Register Allocation for Common Subexpressions in DSP Data Paths

Rainer Leupers
University of Dortmund
Dept. of Computer Science 12
D-44221 Dortmund, Germany
Rainer.Leupers@cs.uni-dortmund.de

Abstract— This paper presents a new code optimization technique for DSPs with irregular data path structures. We consider the problem of generating machine code for data flow graphs with common subexpressions (CSEs). While in previous work CSEs are supposed to be strictly stored in memory, the technique proposed in this paper also permits the allocation of special purpose registers for temporarily storing CSEs. As a result, both the code size and the number of memory accesses are reduced. The optimization is controlled by a simulated annealing algorithm. We demonstrate its effectiveness for several DSP applications and a widespread DSP processor. [1]

I. Introduction

More and more embedded systems with DSP functionality are based on programmable DSP processors. While a processor based design style benefits from high flexibility and opportunities for reuse, software development for DSPs still suffers from the fact, that there is no adequate tool support by C compilers. Since DSPs are tuned for compute-intensive applications, they often show an irregular data path structure with different functional units and special-purpose registers. An example is given in fig. 1. The Texas Instruments TMS320C25 is a widespread DSP, whose data path comprises a multiplier, an ALU, three special-purpose registers TR, PR, and ACCU, and a data memory (MEM).

Such domain-specific architectures pose problems for C compilers, since special constraints on code selection and register allocation have to be taken into account during code generation. In contrast to regular RISC-like data path structures, arguments for operations executed on functional units have to reside in particular registers, and extra instructions may be required for moving values between those registers. In order to illustrate this, consider the subset of available 'C25 machine instructions in fig. 2. Multiplier and ALU results have to be stored in registers PR and ACCU, respectively, and one multiplier argument has to reside in register TR. Additionally, memory access is quite restricted. For instance, PR cannot be directly loaded from memory, and register TR cannot be directly stored.

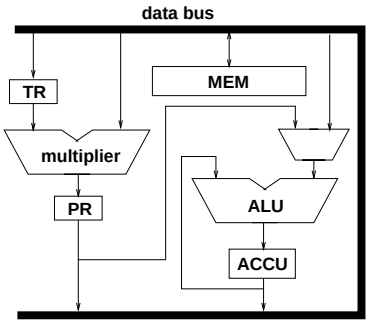

Fig. 1. *TMS320C25 data path structure*

```
"lac":     ACCU := MEM
"addk":    ACCU := ACCU + CONSTANT
"add":     ACCU := ACCU + MEM
"pac":     ACCU := PR
"apac":    ACCU := ACCU + PR
"mpy":     PR := TR * MEM
"lt":      TR := MEM
"sacl":    MEM := ACCU
"spl":     MEM := PR
```

Fig. 2. *TMS320C25 data path instructions*

Classical compiler technology can hardly cope with such irregularities. As a consequence, the machine code generated by many current DSP compilers is of unacceptable quality [1], and the largest part of DSP software still has to be written manually in assembly languages. Recent research efforts aim at eliminating this significant productivity bottleneck in embedded system design by investigating novel DSP-specific code generation and optimization techniques [2]. Such techniques often exploit the fact, that high compilation speed is not a major issue for DSP compilers, so that comparatively time-intensive optimizations may be used.

This paper presents a new code optimization technique that falls into this category. It uses *simulated annealing*

[1]Publication: ASP-DAC 2000, Yokohama/Japan, ©2000 IEEE

to perform an optimized register allocation for *common subexpressions* during code generation for irregular DSP data paths. The code quality gain achieved by this technique is measured both in terms of code size reduction and reduction of memory accesses and thus performance and/or power consumption.

The paper is structured as follows. In the next section, we discuss related work in the area of DSP code generation. The problem we consider is defined in section III, and the proposed optimization algorithm is described in section IV. An experimental evaluation for several DSP applications is given in section V.

II. Related work

We consider the problem of generating sequential assembly code for basic blocks in a program. Basic blocks are commonly represented by data flow graphs (DFGs). DFG nodes represent operations, memory accesses and constants, while DFG edges denote data dependencies between operations. Common subexpressions (CSEs) are those DFG nodes with a fanout larger than one. We assume that recomputation of a CSE is always more expensive than keeping it in memory or a register.

