EFFICIENT IMPLEMENTATION OF THE ROW-COLUMN 8×8 IDCT ON VLIW ARCHITECTURES*

Rizos Sakellariou[1], *Christine Eisenbeis*[2], *and Peter Knijnenburg*[3]
[1] Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, U.K.
[2] INRIA, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France.
[3] Department of Computer Science, Leiden University, P.O. Box 9512,
2300 RA Leiden, The Netherlands.

ABSTRACT

This paper experiments with a methodology for mapping the 8×8 row-column Inverse Discrete Cosine Transform on general-purpose Very Long Instruction Word architectures. By exploiting the parallelism inherent in the algorithm, the results obtained indicate that such processors, using sufficiently advanced compilers, can provide satisfactory performance at low cost without need to resort to special-purpose hardware or time-consuming hand-tuning of codes.

1 INTRODUCTION

The Discrete Cosine Transform (DCT) is one of the most widely used techniques for image compression and, over the last years, significant research effort has been spent into finding fast algorithms and designing special-purpose chips to implement this. However, the performance advantage of the latter comes at an extra cost, which may be prohibitive for an end-product to be successful into the ever-competitive multimedia market. On the other hand, as processor cost drops, it becomes more attractive to use a general-purpose processor for DSP and multimedia applications rather than specially designed hardware.

Very Long Instruction Word (VLIW) processors provide a cost-effective solution as they can deliver potentially high performance, due to multiple functional units operating in parallel, and are relatively cheap to manufacture due to the simplicity of their architecture. These characteristics make them particularly suitable for embedded systems. Multimedia technologies are typical of the growing importance of the latter; as a result, several manufacturers are investing into VLIW technology, and the first VLIW chips targeting multimedia applications have already appeared in the market [3].

This paper describes research motivated by the work carried out under the ESPRIT project OCEANS [1], whose aim is to investigate and develop advanced compiler infrastructure for embedded VLIW processors targeting at multimedia applications. Such a compiler can use a number of program restructuring techniques to generate efficient VLIW code. In order to provide an indication of the performance that can be obtained using such compilation techniques, this paper considers their application to the Inverse DCT (IDCT). After a brief introduction to the IDCT and VLIW architectures, Section 4 describes a standard methodology for mapping the IDCT on a VLIW architecture. Using this methodology, experiments are carried out to obtain the performance that can be obtained on different configurations.

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

2 THE 8×8 ROW-COLUMN IDCT

Both the MPEG and JPEG standards apply DCT to transform 8×8 blocks of pixels from the spatial domain to the frequency domain. When decoding compressed images, the reverse process, i.e., the Inverse DCT (IDCT), is applied. For an 8×8 block this is defined as

$$f(x,y) = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v) \cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16},$$

where x, y are spatial coordinates in the image block, u, v are coordinates in the DCT coefficient block, and

$$C(u), C(v) = \begin{cases} 1/\sqrt{2}, & \text{for } u, v = 0, \\ 1, & \text{otherwise.} \end{cases}$$

Two basic methods can be used to compute the above two-dimensional IDCT: either compute an one-dimensional IDCT first along the rows, after along the columns, or work directly on the entire two-dimensional data set. Although the latter, direct approach may result in fewer multiplications and additions [4], corresponding algorithms have a rather complex structure and seem not to be any faster in practice. As a result, the software codification of the MPEG standard [9] implements the IDCT using the row-column approach. The one-dimensional IDCT of the 8 data elements input

```
 1:  x0 = Y0 << 11        32:  x6 = x5 + x7
 2:  x0 = x0 + 128        33:  x5 = x5 - x7
 3:  x1 = Y4 << 11        34:  x7 = x8 + x3
 4:  x2 = Y6              35:  x8 = x8 - x3
 5:  x3 = Y2              36:  x3 = x0 + x2
 6:  x4 = Y1              37:  x0 = x0 - x2
 7:  x5 = Y7              38:  x2 = x4 + x5
 8:  x6 = Y5              39:  x2 = 181 * x2
 9:  x7 = Y3              40:  x2 = x2 + 128
10: temp= x4 + x5         41:  x2 = x2 >> 8
11:  x8 = w7 * temp       42:  x4 = x4 - x5
12: temp= w1m7 * x4       43:  x4 = 181 * x4
13:  x4 = x8 + temp       44:  x4 = x4 + 128
14: temp= w1p7 * x5       45:  x4 = x4 >> 8
15:  x5 = x8 - temp       46:  y0 = x7 + x1
16: temp= x6 + x7         47:  y0 = y0 >> 8
17:  x8 = w3 * temp       48:  y1 = x3 + x2
18: temp= w3m5 * x6       49:  y1 = y1 >> 8
19:  x6 = x8 - temp       50:  y2 = x4 + x0
20: temp= w3p5 * x7       51:  y2 = y2 >> 8
21:  x7 = x8 - temp       52:  y3 = x8 + x6
22:  x8 = x0 + x1         53:  y3 = y3 >> 8
23:  x0 = x0 - x1         54:  y4 = x8 - x6
24: temp= x3 + x2         55:  y4 = y4 >> 8
25:  x1 = w6 * temp       56:  y5 = x0 - x4
26: temp= w2p6 * x2       57:  y5 = y5 >> 8
27:  x2 = x1 - temp       58:  y6 = x3 - x2
28: temp= w2m6 * x3       59:  y6 = y6 >> 8
29:  x3 = x1 + temp       60:  y7 = x7 - x1
30:  x1 = x4 + x6         61:  y7 = y7 >> 8
31:  x4 = x4 - x6
```

