COMPILING MULTIMEDIA APPLICATIONS ON A VLIW ARCHITECTURE*

Rizos Sakellariou, Elena A. Stöhr
Department of Computer Science
University of Manchester
Oxford Road, Manchester M13 9PL, U.K.
{rizos,elena}@cs.man.ac.uk

Michael F. P. O'Boyle
Department of Computer Science
University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, U.K.
mob@dcs.ed.ac.uk

Abstract

This paper describes a new compiler-directed approach to obtaining high-performance for multimedia applications. By integrating high-level program restructuring and low-level scheduling, an iterative approach to optimizing for embedded VLIW processors is presented. The benefit of such an approach is highlighted by way of a case study.

1. INTRODUCTION

In order to enable the implementation of cost-efficient DSP applications, Digital Signal Processors (DSPs) were introduced as special-purpose processors in the early 1980s. DSPs have a number of special features to support frequently used operations in DSP applications. Such features include support of instruction level parallelism, fast addition/multiplication instructions, support for low loop overhead, and modulo and bit-reversed addressing modes [1]. However, compilers for DSPs have generally been unable to exploit these features efficiently [2, 3]. As a result, a significant part of DSP applications has to be optimized manually in assembly; this is a costly and lengthy process, which, as DSP applications become larger and more sophisticated, is no longer desirable.

One reason for the failure of compilers to produce efficient code can be attributed to the compiler-unfriendly special architectural features of DSPs, which require extensive analysis [3]. In order to overcome this, it has been suggested to design DSP architectures and compilers in parallel, in an attempt to get feedback from each other's design [2, 4]. However, as processor cost drops, it becomes more attractive to use a general-purpose processor rather than designing specific hardware. In particular, *Very Long Instruction Word* (VLIW) processors are an attractive solution as they provide potentially high performance, due to multiple parallel functional units, and are relatively cheap to manufacture due to the simple processor architecture. These characteristics make them particularly suitable for embedded systems; multimedia applications are typical of the growing uses of the latter.

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

This paper describes research carried out under the ESPRIT project OCEANS [5]. The main aim of this project is to investigate and develop advanced compiler infrastructure for embedded VLIW processors, such as the new Philips TriMedia processor [6], targeting at multimedia applications. In this paper, we present the structure of such a compiler; its main innovation is the use of an iterative approach to compilation. We further consider the importance of high-level (i.e., source-to-source) optimization techniques for such applications. Preliminary results support this assertion.

2. VLIW ARCHITECTURES

VLIW architectures provide a platform for executing more than one instruction every clock cycle. In order to do this, they comprise multiple functional units which can operate concurrently. As opposed to *superscalar architectures* [7], which employ special hardware to reorder machine instructions for parallel execution, VLIW architectures leave this task to the software. As a result they have a simple architecture, but sophisticated compiler technology is needed to exploit the fine-grain parallelism. With current compiler technology, the average number of operations per cycle in VLIW processors is only 2 to 2.5 operations [8].

Recently, Philips have announced a general-purpose microprocessor, the Philips TriMedia (TM-1) [6], which has been enhanced to boost multimedia performance and may become a standard in future embedded applications. At the heart of the TriMedia chip, 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 deep to 3 deep. The processor also includes 32 KB of instruction cache memory and 16KB of data cache memory.

When using such a chip as the main back-end target for running multimedia applications, an important factor for obtaining good performance is the compiler. The structure of a suitable compiler is presented in the next section.

3. A VLIW COMPILER

The compiler consists of a *front-end*, incorporating a high-level restructuring tool, and a *back-end*, incorporating a system for assembly language transformation and optimization. The high-level restructuring tool accepts Fortran 77 or C input and produces a transformed optimized source program. Apart from standard compiler optimizations, such as interprocedural constant propagation, the high-level restructurer performs data dependence analysis and can also apply a number of advanced restructuring transformations, such as loop distribution or loop interchange [9]. The impact of these transformations on performance is evaluated by a cost model which can use information from critical path analysis or previous program runs. After restructuring the source program, the code generator generates assembly code which is fed into the low-level. This includes tools for low-level transformations, such as software pipelining, assembly code optimization, and register allocation. The final product of this process is long instruction assembly code (executable), tailored to the VLIW architecture. The structure of the compiler is shown diagramatically in Figure 1. More detailed information can be found in [5].