In order to reduce the problem complexity, the classical approach to code generation [3] is to decompose a given DFG into a set of data flow trees (DFTs) by breaking the DFG at its CSEs, and communicating CSEs between the DFTs via a fixed storage component, typically the memory. We call the successors of a CSE in a DFG the *CSE uses*. An example is given in fig. 3. There is one CSE "a * b" (variables are assumed to reside in memory) with two uses. Breaking the DFG at the edges marked with a dot results in a set of three DFTs, for each of which code can be generated separately. The scheduling order of the different DFTs may be constrained by further dependencies, such as control or output dependencies imposed by the source program.

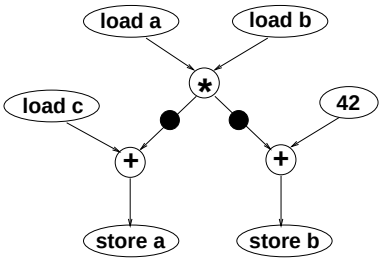

Fig. 3. *Example DFG 1*

Araujo and Malik [4] showed, how *tree pattern matching with dynamic programming* can be used to generate optimal code for DFTs in linear computation time for irregular DSP data paths satisfying certain architectural criteria. The instruction set is modeled as a tree grammar, and the code generator generator "olive" is used to generate a tree pattern matcher based on this model. In the RECORD project [5], this technique has been embedded into a user-retargetable compiler for DSPs.

Using the approach from [4], also a heuristic code generation technique for full DFGs has been proposed [6]. The basic idea is to cut a DFG with highest priority at those edges, where the CSE uses have to pass the memory anyway due to architectural constraints. As compared to an approach where all CSE uses are rigidly loaded from memory, this technique frequently yields higher code quality by avoiding some redundant loads and stores. However, it still requires that all CSEs are stored to memory and all but one CSE uses are loaded from memory, which is not always necessary. The same limitation holds for the binate covering formulation given in [7], which presents an exact technique for mapping DFGs to DSP data paths.

A further approach is presented in [8], which uses a branch and bound algorithm for DFG code generation. However, unlike [6] and [7], it separates detailed register allocation from code selection. Thus, for architectures with single special-purpose registers (instead of register files) like the one shown in fig. 1, it may generate inferior code in presence of CSEs. The tree pattern matching approach to code generation has recently been extended to full DFGs [9], but due to several constraints on the underlying tree grammars so far only regular data paths can be handled.

The idea of temporarily storing CSEs in registers instead of the memory has also been implemented in a code generator based on *constraint logic programming* [10]. However, that approach still neglects the mutual dependence between CSE register allocation and DFT scheduling which will be illustrated in the following section.

III. Problem definition

For sake of easier illustration, we will refer to the 'C25 data path and instruction set (figs. 1 and 2) in the following. The 'C25 data path has four possible locations $\{MEM, ACCU, PR, TR\}$ to hold CSEs. Given a DFG with k CSEs $C = \{c_1, \ldots, c_k\}$, a *CSE register allocation* is a mapping

$$K : C \rightarrow \{MEM, ACCU, PR, TR\}$$

which assigns each CSE to one of the possible locations. The mapping $K(c_i) = L$ implies, that CSE c_i is stored to location L and all uses of c_i also read the CSE value from L.

Note that not all CSE register allocations are valid. If, for instance, a CSE c_i is assigned to a register, and some instruction writes to that register before all uses of c_i have been scheduled, then a conflict is exposed and a different CSE location (e.g. MEM) has to be selected. However, there is no need to give priority to MEM as a CSE location, but there are good reasons to use registers (ACCU,

PR, TR) instead:

1) Reading a CSE from a register can result in lower **code size** and higher **performance** since the number of data moves between registers and memory might be reduced.

2) If external memory instead of on-chip memory is used, then a memory access is generally **slower** than a register access.

3) Memory accesses typically consume **more power** than register accesses, since they cause signal transitions on address and data busses.

4) Additional instructions may be required to compute **memory addresses** for CSE loads and stores.

