build an internal data structure that can be accessed by the TDL by means of user-defined functions. For the SSL, identity transformations are defined that match against an arbitrary program fragment, such as a `DO` loop. User-defined functions in the condition can check for the identity of the loop and the suggestions made by SALTO. When such a transformation is used as the condition for an IF construct in the SSL, we are able to select the transformations we want to apply next to this fragment.

Initially, MT1 compiles the program without performing any restructuring and the compiled program is scheduled by SALTO. SALTO identifies the code fragments that can be improved. It reports its diagnostics to a cost model that makes a decision on what kind of restructuring could be performed next. Then, MT1 reads the suggestions for restructuring and performs these. It is intended that the optimum transformation sequence for a program fragment is found by following a systematic approach for searching through a domain of possible transformations. Thus, the optimisation process is represented by a search tree whose nodes contain the parameter settings used for a given transformation and the performance returned. First, each different transformation is applied once and then the same follows for each branch of the tree. The search space is minimised by using a threshold condition for terminating branches that are not likely to yield an optimum result in their descendants. Some preliminary experiments us-

```
// final code structure description
#(SS 9961 , DESC = { #(SS 9870 )})

//basic blocks data
#(BB 9870).nbAsm := 73
#(BB 9870).CriticalPath := 18
#(BB 9870).nbCycles := 20
#(BB 9870).nbLoad := 6
#(BB 9870).nbStore := 3
#(BB 9870).nbAlu := 60

// transformation history for block 9870
// order is the order number of the transformation
#(BB 2).became := {#(BB 9870)}
#(BB 9870).renameLocalReg := {{order, 0}, {from, 2}, {to, 2}, {Status, oK}}
#(BB 9870).collapsing := {{order, 1}, {from, 2}, {to, 9870}, {Status, oK}}
#(BB 9870).guards := {{order, 2}, {from, 9870}, {to, 9870}, {Status, oK}}
#(BB 9870).scheduling := {{order, 3}, ..., {Status, oK}, {nbCycles,  20}}
#(BB 9870).registerAllocation := {{order, 4},..., {Status, oK}}

// transformation history for loop 1
#(SS 1).became := {#(SS 9961)}
#(SS 9961).copy := {{order, 0}, {from, 1}, {to, 9976}, {Status, oK}}
#(SS 9961).copy := {{order, 1}, {from, 9976}, {to, 9961}, {Status, oK}}
#(SS 9961).unroll := {{order, 2}, ... , {Status, oK}, {Unroll,  3}}
```

Fig. 4. Feed back information.

ing this strategy can be found in [12]; ongoing work is investigating the feasibility of this approach.

6 Validation of the Initial System

In order to validate the compiler, four public domain multimedia codes have been selected. These are a low bit-rate encoder/decoder for H.263 bitstreams, an MPEG2 encoder/decoder, an implementation of the CCITT G.711, G.721 and G.723 voice compression standards, and the Persistence of Vision Ray-Tracer for creating 3D graphics. These codes were profiled and analysed and the most time-consuming parts were identified and provided an initial focus of attention [5].

At the high-level, initial experimentation aimed at identifying those transformations that appear to be the most crucial in optimising code scheduling as well as to improve the performance of the code generator. The inspection of the benchmarks revealed that they contain many imperfectly nested double or triple loops where much overhead can be caused due to branch delays. In order to deal with such loops, a transformation that converts the imperfectly nested loop into a single loop has been suggested. This is achieved by: moving all the code

	S_0	$S_1(u)$	$S_2(u)$	S_3
step 1	local scheduling	unroll$\times u$	unroll$\times u$	software pipelining
step 2	reg. allocation	superblock	superblock	
step 3		scheduling	reg. allocation	
step 4		reg. allocation	scheduling	

Fig. 5. Sequences of optimisations; u is the unroll factor.

inside the innermost loop and properly guarding it; collapsing (or coalescing) the resulting perfect loop nest to a single loop; and, finally, adding extra code to maintain the proper values of the original loop indices without resorting to the expensive `div` and `mod` operations. Preliminary experiments have shown that the length of the resulting schedule can be as much as 40% shorter than the original schedule [5].