Figure 1: Three-address code for computing the one-dimensional IDCT along one row of 8 data elements.

sequence, defined as

$$f(j) = \sum_{k=0}^{7} C(k)F(k)\cos\frac{(2j+1)k\pi}{16},$$

is coded based on the method described in [10].

The version of the code used for computing the one-dimensional IDCT along one row is shown in Figure 1; this is presented in the form of *three-address code*, where each instruction contains a result and a maximum of two source operands.[1] The values of `Y0`, `Y1`, ..., `Y7` denote the data input set and the values of `y0`, `y1`, ..., `y7` the corresponding data output set. The values of `w1`, `w2`, ..., `w7` are equal to $2048\sqrt{2}\cos(i\pi/16)$, $i = 1, 2, \ldots, 7$, and the values of `w1m7`, `w1p7`, `w3m5`, `w3p5`, `w2p6`, `w2m6` correspond to `w1-w7`, `w1+w7`, `w3-w5`, and so on. Finally, note that instructions 4 to 9 have been added to the code for clarity; they can be eliminated by replacing the first occurrence of the corresponding left-hand side variable appropriately. In order to execute the whole 8×8 row-column IDCT, the above is repeated 8 times (one for each row), followed by the code for computing the one-dimensional IDCT along one column also repeated 8 times (one for each column). The latter, albeit similar in nature with the code shown in Figure 1, requires an extra 6 shift and 3 addition operations.

[1] This is only one of the possible implementations of IDCT.

3 VLIW ARCHITECTURES

In recent years, microprocessors are designed in a way that exploits aggressively the parallelism inherent in computer programs. This parallelism comes from two main sources. First, there exists parallelism in the execution of each instruction. The execution is divided into several stages and these stages are overlapped. This gives rise to *pipelined execution* of instructions [6]. RISC processors are the primary example of this kind of pipelined processor. Second, there exists parallelism between different instructions. Two instructions are *independent* if the order they are executed does not affect the program output. Independent instructions may be executed in parallel.

There exist two approaches of detecting whether two instructions are independent or not. In the first case, the hardware takes the responsibility for this. Processors in this class are called *superscalar* [7]; Well-known examples are the Pentium, PowerPC, MIPS R10000, DEC Alpha and HP-PA. In the second case, the compiler is responsible for detecting independent operations and packing them into a single wide instruction. These processors are called *Very Long Instruction Word (VLIW)* processors [5]. One VLIW instruction contains a fixed number of issue slots. Each such slot may contain one conventional instruction. The processor contains several functional units each of which is capable of executing one operation (possibly of a restricted class). The advantage of VLIW over superscalar processors is that the instruction issuing logic of the former is much simpler.

Recently, a powerful VLIW processor, the TriMedia TM-1000 [3], has been released by Philips. The TriMedia is a state-of-the-art general-purpose microprocessor that has been enhanced to boost multimedia performance. At the center 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 coprocessor, 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 32KB of instruction cache memory and 16KB of data cache memory.

