Pel Reconstruction on FPGA-Augmented TriMedia

Mihai Sima, *Member, IEEE*, Sorin D. Coţofană, *Senior Member, IEEE*, Stamatis Vassiliadis, *Fellow, IEEE*, Jos T. J. van Eijndhoven, and Kees A. Vissers

***Abstract*—This paper presents a TriMedia processor extended with three reconfigurable designs for entropy decoding (ED), inverse quantization (IQ), and two-dimensional (2-D) inverse discrete cosine transform (IDCT), and assesses the performance gain that is provided by such extensions when performing MPEG2-compliant pel reconstruction. We first describe an extension of the TriMedia architecture, which consists of a multiple-context field programmable gate array (FPGA)-based reconfigurable functional unit (RFU), a configuration unit managing the reconfiguration of the RFU, and their associated instructions. Then, we address the computation of the ED, IQ, and 2-D IDCT tasks, and propose to provide reconfigurable hardware support for a variable-length decoder that can decode two symbols per call (VLD-2), an inverse quantizer that can dequantize four coefficients per call (IQ-4), and an 1-D IDCT (1-D IDCT). The most important aspects concerning the implementation of the FPGA-mapped VLD-2, IQ-4, and 1-D IDCT units, as well as the organization of the software routines calling these FPGA-mapped computing units are outlined. Experimental results indicate that by configuring each of the VLD-2, IQ-4, and 1-D IDCT units on a different FPGA context, and by activating the contexts as needed, the FPGA-augmented TriMedia can perform MPEG2-compliant pel reconstruction with an average speed-up of 1.4 × over the standard TriMedia.**

***Index Terms*—Field programmable gate arrays (FPGAs), MPEG2 decoding, reconfigurable computing, very long instruction word (VLIW) processors.**

I. Introduction

PEL reconstruction constitutes a computationally-intensive stage of video compression standards, e.g., MPEG [1]. Traditionally, it has been implemented in application-specific integrated circuits (ASIC) or in hardwired assists for application-specific instruction processors (ASIP). Due to the lack of flexibility of these fixed-function devices, a different full-custom implementation is needed for each particular task. On the other side, a programmable computing platform allows functions to be implemented in software rather than in custom hardware. This dramatically reduces the development cost and time-to-market versus the traditional fixed-function design approach, and ensures that a single device can be applied in a range of different products and adapt to quickly evolving standards in the media domain.

In this paper, we propose a reconfigurable pel reconstruction design for a field-programmable custom computing machine composed of the 64-bit instance of TriMedia [2] augmented with a reconfigurable array. That is, we disclose a compound consisting of user-defined computing units, which are mapped on the reconfigurable hardware, and software routines, which include calls to these computing units.

We first consider the TriMedia processor extended with a multiple-context field-programmable gate array (FPGA)-based reconfigurable functional unit (RFU), and a configuration unit managing the reconfiguration of the RFU. Then we address pel reconstruction, and propose a reconfigurable design for each of its most important constituents: entropy decoding, inverse quantization (IQ), and inverse discrete cosine transform (IDCT). In particular, we provide reconfigurable hardware support for a variable-length decoder that can decode two symbols per call (VLD-2), an inverse quantizer that can dequantize four coefficients per call (IQ-4), and one–dimensional (1-D) IDCT. In our decision, we took into account the bottlenecks encountered in a (highly optimized) pure-software implementation, TriMedia organization constraints, as well as the logic capacity of a hypothetical multiple-context FPGA having the architecture of the raw hardware identical to that of an ACEX EP1K100 device from Altera. Finally, we combine these reconfigurable designs into a larger one, and establish the gain in performance when performing MPEG2-compliant pel reconstruction.

Experimental results carried out on a TriMedia cycle-accurate simulator indicate that by configuring each of the VLD-2, IQ-4, and 1-D IDCT facilities on a different FPGA context, and by activating the contexts as needed, the augmented TriMedia can compute MPEG2-compliant pel reconstruction with a speed-up of 1.4 × over the standard TriMedia. Given the fact that the experimental TriMedia instance is a 5-issue slot VLIW processor with a 64-bit datapath and a very rich multimedia-oriented instruction set [2], such an improvement within its target media processing domain [3], [4] indicates that TriMedia + FPGA hybrid is a promising approach.

The paper is organized as follows. Background information regarding the MPEG standard and the architecture of the FPGA core is provided in Section II. Section III outlines the TriMedia architectural extension that incorporates support for the reconfigurable hardware. Several issues on the programming model of FPGA-augmented TriMedia are presented in Section IV.