Our goal is to determine the CSE register allocation which, among all valid allocations, results in the minimum cost machine code for an entire DFG. The cost metric is given by two components. The *cover costs* of a DFG is the sum of the cost values of all selected instructions. The cost value of a single instruction, which is supposed to be given, may represent its size, execution time, or power consumption.

Additionally, we estimate the *addressing costs* implied by a certain CSE register allocation. This cost value reflects the fact, that the addresses of CSEs stored in MEM, which are normally allocated in the stack frame of a function, might need to be computed by extra instructions.

A small example illustrates that holding CSEs in registers rather than only in memory may yield better code. The 'C25 assembly code in fig. 4 implements the DFG from fig. 3. In the left column, the CSE "a * b" is assigned to a memory cell "temp", which leads to a total of 9 instructions. In the right column, the CSE has been assigned to PR, which saves one instruction and two memory accesses. One can also find DFG structures, where keeping CSEs in registers ACCU or TR is favorable.

```
lt a          lt a
mpy b         mpy b
spl temp      pac
pac           add c
add c         sacl a
sacl a        pac
lac temp      addk 42
addk 42       sacl b
sacl b
```

Fig. 4. *Alternative assembly codes for DFG 1*

It is important to observe the mutual dependence between CSE register allocation and scheduling. This is exemplified by the DFG in fig. 5, which after breaking it at its CSEs consists of three DFTs T_1, T_2, and T_3. Due to dependencies, both T_2 and T_3 have to be scheduled after T_1, but the scheduling order of T_2 and T_3 is arbitrary. Let the CSE "a * b" be assigned to PR. Since T_2 comprises

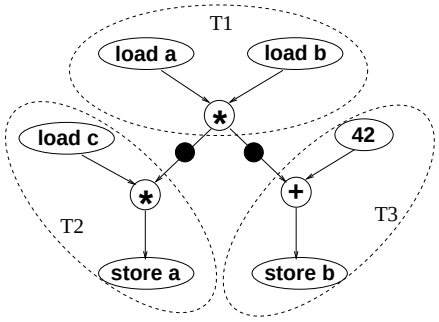

Fig. 5. *Example DFG 2*

another multiplication, PR would be overwritten before the second CSE use in T_3 has been scheduled, resulting in a conflict and thus in an invalid allocation. However, if T_3 is scheduled before T_2, then the CSE in PR is still available when scheduling T_2. Thus, optimizing the CSE register allocation obviously requires to take into account alternative DFG schedules. This is accomplished by the technique described in the following section.

IV. Problem solution

Due to the generally large number of alternative CSE register allocations and DFG schedules, we use simulated annealing (SA) to approach a global optimum. SA is a well-tried technique for complex optimization problems since it is capable of skipping local minima in the objective function. The main algorithm is outlined in fig. 6. It reads a DFG, globally optimizes the CSE register allocation while trying alternative schedules, and finally emits assembly code for the DFG.

First, the input DFG G is decomposed into a set of DFTs by breaking it at its k CSEs and inserting dedicated CSE write and read nodes. The modified DFG is called G'. During the optimization, the current location (MEM, ACCU, PR, or TR) of each CSE is kept in the array "CSE_alloc". Initially, all CSEs are assigned to MEM, the initial costs are computed, and the starting temperature is set[2]. The outer while-loop is executed until the temperature is "frozen". The inner for-loop is executed for a fixed number of times.

In each iteration of the inner loop, function DoModification modifies the current solution in one of the two following ways (with equal probability):

- The DFG G' is modified by adding or removing (each with a probability of 0.5) a random *sequencing edge* between two DFTs in G'. Naturally, dependency edges originally present in the input DFG are unremovable, in order to preserve correctness of the resulting code. Likewise, additional sequencing edges

[2]In fig. 6, the SA parameters (temperature interval, iterations per temperature value, cooling factor) have been set to those values we found most appropriate during experimental evaluation.

are inserted only if they do not cause cycles in G'. These additional edges influence the scheduling possibilities for G' and thereby permit to explore alternative schedules. Since such edges may be removed again in a later step, they do not lead to an unnecessary restriction of the solution space.