4 MAPPING THE IDCT ON VLIW ARCHITECTURES

In order to transform sequential code into a VLIW form we have to consider dependences between instructions that place constraints on their execution order. For the code shown in Figure 1 these are illustrated by means of the dataflow graph shown in Figure 2. Each node of the graph corresponds to one instruction of the code. Each edge of the graph represents the flow of data be-

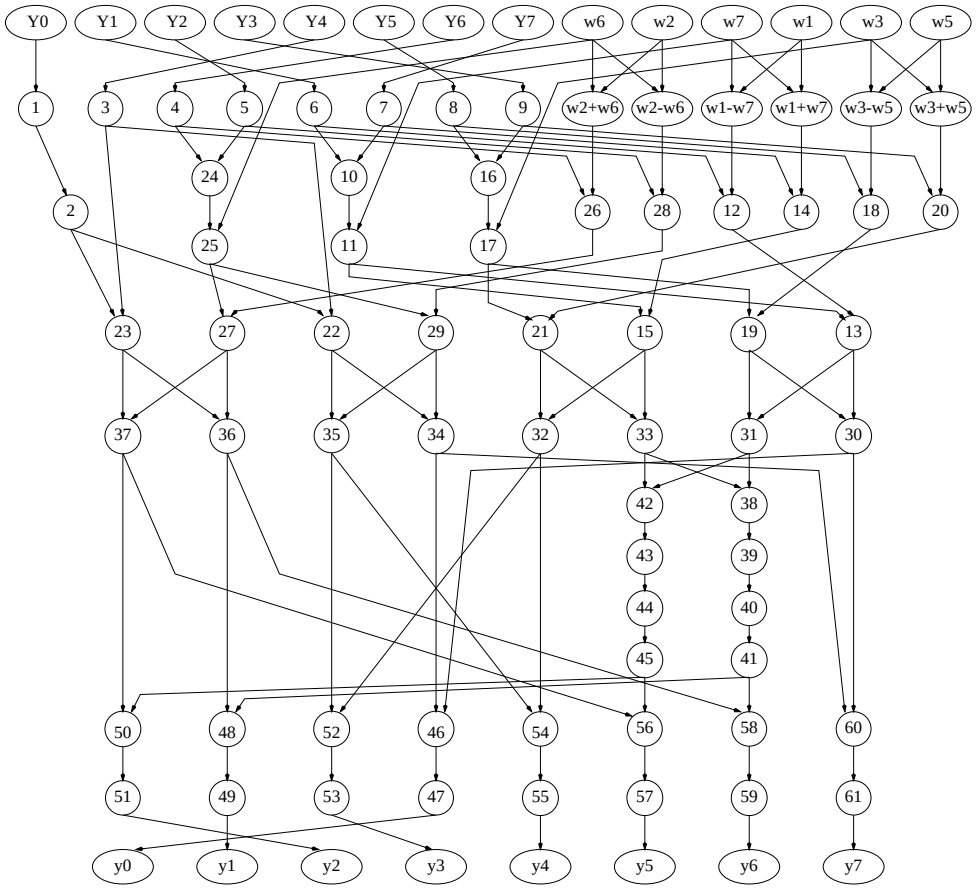

Figure 2: Dataflow graph of the code in Figure 1 showing dependences between instructions.

tween different instructions. Clearly, the execution of an instruction can begin only when all necessary data is available.

Apart from dependence constraints, architectural constraints may also exist restricting the number and the type of instructions that can be executed in parallel. Assuming an infinite number of resources, the number of clock cycles needed to execute the code would be determined by the path (from a Y_i to a y_j) with the maximum number of cycles as resulting by summing the cost (in cycles) for executing each instruction (instruction latency); this is the *critical path* that can be considered as a lower bound of the performance of the code with regard to dependence constraints. Alternatively, if up to n instructions can be scheduled in parallel in one cycle, a lower bound of the performance with regard to resource constraints, is obtained by $\lceil s/n \rceil$, where s is the total number of instructions in the code. For instance, assuming a latency of 1 cycle for additions and shift operations and 3 cycles for multiplications (i.e., corresponding to the cost of these instructions in [3]), the critical path of the dataflow graph shown in Figure 2 is 14 cycles (which corresponds to 6 additions/subtractions, 2 multiplications and 2 shift operations; check, for example, the path 16-17-19-31-38-39-40-41-58-59) and there is a total of 55 instructions. Note, that it is further assumed that instructions 4 to 9 are eliminated. Similarly, in the code used for computing the one-dimensional IDCT along one column the critical path is 15 cycles and there are 64 instructions.