Manuscript received July 3, 2002; revised November 18, 2003. This work was supported by the Doctoral Fellowship RWC-061-PS-99047-ps, Philips Research Laboratories, Eindhoven, The Netherlands.

M. Sima is with the Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC V8W 3P6, Canada (e-mail: msima@ece.uvic.ca).

S. D. Coţofană and S. Vassiliadis are with the Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: S.D.Cotofana@et.tudelft.nl; S.Vassiliadis@et.tudelft.nl).

J. T. J. van Eijndhoven is with the Department of Information and Software Technology, Philips Research Laboratories, 5656 AA Eindhoven, The Netherlands (e-mail: jos.van.eijndhoven@philips.com).

K. A. Vissers is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720-1774 USA (e-mail: vissers@eecs.berkeley.edu).

Digital Object Identifier 10.1109/TVLSI.2004.827594

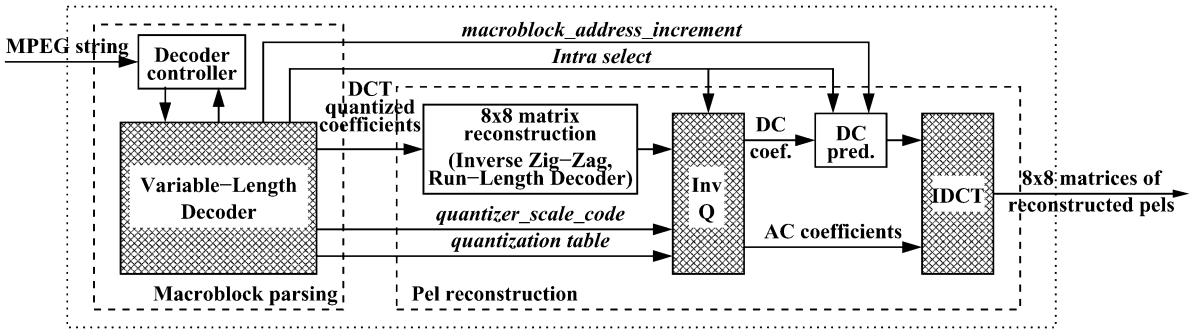

Fig. 1. The conceptual diagram of the pel reconstruction module—adapted from [5].

The VLD-2, IQ-4, and 1-D IDCT user-defined operations, as well as their FPGA-based implementations are discussed in Section V. The pel reconstruction execution scenario on both standard and extended TriMedia along with experimental results are presented in Section VI. Section VII completes the paper with some conclusions and closing remarks.

II. Background

To make the presentation self-consistent, we would like to review the principal stages of the pel reconstruction task. We also outline the architecture of the FPGA that we use as an experimental reconfigurable core.

A. Pel Reconstruction

A video coder is typically composed of a *lossy source coder*, which performs filtering, transformation, and/or quantization, and a *lossless entropy coder*, which removes the statistical dependencies remained after source coding [6]. In MPEG [5], [7], the couple DCT + Quantization is used as a lossy coding technique. The DCT algorithm processes the video data in blocks of 8×8 pixels, decomposing each block into a weighted sum of amplitudes (the DCT coefficients) of 64 spatial frequencies. Since the human eye cannot readily perceive high frequency activity, a quantization step is carried out to force as many DCT coefficients as possible to zero within the boundaries of a prescribed video quality. Then, a zig-zag operation transforms the matrix into a vector which contains large series of zeros. This vector is further compressed by an entropy coder which consists of a run-length coder (RLC) and a variable-length coder (VLC). The RLC represents consecutive zeros by their run length, and generates *run-level* pairs. The *run* value indicates the number of zeros that a (nonzero) DCT coefficient is preceeded. The *level* value represents the value of the DCT coefficient. When all the remaining coefficients in a vector are zero, they are all coded by the special symbol *end-of-block*. The variable-length coder maps the *run-level/end-of-block* symbols to *variable length codewords* according to the VLC Tables B12, B13, B14, and B15 defined by the MPEG2 standard [1].

From the syntax point of view, an MPEG video sequence is structured hierarchically on layers, each layer providing a wrapper around the encompassed layer. A *video sequence* includes a series of *Groups of Pictures* (GOPs). A GOP is divided into a series of *pictures* (frames), which begins with an Intra-coded picture (I-picture) followed by an arrangement of Forward Predictive-coded pictures (P-pictures), and Bidirectionally Predicted pictures (B-pictures). A picture is further subdivided into *slices*. A slice is composed of a series of *macroblocks*, and a macroblock is composed of 6 or fewer *blocks* (4 for luminance and 2 for chrominance[1]) and possibly motion vectors.