The main innovation in the design of the compiler is the use of an iterative approach to compilation. Current compilers usually ignore information about the performance success or failure of previous compilations. However, in the case of a compiler targeted at embedded applications, where long compilation times can be afforded, this information may be used to develop more sophisticated program optimization techniques. As a result, compilation is viewed as an iterative process rather than a single stage pass. A further innovation of the compiler is the use of both high-level and low-level transformation techniques to obtain high performance. A compiler for a VLIW-based DSP architecture has also been described previously in [2]; however, this uses only low-level optimizations for exploiting instruction level parallelism. Our approach considers the use of high-level transformations as well, as a means for enabling the exploitation of instruction level parallelism. The need for going beyond low-level transformations when compiling

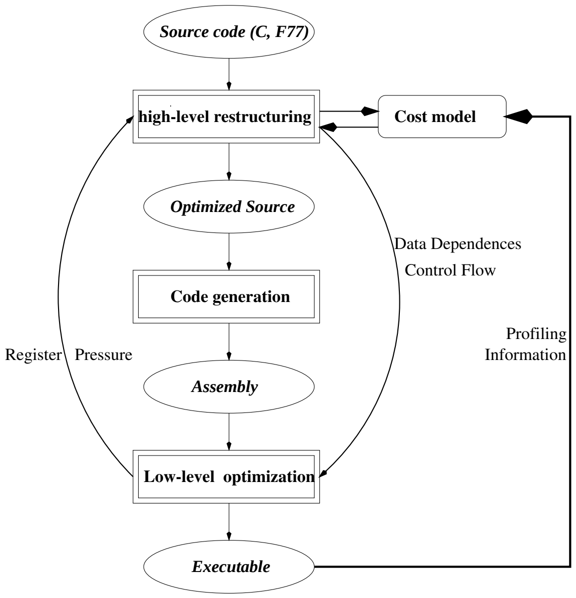

Figure 1. The structure of the compiler.

```
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;
}
```

Figure 2. Code fragment from MPEG2 encoder.

multimedia codes on a VLIW architecture is illustrated in the following section.

4. A CASE STUDY

We consider a program implementing an MPEG2 encoder/decoder for converting uncompressed video frames into MPEG1/2 and *vice versa*, publically available on the Web. Profiling information has been obtained using `gprof` and further analysed using `tcov`. We concentrate on the code fragment shown in Figure 2; the corresponding loop nest constitutes a typical program operation of a function computing the absolute difference of two blocks, which, in several cases, takes up to 67% of the program execution time.

For typical RISC instruction sets, the innermost loop will require 14 instructions to execute one iteration; the corresponding code is shown in Figure 3. If we assume a latency of 3 cycles for a load and a delay of 3 cycles for a jump, a naive sequential schedule would require $21 \times 16 = 336$ cycles to execute the entire `i` loop. Conversely, if we assume that a schedule was able to utilise fully all five functional units, without regard to resource

```
 0:LD     R2,$p1[Ri]    ; R2=p1[0]
 1:ADD    Ri,Ri,1       ; Ri=Ri+1
 2:MOV    R3,R2         ; R3=R2
 3:LD     R2,$p1[Ri]    ; R2=p1[i+1]
 4:ADD    R4,R3,R2      ; R4=p1[i]+p1[i+1]
 5:ADD    R4,R4,1       ; R4=R4+1
 6:SHR    R4,1          ; R4=R4>>1
 7:LD     R5,$p2[Ri-1]  ; R5=p2[i]
 8:SUB    R6,R4,R5      ; R6=R4-R5
 9:UGEQ   R7,R6,0       ; if (R6>=0) R7=TRUE
10:ADDIF  R7,R8,R8,R6   ; if R7 R8=R8+R6
11:ULT    R9,R6,0       ; if (R6<0) R9=TRUE
12:SUBIF  R8,R8,R6      ; if R9 R8=R8-R6
13:ULT    R10,Ri,16     ; if (Ri<16) R10=TRUE
14:JMPT   R10,Label     ; if R10 goto 1
```

Figure 3. Assembly code for the `i` loop of Figure 2.