At the low-level, the initial validation of the system has been carried out by applying four different optimisation sequences:

- S_0 is the simplest sequence. First, the code is scheduled locally and then register allocation is performed.
- $S_1(u)$ is based on unrolling the loop body u times. The unrolled body is transformed into a superblock, and conditional instructions are eliminated through the insertion of guards, resulting in a large basic block. As in S_0, register allocation is performed after local scheduling.
- $S_2(u)$ is similar to $S_1(u)$ except that register allocation is performed before scheduling. This decreases the code compaction potential, but usually requires less registers, allowing this sequence to succeed when $S_1(u)$ fails due to a lack of registers. Currently, this is the least effective optimisation sequence, since register allocation introduces many anti-dependences.
- S_3 consists in applying a software pipelining algorithm. This sequence is limited to loops whose bodies constitute a single basic block.

A summary is given in Figure 5.

The above optimisation sequences were validated using the OCEANS set of benchmarks and targeting the TriMedia architecture [10]. Indicative results, using the most-time consuming loops of the H263 application, are illustrated in Figure 6. Every optimisation sequence has been applied to each of the six selected loops and the size of the resulting VLIW code and the speed of the loop, i.e. the number of cycles per iteration, were computed. From the table, a well-known result is observed: the more we unroll a loop, the faster it runs — cf. columns $S_1(2)$, $S_1(3)$, $S_1(4)$ — but at the expense of a larger code size. As expected $S_2(2)$ yields too poor performance and large code because of the presence of false dependences. Finally, software pipelining (S_3) gives the best performance but at the expense of a very large increase in code size. Note that this transformation failed with the last loop, due to a lack of registers.

Using these results more general problems are addressed by the compiler. For instance, in most embedded applications, it is necessary to answer globally

	Optimisation sequences						C code
	S_0	$S_1(2)$	$S_1(3)$	$S_1(4)$	$S_2(2)$	S_3	
speed	8	6	5	5	7	3	`for (i=xa;i<xb; i++) { d[i]=s[i]*om[i]; }`
size	8	12	16	20	13	75	
speed	9	7	6	6	10	5	`for (i=xa; i<xb; i++) { d[i]+=s[i]*om[i]; }`
size	9	13	18	22	19	55	
speed	12	8	8	7	12	6	`for (i=xa; i<xb; i++) { dp[i]+=(((unsigned int)(sp[i] +sp2[i]+1))>>1)*om[i]; }`
size	12	16	24	28	24	121	
speed	15	10	9	9	16	6	`for (i=xa; i<xb; i++) { dp[i]+=(((unsigned int)(sp[i]+ sp[i+1]+1))>>1)*OM[c][j][i]; }`
size	15	20	28	34	31	172	
speed	15	10	10	8	17	7	`for (i=xa; i<xb; i++) { dp[i]+=(((uint)(sp[i]+sp2[i]+ sp[i+1]+sp2[i+1]+2))>>2)*om[i]; }`
size	15	19	29	33	33	179	
speed	19	13	12	11	30	–	`for (k=0; k<5; k++) { xint[k]=nx[k]>>1; xh[k]=nx[k] & 1; yint[k]=ny[k]>>1; yh[k]=ny[k] & 1; s[k]=src+lx2*(y+yint[k])+x+xint[k]; }`
size	19	25	36	44	59	–	

Fig. 6. Time consuming loops extracted from H263

questions such as: "Given a maximum code size, what is the highest performance that can be achieved?", or "Given a performance goal, what is the smallest code size that can be achieved?". Within the OCEANS compiler this trade-off is evaluated quantitatively by applying a novel compiler strategy based on an integer linear optimisation model. Thus, the choice of the most suitable optimisation is made *a posteriori*, when the impact of each possible transformation is known. More details can be found in [7].

