OCEANS:
Optimizing Compilers for Embedded Applications*

Bas Aarts[1], Michel Barreteau[2], François Bodin[3], Peter Brinkhaus[4], Zbigniew Chamski[3], Henri-Pierre Charles[2], Christine Eisenbeis[5], John Gurd[6], Jan Hoogerbrugge[1], Ping Hu[5], William Jalby[2], Peter M. W. Knijnenburg[4], Michael F. P. O'Boyle[7], Erven Rohou[3], Rizos Sakellariou[6], Henk Schepers[1], André Seznec[3], Elena Stöhr[6], Marco Verhoeven[1], and Harry A. G. Wijshoff[4]

[1] Philips Research, Information and Software Technology, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands.
[2] Laboratoire PRiSM, Université de Versailles, 78035 Versailles, France.
[3] IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, France.
[4] Department of Computer Science, Leiden University, P.O. Box 9512, 2300 RA Leiden, 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] Department of Computer Science, The University, Edinburgh EH9 3JZ, U.K.

Abstract. This paper describes the recently funded ESPRIT project OCEANS. Its aim is to investigate and develop advanced compiler infrastructure for embedded VLIW processors, such as the Philips TriMedia. Such processors promise high performance at low unit cost. This paper outlines the project's aims, presents the compiler infrastructure and its application to a typical case study.

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 [1].

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

The goal of the OCEANS project 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. A brief presentation of the project is provided in this paper. In particular, the project objectives are highlighted in Section 2; Section 3 presents the structure of the compiler, that is, the front-end, the back-end, and their interaction. Finally, Section 4 examines the possible implications for the compiler when optimizing codes that are frequently used in embedded applications.

2 Project Objectives

Within the OCEANS project, we intend to meet the following objectives:

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 will be based on a cost model of the processor and will be 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 validation and the evaluation of the efficiency of this approach will be carried out in close collaboration with the industrial validator, the compiler technology group at Philips Research. The main back-end target for this project is the Philips TriMedia (TM-1) processor [5]. 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.

3 The OCEANS Compiler

The OCEANS compiler consists of a *front-end*, incorporating a high-level restructuring tool, MT1, and a *back-end*, incorporating a system for assembly

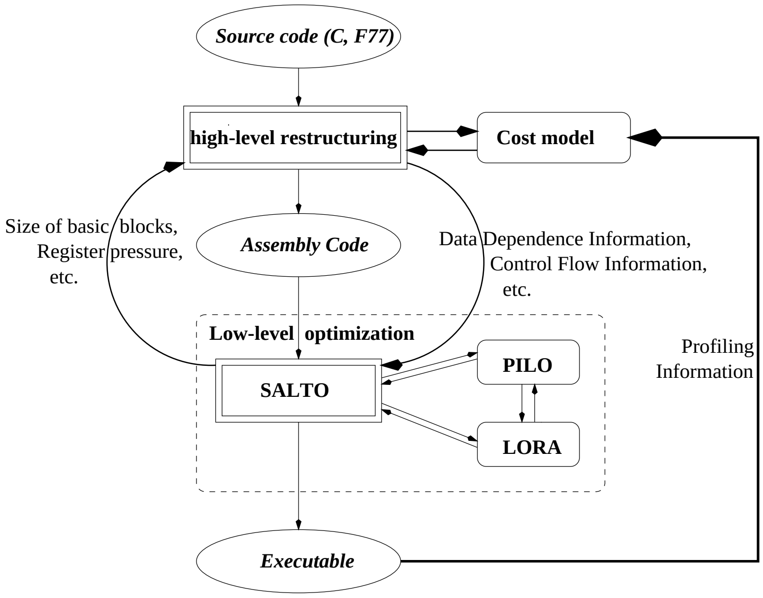

Fig. 1. The OCEANS compiler.

language transformation and optimization, Salto; the latter makes use of PiLo and LoRa, which are packages for software pipelining and loop register allocation, respectively. Their interaction is illustrated in Figure 1. It is intended that quantitative (e.g. number of instructions, slot occupancy, execution time) and qualitative information (e.g. pipelining failed due to register pressure) be used to guide the compilation process. By using an accurate cost model, the system can iterate until an acceptable level of performance is achieved.

The main back-end target of the OCEANS compiler is the Philips TriMedia (TM-1) processor [5], a state-of-the-art general purpose microprocessor, which has been enhanced to boost multimedia performance. At the heart of it, there is a 400 MB/s bus, which connects autonomous modules that include video-in, video-out, audio-in, audio-out, an MPEG variable length decoder, an image co-processor, a communications block and a VLIW processor. The VLIW processor includes a rich instruction set with many extensions for handling multimedia, and is capable of sustaining 5 RISC operations per clock cycle at 100 MHz. It contains 27 functional units which are pipelined ranging from 1 to 3 deep. The processor also includes 32 KB of instruction cache memory and 16KB of data cache memory.

The MT1 Restructuring Compiler: Over the past few years, a full Fortran 77 compiler, called MT1, has been developed at Leiden University [2]. An important aspect of the MT1 compiler is that it provides a facility for specifying program transformations that can be applied interactively. 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.

SALTO: A Retargetable System for Assembly Language Transformation and Optimization: Salto [7] 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 [8], 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 [6]. PiLo and Lora are connected to Salto via an interface describing architectural and dependency constraints among instructions.