- The current CSE register allocation is changed by randomly replacing one CSE location by another location, i.e., for some CSE c_i, the current mapping $K(c_i)$ is set to one element of $\{MEM, ACCU, PR, TR\}$ different from the current $K(c_i)$.

---

algorithm CSE_RegisterAllocation
input: data flow graph G with k CSEs;
output: sequential assembly code for G;
begin
  G' = Decompose(G);
  CSE_alloc[1..k] = MEM;
  best = InitialCost(G');
  temp = 50;
  while temp > 0.1 **do**
    for count = 1 **to** 10 **do**
      DoModification(G', CSE_alloc);
      schedule = TopologicalSort(G');
      cost = 0;
      for all trees T in schedule **do**
        cost += CoverCost(T);
        if RegisterConflict(T) **then** cost = ∞;
      end for
      cost += AddrCost(schedule);
      delta = cost - best;
      if delta < 0 **or** Random(0,1) < exp(-delta/temp)
      then
        best = cost;
        best_schedule = schedule;
      else UndoModification(G', CSE_alloc);
      end if
    end for
    temp = 0.9 * temp;
  end while
  for all trees T in best_schedule **do**
    EmitAssemblyCode(T);
  end for
end algorithm

Fig. 6. *Optimization algorithm*

Next, a schedule for the DFTs in G' is determined by topological sorting, and a cost value is computed. For each DFT T in G', function CoverCost returns the sum of the costs of instructions selected for T. For code generation and scheduling of single DFTs, we use the technique by Araujo and Malik [4], which is based on tree pattern matching with dynamic programming. In our approach, the underlying tree grammar model of the machine instruction set uses a *dynamic* instruction cost function, which ensures that a CSE can be read and written only from/to that location, to which it is currently assigned.

As an additional cost metric for the current schedule, function AddrCost estimates the costs of extra instructions needed for address computation for those CSEs currently assigned to MEM and for intermediate results stored in memory. Such CSEs and intermediate results can be considered as "local variables", which have to be laid out in memory. Since DSPs typically show auto-increment capabilities for address registers, the concrete memory layout of local variables plays an important role for the overall code quality. This is illustrated in fig. 7.

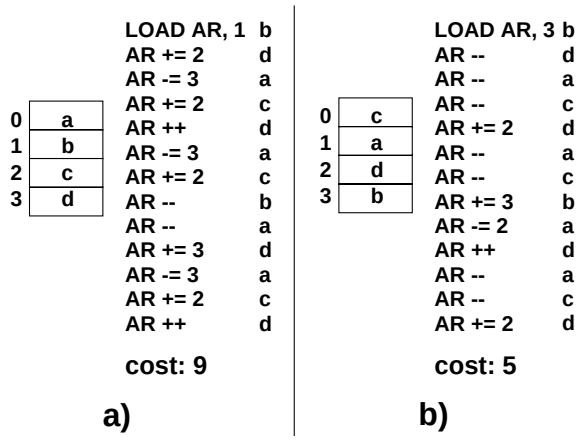

Fig. 7. *Memory layout optimization*

Suppose, a set of symbolic local variables $\{a, b, c, d\}$ is given, and the access sequence to these variables within the schedule is

$$S = (b, d, a, c, d, a, c, b, a, d, a, c, d)$$

If the variables are mapped to memory cells simply in lexicographic order (fig. 7 a), and one address register AR is used to compute the memory addresses of the sequence S, then a certain sequence of address computation instructions has to be added to the code. First, AR has to be initialized with the address of the first element in S. Then, for each subsequent variable access, AR has to be set to the next address by adding or subtracting a certain constant to/from AR.

The key idea in optimizing the memory layout is, that those address computations which modify AR by +1/-1 do not result in extra instructions, since they can be mapped to parallel auto-increment/decrement operations on AR. The goal, therefore, is to permute the local variables in memory in such a way, that the utilization of auto-increment/decrement is maximized.

Fig. 7 b) shows the sequence of address computations for a different memory layout. The number of extra machine instructions (the "cost" of the layout) has been reduced from 9 to 5, since more address computations have been implemented by auto-increment/decrement.