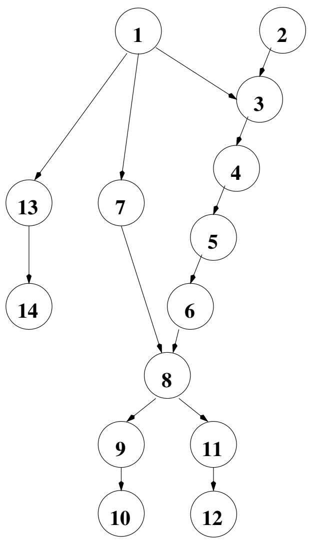

Figure 4. Data Dependence Graph.

and dependence constraints, and the latency of loads and jumps was masked, then the minimal execution time is $\lceil(14 \times 16)/5\rceil = 45$ cycles. Thus, we have an upper and a lower bound on expected performance.

In order to transform the code shown in Figure 3 into a VLIW form, we have to consider the dependences between instructions. The dependence graph is shown in Figure 4. It is noted that the graph shows dependences only within a single iteration. Cross-iteration dependences are not shown, but they can easily be satisfied. Aside from dependence constraints, architectural constraints also exist on which instructions can be executed by each functional unit. In our case, ALU operations can be executed by all 5 functional units, branch operations by three functional units (say the 2nd, 3rd and 4th), memory operations by two functional units (say the 4th and 5th), and shift operations also by two functional

cycle	Functional Unit				
	1	2	3	4	5
x	1_i	2_i	13_{i-1}	5_{i-1}	—
$x+1$	6_{i-1}	—	14_{i-1}	7_i	3_i
$x+2$	8_{i-1}	—	—	—	—
$x+3$	9_{i-1}	11_{i-1}	—	—	—
$x+4$	10_{i-1}	12_{i-1}	—	—	4_i

Figure 5. Scheduling with $II = 5$.

cycle	Functional Unit				
	1	2	3	4	5
x	1_i	6_{i-2}	2_i	14_{i-1}	—
$x+1$	—	13_i	8_{i-2}	7_i	3_i
$x+2$	9_{i-2}	—	11_{i-2}	—	4_{i-1}
$x+3$	10_{i-2}	—	12_{i-2}	5_{i-1}	—

Figure 6. Scheduling with $II = 4$.

units (say the 1st and 2nd).

Taking into account the above constraints, we use *modulo scheduling* [10], a form of software pipelining where the same schedule is used for every iteration of a loop. Successive iterations are initiated after a constant number of cycles; this number is called the *Initiation Interval* (II). The code can be scheduled following the kernel shown in Figure 5. Each row represents the instructions that are executed by each functional unit during one machine cycle; the '—' corresponds to a `NOP` operation. The subscript in each instruction refers to the iteration for which the particular instruction is executed. All instructions are executed once every 5 cycles (that is, with an initiation interval $II = 5$). In order to execute the `i` loop (i.e., 16 iterations), 17 repetitions of the kernel are required (note that during the first and the last repetition not all instructions are executed). Thus, the execution of the `i` loop takes 85 cycles.

Assuming that a larger number of registers is available, the code can be scheduled following the kernel shown in Figure 6. The initiation interval, II, has been reduced to 4, but each iteration spans across three repetitions of the kernel; this implies a longer lifetime of registers. The execution of the `i` loop requires 18 repetitions of the kernel, four of which form a start-up part (6 cycles) and a wind-down part (8 cycles). Thus, the execution time is $14 \times 4 + 6 + 8 = 70$ cycles.