Integration: In order to implement an iterative compilation process, MT1 constructs two output files for each input program. One consists of sequential TM-1 assembly code, to be scheduled by Salto, and the other consists of the result of program analysis by MT1, written in a format that can be read by Salto. This file includes high-level dependence information and information about loop structures. It also contains questions for Salto, such as “How does software pipelining perform on this loop?”. After code scheduling by Salto and possibly profiling the resulting code, answers to these questions are used in the next compilation round and drive the selection of transformations and strategies to be applied.

4 A Case Study

In this section we present a case study of the optimizations a compiler can perform on multimedia applications. We consider four sets of benchmark programs, publically available on the Web. They consist of an MPEG2 encoder/decoder for converting uncompressed video frames into MPEG1/2 and *vice versa*, an MPEG1/2 player, an implementation of the CCITT G.711, G.721, and G.723 voice compression standards, and a very low bit-rate video encoder producing H.263 bitstreams. Profiling information has been obtained using `gprof` and further analysed using `tcov`. Although there are differences between the computationally most expensive functions of various programs, a regular pattern can be observed: typically, such functions contain double nested loops, having a rather small number of iterations, and embodying only a few statements mainly involving operations between array elements. For a more concrete example, consider the code fragment shown below taken from a function of the MPEG2 encoder in which approximately 67% of the program's execution time is spent.

```
for (j=0; j<h; j++)
{ for (i=0; i<16; i++)
  {  v = ( (unsigned int)(p1[i]+p1[i+1]+1)>>1 ) - p2[i];
     if (v >= 0)  s += v;  else  s -= v;
  }
  p1+= lx;  p2+= lx;
}
```

For typical RISC instruction sets, the innermost loop will require 14 instructions to execute one iteration. If we assume a latency of 3 cycles for load/store and a delay of 3 cycles for a jump, a naive sequential schedule would require 336 cycles to execute the entire `i` loop. Conversely, if we assume that a scheduler was able to utilise fully all five function units, without regard to resource and dependence constraints, and the latency of loads and jumps was masked, then the minimal execution time is 45 cycles. Thus, we have an upper and a lower bound on expected performance.

Modulo scheduling with an initiation interval $II = 5$ cycles generates code with prologue and epilogue costs of 5 cycles each, giving a total of 80 cycles for the inner loop. If more sophisticated scheduling with $II = 4$ is used, then the inner loop takes 68 cycles. The prologue and epilogue costs increase in this case to 6 and 10 cycles respectively. However, even this optimized schedule takes 50% longer to execute than our ideal lower bound. This is largely due to the prologue and epilogue overhead for small iteration counts. Unlike many scientific benchmarks, multimedia application codes are characterised by short inner loops, and therefore additional techniques are required to improve function unit utilisation.

Since scheduling with $II = 3$ is not possible due to dependence constraints, we cannot improve the performance by reducing II further and other techniques should be devised. Unroll and jam [3] is a technique which may be used to increase the size of inner loop bodies, thus reducing the loop overhead. In [4], a quantitative approach to the application of this technique is described showing improvements in most cases when applied to the Perfect Benchmarks. Unrolling the j loop once in this example and fusing (or jamming) the resulting two inner loops allows the new inner loop body to be scheduled with $II = 6$ and a prologue and an epilogue cost of 6 cycles each. Thus, two iterations of the outer loop take 96 cycles or 48 cycles for one iteration – just 6% longer than the lower bound. This has been achieved due to the greater freedom in scheduling more instructions and fewer jump instructions. This illustrates the importance of looking beyond simple pipelining of inner loops and applying high-level transformations when examining multimedia applications.

The above analysis, although encouraging, has not considered the impact of cache misses. Typical cache lines can hold 64 bytes or 16 words and therefore 2 new cache lines will be loaded on each iteration of the j loop. If we assume an 11 cycle delay per cache miss then the execution time for the inner loop will increase to 102, 90 or 70 cycles depending on the scheduling employed. In order that any gains from exploiting instruction level parallelism are not lost due to cache misses, it is necessary to prefetch the cache line towards the end of the execution of the inner loop. Thus, careful attention to prefetching is needed.

5 Conclusion

A brief overview of the OCEANS project has been presented. The main innovation of this project is the use of an iterative approach to compilation applying both high and low-level optimizations. These are guided by information gained from either level as well as previous compilation runs. A small example illustrated the need for both high-level transformations and low-level scheduling when optimizing typical multimedia codes, and indicated that VLIW architectures, given sufficient compiler support, are capable of delivering high performance.

References

1. G. Araujo *et al.* Challenges in Code Generation for Embedded Processors. In *Code Generation for Embedded Processors.* Kluwer Academic Publishers, pp. 49–64, 1995.
2. A. J. C. Bik, H. A. G. Wijshoff. MT1: A Prototype Restructuring Compiler. Technical Report 93-32, Department of Computer Science, Leiden University, Oct. 1993.
3. S. Carr, K. Kennedy. Improving the ratio of memory operation to floating-point operations in loops. *ACM ToPLaS*, 16(6), Nov. 1994, pp. 1768–1810.
4. S. Carr. Combining Optimizations for Cache and Instruction-Level Parallelism. *Proceedings of PACT'96.*
5. 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/
6. C. Eisenbeis, S. Lelait, B. Marmol. The meeting graph: a new model for loop cyclic register allocation. *Proceedings of PACT'95.*
7. 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
8. J. Wang, C. Eisenbeis, M. Jourdan, B. Su. Decomposed Software Pipelining: a New Perspective and a New Approach. *International Journal on Parallel Processing*, 22(3), 1994, pp. 357–379.