This effect should be taken into account in code generation for DFGs in order to achieve accurate code quality estimations. A number of different graph-based *offset assignment* algorithms for computing good memory layouts for local variables are already available. In our approach, we use the algorithm presented in [11], an improvement of the work described in [12]. An experimental evaluation of these algorithms w.r.t. code quality improvement for DSP algorithms has been given in [13].

The total costs of the schedule are given as the sum of the covering costs for the single DFTs plus the addressing costs of the complete schedule. During cost computation, function RegisterConflict checks whether the current CSE register allocation is invalid. This is the case, whenever an instruction selected for T overwrites a register containing a CSE, whose uses have not yet all been scheduled. If such a register conflict is detected, the costs are set to an "infinite" value.

If the current cost value indicates an improvement of the best solution found so far, the last modification performed is accepted. Also modifications resulting in worse solutions may be randomly accepted with a probability inversely related to the temperature. This is important to skip locally optimal solutions. If a modification of G' or "CSE_alloc" is not accepted, then the last modification is undone and the previous solution is restored. After termination of the while-loop, assembly code is emitted for all DFTs according the the optimized CSE register allocation and schedule.

V. Experimental results

For an experimental evaluation of CSE register allocation, we have implemented a TI 'C25 code generator prototype using our LANCE compiler environment [14]. The LANCE system comprises an ANSI C frontend and a library of machine-independent code optimizations, including global CSE detection. LANCE compiles ANSI C source code into an optimized intermediate representation (IR), and machine-dependent compiler backends working on the IR can be easily integrated. All LANCE tools are invoked from a common graphical user interface (fig. 8).

Using the TI 'C25 code generator, we have compiled different ANSI C source codes into TI 'C25 assembly code. The sources are taken from DSPStone [1], as well as from GSM, JPEG [15], MPEG-2 [16], and MPEG-4 packages.

For each source code, columns 3 and 4 in table I show the number of CSEs and CSE uses. Column 5 gives the number of CSEs that were assigned to registers by the algorithm in fig. 6. Column 6 shows the percentage of costs for those DFGs containing CSEs, as compared to

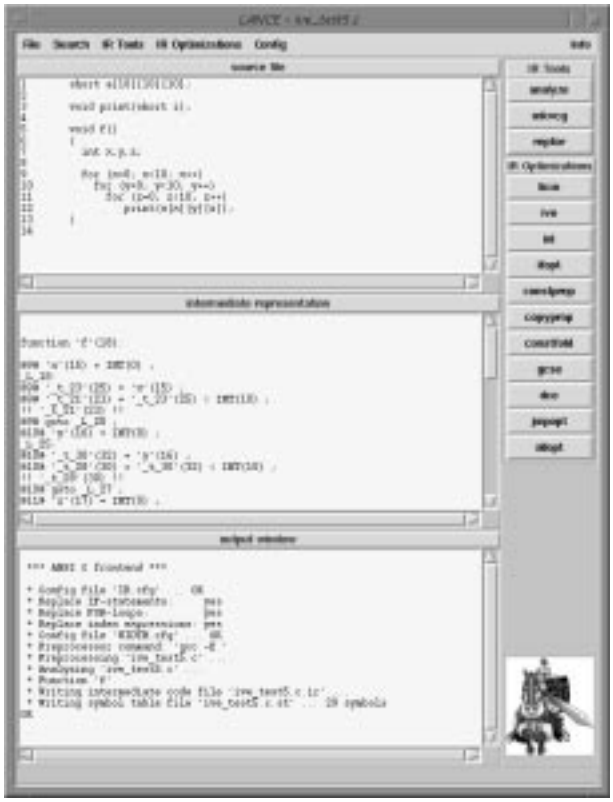

Fig. 8. *User interface of the LANCE system*

the initial solution (100 %) which only uses the memory as a CSE location. Column 7 gives the percentage of the number of memory accesses for CSEs and temporary values remaining after optimization. Finally, column 8 gives the CPU time on a Sun Ultra-1 workstation.

The cost metric we have used only reflects code size and does not penalize potentially slow or power-consuming memory accesses. Thus, the primary optimization goal was to minimize the number of instructions required for data transport in the 'C25 data path. Even though the average cost reduction achieved (7 %) seems moderate, it is important to note that for embedded systems with on-chip program memory every byte matters. For cost metrics also reflecting memory access time or power consumption, much higher cost savings are possible. This is indicated by the average reduction of memory accesses for CSEs (column 7) to only 67 %. Even when optimizing for code size, the number of memory accesses are significantly reduced as a secondary effect.