Since scheduling with $II = 3$ is not possible because of the branch delay of 3 cycles, the performance cannot be improved by reducing II further. However, we can consider a high-level transformation, *unroll and jam* [11], for improving performance further. Thus, unrolling the `j` loop once (by replicating the `i` loop) and fusing the two inner loops (by adjusting the array subscripts appropriately), the body of the `i` loop contains twice the num-

cycle	Functional Unit				
	1	2	3	4	5
x	1_i	6_{i-1}	2_i	13_i	14_i
$x+1$	8_{i-1}	18_{i-1}	25_i	3_i	15_i
$x+2$	9_{i-1}	20_{i-1}	26_i	7_i	19_i
$x+3$	21_{i-1}	23_{i-1}	10_{i-1}	11_{i-1}	—
$x+4$	22_{i-1}	16_i	4_i	12_{i-1}	—
$x+5$	24_{i-1}	17_i	5_i	—	—

Figure 7. Scheduling two iterations at a time.

Approach	Function		Whole
	cycles	speed-up	Program
Sequential	336	1.00	1.00
VLIW (trivial)	85	3.95	2.00
VLIW (optimized)	70	4.80	2.13
VLIW (+ high-level)	51	6.59	2.32
Optimum	45	7.47	2.38

Table 1. Performance of various approaches.

ber of the original instructions. This makes a total of 26 instructions (instructions `13` and `14` are needed only once at the end of the loop body). These instructions can be scheduled as shown in Figure 7. 17 repetitions of the kernel are required. Thus, the execution of two iterations of the `j` loop takes 102 cycles, or 51 cycles per iteration.

The performance results obtained are summarised in Table 1. The first column shows the approach followed for running the code fragment shown in Figure 2. The second column shows the number of cycles taken to run the code for each approach, while the third column shows the speed-up achieved over the sequential version. Finally, the fourth column shows the whole program speed-up, assuming that the particular code fragment examined constitutes 67% of the program's execution time. It can be seen that by integrating high and low-level compiler techniques, we are able to obtain an implementation that is just 6 cycles longer than the ideal.

5. CONCLUSION

This paper briefly described a novel compiler infrastructure targeted at efficient implementation of embedded multimedia applications for VLIW architectures. A case study was presented which highlights the need for both high-level program restructuring and low-level software pipelining based on modulo scheduling. Our next goal is to develop an accurate cost model to drive the selection of program transformations based on information available from previous program executions.

Acknowledgements

The authors would like to thank their OCEANS project partners for providing useful information during the preparation of this paper.

References

[1] Edward A. Lee. Programmable DSP Architectures. *IEEE ASSP Magazine*. Part I: October 1988, pp. 4–19; Part II: January 1989, pp. 4–14.

[2] M. A. R. Saghir, P. Chow, and C. G. Lee. Towards Better DSP Architectures and Compilers. *Proceedings of the 5th International Conference on Signal Processing Applications and Technology*, October 1994, pp. I-658–I-664.

[3] V. Živojnović, Compilers for Digital Signal Processors, *DSP & Multimedia Technology Magazine*, 4(5), July/August 1995, pp. 27–45.

[4] M. A. R. Saghir, P. Chow, and C. G. Lee. Application-Driven Design of DSP Architectures and Compilers. *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1994, pp. II-437–II-440.

[5] B. Aarts, *et al.* OCEANS: Optimizing Compilers for Embedded Applications. Submitted to *EuroPar'97*.

[6] 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/

[7] G. Böckle. Exploitation of Fine-Grain Parallelism. *Lecture Notes in Computer Science 942*. Springer-Verlag, 1995.

[8] G. Araujo *et al.* Challenges in Code Generation for Embedded Processors. In *Code Generation for Embedded Processors*. Kluwer Academic Publishers, pp. 49–64, 1995.

[9] D. F. Bacon, S. L. Graham, O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4), Dec. 1994, pp. 345–420.

[10] B. R. Rau. Iterative Modulo Scheduling. *International Journal of Parallel Processing*, 24(1), Feb. 1996.

[11] S. Carr, K. Kennedy. Improving the ratio of memory operation to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6), Nov. 1994, pp. 1768–1810.