In this paper, we will focus on the video decoding, i.e., operation inverse to video coding. Four major stages can be distinguished in MPEG decoding: entropy decoding (which is composed of variable-length decoding, inverse zig-zag, and run-length decoding), IQ, IDCT, and motion compensation. Since motion compensation is a memory-dominant task, the required arithmetic being a simple addition per pixel, it is likely not to be subject for acceleration by means of reconfigurable logic. Thus, all the above mentioned stages but motion compensation are considered during the subsequent experiment. The joined task of these stages is generally referred to as *Pel Reconstruction* [5], which is outlined subsequently.

The pel reconstruction process is depicted in Fig. 1. First, the headers at video sequence layer down to macroblock layer are decoded and various symbols are extracted: *decoding parameters*, e.g., *macroblock_address_increment*, *quantizer_scale_code*, *intra_dc_precision*, and *motion values*. The motion values are used by the motion compensation process which is not considered here. However, since these values are decoded during header parsing, the overhead associated with the decoding of the motion values will be taken into consideration in the subsequent experiment. After header parsing, the MPEG string still contains *composite symbols* (*run-level* pairs and *end_of_block*), which are decoded by the variable-length decoder (VLD). Then, the run-length decoder (RLD) recreates the 8×8 matrices that include DCT quantized coefficients. Next, using a quantization table and a *quantizer_scale*, an IQ is performed on each DCT coefficients. Finally, after the dc prediction unit reconstructs the dc coefficient in intracoded macroblocks, an IDCT is carried out.

In connection with Fig. 1 and the subsequent experiment, we would like to mention that the variable-length decoder, inverse quantizer, and IDCT will benefit from reconfigurable hardware support. Next, we will outline the architecture of the FPGA that we use as an experimental reconfigurable core.

[1]Luminance is the monochrome representation of the signal, while chrominance provides the color information for the video.

B. The Experimental FPGA Architecture

Field-programmable gate arrays (FPGA) [8] are devices that can be configured *in the field* by the end user. In essence, an FPGA is composed of two constituents: *raw hardware* and *configuration memory*. The information stored into the configuration memory defines the function performed by the raw hardware. Generally speaking, a multiple-context FPGA [9] has its configuration memory replicated in order to contain several configurations for the raw hardware, which are referred to as *contexts*, from which only one is active at a time. That is, a cache of contexts is available on-chip. Such a cache allows a context switch to occur on the order of nanoseconds [10]. However, loading a new configuration from off-chip is still limited by the low off-chip transfer latency, which is on the order of 50 ns/byte for Altera's FPGAs [11].

For experimental purpose, we assume that at most four contexts can be simultaneously stored on the FPGA chip. Our assumption does not violate the general accepted figures regarding multiple-context FPGAs—see for example [10], [12]. Since a multiple-context FPGA is not commercially available for the time being, we also assume a hypothetical FPGA having the architecture of the raw hardware and context reconfiguration scheme identical to those of a (single-context) ACEX 1K device from Altera [13]. Our choice could allow future single-chip integration, since both ACEX 1K and TriMedia families are manufactured in the same TSMC technological process. Briefly, an ACEX 1K device contains an array of 4-input look-up tables (LUT), a number of embedded array blocks (EAB), each EAB being a RAM block with 8 inputs and 16 outputs, and a rich interconnection network. The reconfiguration of such device can be performed according to the common *Passive Parallel Asynchronous* scheme, in which a master unit drives data to the FPGA serially, one word at a time [13]. There are no partial reconfiguration capabilities for the considered device; thus, a global reconfiguration of a context is required even for changing 1 bit of its configuration data. Since we envision that circuits without many commonalities are to be configured on the programmable array, this limitation is not a serious restriction.

Subsequently, we also assume that a context can be configured only when it is not active. Thus, the active context can continue its operation while the idle contexts are being reconfigured. As mentioned, the FPGA context switching occurs on the order of nanoseconds. Trimberger *et al.* [10] announce that a context switching can be completed in 30 ns for a 20×20 array of Logic Blocks. For an ACEX EP1K100 device that includes about 4,992 LUT's and 12 EAB's, that is, for an FPGA which is more than 15 times larger, we make a conservative assumption and consider that the context switching penality is on the order of 500 ns. The rationale behind the assumption that the reconfiguration time increases with the FPGA size is related to power consumption, which can become a concern with large FPGAs as identified by Trimberger *et al.* [10].

Section III introduces the architectural extension for the TriMedia-CPU64, which is the particular 64-bit TriMedia instance we use as an experimental platform.