VI. Conclusions

In order to increase the quality of DSP code generated by C compilers, code optimization techniques tuned for the special data path structures of DSPs are required. Such techniques will enable the use of compilers instead of assembly programming and thus enhance the produc-

package	source	CSEs	CSE uses	reg CSEs	cost (%)	mem (%)	CPU sec
DSPStone	IIR filter 1 section	2	5	1	91	57	1
	IIR filter n sections	12	26	3	89	72	10
	FFT	13	32	2	97	95	7
	2-dim FIR filter	3	6	3	94	42	2
	ADPCM transcoder	18	36	12	93	31	19
	LMS filter	2	5	0	100	100	1
	n complex updates	12	26	2	96	89	7
GSM	basic functions	2	4	2	76	40	3
	receiver	52	113	17	95	73	34
JPEG	main buffer ctrl	3	8	3	97	10	4
	compression preproc ctrl	14	35	12	97	44	16
	IDCT	41	100	5	97	85	79
	downsampling	37	81	17	92	66	32
	transcoding compression	4	8	2	85	67	4
MPEG-2	IDCT	49	99	2	100	97	85
	motion vector decoding	24	58	7	94	80	20
	DCT block decoding	24	54	18	83	59	41
	motion comp prediction	48	127	13	100	89	38
	quantization	11	22	11	76	47	12
MPEG-4	grey scale coding	15	31	8	93	74	33
	bitstream functions	3	7	2	100	83	4
	DCT coeff quantization	18	38	12	92	64	18
average					**93**	**67**	

TABLE I
Experimental results for CSE register allocation

tivity in DSP software development. For irregular data paths, the mapping of CSEs to registers or memory plays an important role for code quality. While in previous work CSEs are supposed to be kept in memory, we have pointed out that in principle any special-purpose register may be used for storing CSEs and we have proposed a new optimization algorithm which exploits this fact to improve code quality. The efficacy has been demonstrated experimentally for a widespread DSP. We expect that comparable results can be achieved for other DSP architectures also showing irregular data paths, which, however, might require different optimization parameters.

References

[1] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[2] P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[3] A.V. Aho, R. Sethi, J.D. Ullman: *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, 1986

[4] G. Araujo, S. Malik: *Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures*, 8th Int. Symp. on System Synthesis (ISSS), 1995, pp. 36-41

[5] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997

[6] G. Araujo, S. Malik, M. Lee: *Using Register Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures*, 33rd Design Automation Conference (DAC), 1996

[7] S. Liao, S. Devadas, K. Keutzer, S. Tjiang: *Instruction Selection Using Binate Covering for Code Size Optimization*, Int. Conf. on Computer-Aided Design (ICCAD), 1995, pp. 393-399

[8] S. Hanono, S. Devadas: *Instruction Selection, Resource Allocation, and Scheduling in the AVIV retargetable code generator*, 35th Design Automation Conference (DAC), 1998

[9] M.A. Ertl: *Optimal Code Selection in DAGs*, ACM Symp. on Principles of Programming Languages (POPL), 1999

[10] S. Bashford, R. Leupers: *Constraint Driven Code Selection for Fixed-Point DSPs*, 36th Design Automation Conference (DAC), 1999

[11] R. Leupers, P. Marwedel: *Algorithms for Address Assignment in DSP Code Generation*, Int. Conf. on Computer-Aided Design (ICCAD), 1996

[12] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: *Storage Assignment to Decrease Code Size*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995

[13] A. Rao, S. Pande: *Storage Assignment using Expression Tree Transformations to Generate Compact and Efficient DSP Code*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1999

[14] LANCE Compilation Environment User's Guide, University of Dortmund, Dept. of Computer Science, WWW: LS12-www.cs.uni-dortmund.de/~leupers, 1999

[15] Independent JPEG group: URL *www.ijg.org*, 1999

[16] MPEG Software Simulation Group: URL *www.mpeg.org*, 1999