Taking into account all constraints, the problem that needs to be addressed in order to produce efficient VLIW code is how to schedule the dataflow graph such that the total number of cycles needed to execute the graph is minimised. Techniques for solving this problem have been studied since long time [8]. In this paper, our approach follows a technique known as list scheduling. Each node of the graph is assigned a weighting value which is equal to the maximum number of cycles needed to compute an output value from this node, assuming unlimited resources and that all data from other instructions is available; clearly, the weighting value is determined purely by instruction latencies. Scheduling starts from the node with the highest weight. Then, the node with the second highest weight is examined. This can be scheduled to the first available slot after the execution of all sources of dependence (if there are any) has finished. This process is repeated for all nodes.

	u	seq.	2	3	4	5	6	7	8	9	10	15	20	25	30	35	40
IDCT-row	1	77	28	19	14	14	14	14	14	14	14	14	14	14	14	14	14
	2	154	55	37	28	22	19	16	14	14	14	14	14	14	14	14	14
	4	308	110	74	55	44	37	32	28	25	22	15	14	14	14	14	14
	8	616	220	147	110	88	74	63	55	49	44	30	22	18	15	14	14
IDCT-col	1	86	32	22	16	15	15	15	15	15	15	15	15	15	15	15	15
	2	172	64	43	32	26	22	19	16	15	15	15	15	15	15	15	15
	4	344	128	86	64	52	43	37	32	29	26	18	15	15	15	15	15
	8	688	256	171	128	103	86	74	64	57	52	35	26	21	18	15	15
IDCT	1	1304	480	328	240	232	232	232	232	232	232	232	232	232	232	232	232
	2	1304	476	320	240	192	164	140	120	116	116	116	116	116	116	116	116
	4	1304	476	320	238	192	160	138	120	108	96	66	58	58	58	58	58
	8	1304	476	318	238	191	160	137	119	106	96	65	48	39	33	29	29

Table 1: Number of cycles of the generic VLIW schedule for a number of functional units and unrolling factor.

5 EXPERIMENTAL RESULTS

The above methodology has been applied to generate a schedule for the row-column 8×8 IDCT on a generic VLIW architectural model with a varying number of functional units. The instruction latencies assumed were those used above, that is, 1 cycle for additions and shift operations and 3 cycles for multiplications. The number of cycles of the VLIW schedule in each case is shown in Table 1. The results are arranged in three groups: the first (IDCT-row) refers to the number of cycles needed to schedule the code for the one-dimensional IDCT along one (or more) row; the second (IDCT-col) refers to the number of cycles needed to schedule the code for the one-dimensional IDCT along one (or more) column; and the third refers to the number of cycles needed to schedule the whole row-column 8×8 IDCT. Within each group there are four lines each of which refers to the results obtained when *loop unrolling* with a varying unrolling factor of u [2] has been applied to the loop surrounding the respective code in order to enhance the parallelism available (i.e., by exploiting parallelism between computations on different rows or columns).

As can be seen from the table, without loop unrolling (i.e., $u = 1$) the results are quickly bound from the critical path and no performance improvements occur for more than 5 functional units; this leads to a maximum speed-up of 5.62 over the sequential execution. Conversely, if both loops are fully unrolled, a maximum speed-up of 44.96 on 35 functional units is obtained. Using a smaller number of functional units the performance is typically bound by resource constraints and high slot occupancy is achieved. However, it is noted that our implementation has assumed that all data elements are loaded in registers and that there are no conflicts between them. This implies that more registers than program variables are present.

6 CONCLUSION

This paper has considered an approach for implementing the 8×8 row-column IDCT on VLIW architectures and experimented with the performance that can be obtained using different configurations. The results indicate that these architectures have the potential of delivering performance, which, assuming infinite resources, is bound only by the inherent characteristics of the algorithm. This implies that on VLIW architectures the decisive factor for an efficient IDCT algorithm should be the critical path of the corresponding dataflow graph rather than the total number of arithmetic operations in the algorithm, a criterion typically used in the past. Thus, for these platforms, more suitable algorithms than the one used in this paper may exist.

References

[1] B. Aarts, *et al.* OCEANS: Optimizing Compilers for Embedded Applications. *Proceedings of EuroPar'97, Lecture Notes in Computer Science 1300*, Springer-Verlag, 1997, pp. 1351–1356.

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

[3] 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/

[4] N. I. Cho and S. U. Lee. A fast 4×4 DCT algorithm for the recursive 2-D DCT. *IEEE Transactions on Signal Processing*, 40(9), Sep. 1992, pp. 2166–2173.

[5] H. Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley, 1997.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[7] W. M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[8] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett. Local Microcode Compaction Techniques. *ACM Computing Surveys*, 12(3), Sep. 1980, pp. 261–294.

[9] MPEG Software Simulation group; see http://www.mpeg.org/

[10] Z. Whang. Fast Algorithms for the Discrete W Transform and the for the Discrete Fourier Transform. *IEEE Transactions on Acoustics Speech Signal Processing*, 32(4), Aug. 1984, pp. 803–816.