III. TriMedia Architectural Extension

TriMedia-CPU64 is a processor model, whose architecture features a rich instruction set optimized for media processing [3], [4]. Specifically, it is a 5 issue-slot VLIW engine, launching a long instruction every clock cycle [2]. It has a uniform 64-bit wordsize through all functional units, the register file, load/store units, internal and external buses. Each of the five operations in a single instruction can in principle read in two register arguments and write back one register result. In addition, each operation can be guarded with the least-significant bit of a fourth register to allow for conditional execution without branch penalty. With the exception of floating point divide and square root unit, all functional units have a recovery of 1, while their latency[2] ranges from 1 to 4. The TriMedia core is assumed to support multiple-slot operations, or super-operations [14]. Such a super-operation occupies two or more adjacent slots in the VLIW instruction, and maps to a wider functional unit. This way, operations with more than two arguments and one result are possible. The architecture also supports subword (SIMD-style) parallelism on byte, half-word, or word entities. The current organization of the TriMedia-CPU64 is presented in Fig. 2.

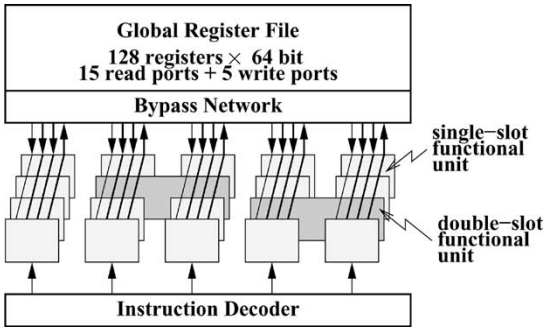

Fig. 2. The 64-bit TriMedia organization—adapted from [2].

In this paper, we propose to augment the TriMedia-CPU64 processor with a RFU consisting of a multiple-context FPGA core and its associated controller, and a configuration unit (CU) managing the reconfiguration of the FPGA. Both RFU and CU are embedded into TriMedia as any other hardwired functional unit, i.e., they receive instructions from the instruction decoder, read their input arguments from and write the computed values back to the register file, as shown in Fig. 3. This way, only minimal modifications of the basic architecture and compilation toolchain are required.

In order to use the RFU, the user is provided a kernel of new instructions: `SET_CONTEXT`, `ACTIVATE_CONTEXT`, and `EXECUTE`. This kernel constitutes the extension of the TriMedia instruction set architecture we propose. Loading context information into the FPGA configuration memory is performed by the CU under the command of a `SET_CONTEXT` instruction, while the `ACTIVATE_CONTEXT` instruction swaps the active configuration with an on-chip idle one. `EXECUTE` instructions launch the operations performed by the RFU-mapped computing units [15]. With these instructions, the user is given the freedom to define and use any computing unit subject to FPGA size and TriMedia organization.

[2]*Latency* is the number of clock cycles between the issue of an operation and availability of its results, while *recovery* is defined as the minimum number of clock cycles between the issue of successive operations.

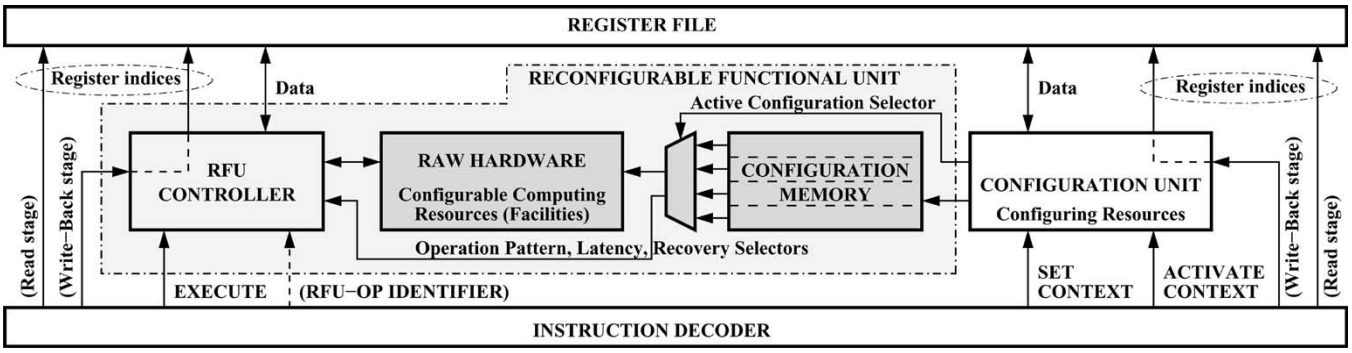

Fig. 3. TriMedia-CPU64 VLIW core extension.