7 Conclusion and Future Work

The previous sections outlined the current status of the OCEANS compiler. Although the results obtained so far, using the initial prototype, are satisfactory (comparing with a production compiler), the implementation work still continues on both the high and low levels. A major part of the work during the next months and until the end of the project is devoted to the integration of the two levels and the development of a framework for iterative compilation. It is also noted that effort is currenly being undertaken to develop a JAVA front-end. Finally, it is intended that the system be made publically available in due time (at the moment SALTO is already available).

Acknowledgements: The work described in this paper has benefited greatly from the contributions of Ronald A. M. Bakker, Aart J. C. Bik, Ben H. H. Juurlink, and Pieter Touber. Many thanks are also due to the members of the compiler technology group at Philips Research, the project reviewers and the project officer.

References

1. B. Aarts, *et al.* OCEANS: Optimizing Compilers for Embedded Applications. In C. Lengauer, M. Griebl, S. Gorlatch (Eds.), *Proceedings of Euro-Par'97*, Lecture Notes in Computer Science 1300, Springer-Verlag, 1997, pp. 1351–1356.
2. R. A. M. Bakker, F. Bregt, P. M. W. Knijnenburg, P. Touber, and H. A. G. Wijshoff. Strategy Specification Language. OCEANS Deliverable D1.2a, 1997.
3. A J. C. Bik. A prototype restructuring compiler. Master's thesis, Utrecht University, 1992. INF/SCR-92-11.
4. 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.
5. A. J. C. Bik, P. J. Brinkhaus, P. M. W. Knijnenburg, P. Touber, H. A. G. Wijshoff, W. Jalby, H.-P. Charles, M. Barreteau. Identification of Code Kernels and Validation of Initial System. OCEANS Deliverable D3.1c, 1997.
6. A. J. C. Bik and H. A. G. Wijshoff. MT1: A Prototype Restructuring Compiler. Technical Report 93-32, Department of Computer Science, Leiden University, 1993.
7. F. Bodin, Z. Chamski, C. Eisenbeis, E. Rohou, and A. Seznec. GCDS: A Compiler Strategy for Trading Code Size Against Performance in Embedded Applications. Research Report 1153, IRISA, 1997.
8. F. Bodin and E. Rohou. High-level Low-level Interface Language. OCEANS Deliverable D2.3a, 1997.
9. P. Brinkhaus. Compiler analysis of procedure calls. Master's thesis, Utrecht University, 1993. INF/SCR-93-13.
10. 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/`
11. C. Eisenbeis, S. Lelait, and B. Marmol. The meeting graph: a new model for loop cyclic register allocation. *Proceedings of PACT'95* (Cyprus, June 1995).
12. J. Gurd, A. Laffitte, R. Sakellariou, E. A. Stöhr, Y. T. Chu, and M. F. P. O'Boyle. On Compile-Time Cost Models. OCEANS Deliverable 1.2b, 1997.
13. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler Technology for Future Microprocessors. *Proceedings of the IEEE*, 83(12), Dec. 1995, pp. 1625–1639.
14. W. Hwu, *et al.* The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1), May 1993, pp. 229–248.
15. D. R. Kerns and S. J. Eggers. Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain. *Conference on Programming Language Design and Implementation*, 1993, pp. 278–289.
16. OCEANS Web Site at `http://www.wi.leidenuniv.nl/Oceans/`

17. E. Rohou, F. Bodin, A. Seznec, G. Le Fol, F. Charot, F. Raimbault. SALTO: System for Assembly-Language Transformation and Optimization. Technical Report 1032, IRISA, June 1996. See also `http://www.irisa.fr/caps/Salto/`
18. R. Sakellariou, E. A. Stöhr, and M. F. P. O'Boyle. Compiling Multimedia Applications on a VLIW Architecture. *Proceedings of the 13th International Conference on Digital Signal Processing (DSP97)* (Santorini, July 1997), vol. 2, IEEE Press, 1997, pp. 1007–1010.
19. J. Wang, C. Eisenbeis, M. Jourdan, and B. Su. Decomposed Software Pipelining: a New Perspective and a New Approach. *International Journal on Parallel Processing*, 22(3), 1994, pp. 357–379.
20. M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.
21. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.