Uploading configuration information to the CU is performed under the command of a double-slot instruction issued on Slot pair 1+2:

$$\mathbf{SET\_CONTEXT}(\mathbf{context})\mathbf{Rs1}, \mathbf{Rs2}, \mathbf{Rs3} \rightarrow \mathbf{Rd}$$

where `context` is the fourth (immediate) argument specifying the context to be reconfigured, while the registers Rs1, Rs2, and Rs3 contain 192 bits of configuration information. If the instruction completes successfully, then the register Rd contains 0, otherwise it contains an error code. For example, if an attempt to reconfigure the active context is made, the instruction has no effect on the configuration and returns 1.

Subject to FPGA architecture, the configuration information being uploaded by `SET_CONTEXT` instructions can be interpreted in different ways before it is sent to the RFU. Thus, different configuration patterns can be supported. For example, assuming an FPGA has partial reconfiguration capabilities (e.g., XC6200 family from Xilinx [16]), or incorporates means to reconfigure only the cells which are different from the current configuration (e.g., AT6000 family from Atmel [17]), complex reconfiguration patterns can be generated by a microprogrammable CU [18]. This case, the `SET_CONTEXT` instructions may also upload a *reconfigurable microprogram* to the CU, giving the user a large flexibility to reconfigure the FPGA.

However, for a global-reconfigurable FPGA with a serial context reconfiguration scheme, which in fact we assume in our subsequent experiment, the CU can be as simple as a parallel-to-serial converter. As mentioned in Section II-B, the average latency for loading new FPGA configuration information from off-chip is about 50 ns/byte, that is 10 cycles/byte. Since the `SET_CONTEXT` instruction places 192 bits = 24 bytes on the CU at a time, it has a latency of 240 cycles. For an EP1K100 FPGA, which has a configuration file of 1 337 000 bit [13], 6,964 `SET_CONTEXT` instructions or $6{,}964 \times 240 = 1\,671\,360$ cycles are needed to completely reconfigure a context.

For FPGA context switching, a single-slot instruction issued either on Slot 1 or Slot 2 is provided:

$$\mathbf{ACTIVATE\_CONTEXT}(\mathbf{context}) \rightarrow \mathbf{Rd}$$

where `context` is an immediate argument specifying the context that is being activated. If the context is already active, the instruction has no effect. An attempt to activate a context prior to its complete reconfiguration has also no effect on the active context, and is signaled by loading an error code into Rd. If the activation completes successfully, Rd contains 0. Given the fact that the FPGA context switching penality is 500 ns (Section II-B), the `ACTIVATE_CONTEXT` instruction has a latency of 100 TriMedia@200 MHz cycles.

Conceptually speaking, computing units of user-definable computing pattern[3] , latency, recovery, and slot-assignment[4] can be configured on RFU. Thus, the RFU can act as five independent single-slot functional units each of them executing a different custom operation, a mixture of single- and multiple-slot functional units, or even a five-slot functional unit. In all these situations, the RFU may receive `EXECUTE` instructions issued on any of the five TriMedia slots, and use all 10 read and 5 write ports of the register file per call.

In connection to the FPGA-augmented TriMedia implementation, we would like to note that the flexibility in defining slot-width and slot-assignments for RFU-mapped operations determines the implementation cost. For example, assuming the maximum freedom degree in defining slot-assignments for RFU operations, a separate RFU controller has to be placed on each issue slot. In addition, the TriMedia instruction decoder has to be able to decode `EXECUTE` instructions on each of the five issue slots. Moreover, since only single- and double-slot operations are currently supported by the compiler and scheduler for the time being, the toolchain has to be modified to support 3-, 4-, and 5-issue slot operations.

Although the maximum flexibility in defining RFU-based operations may be of theoretical value, it is not of practical relevance in the context of our current investigations. As one can notice in Sections IV–VI, all RFU operations for the considered media-processing domain can be implemented with single- and double-slot operations. For this reason, in this paper we consider only a particular instance of FPGA-augmented TriMedia, in which only a single-slot instruction on Slot 1 and a double-slot instruction on Slot pair 1+2 can be issued to RFU.

For each of the single-slot, and double-slot RFU instructions, a separate operation code is allocated: `EXECUTE_1`, and `EXECUTE_2`, respectively. In both cases, the standard TriMedia-CPU64 instruction format is preserved: the opcode is a 9-bit field, and each and every source or destination registers is specified by a 7-bit field. Up to two inputs and one output,

[3]i.e., the operation slot-width and the number of input and output registers.

[4]i.e., the issuing slot(s) that the computing facility is sensitive to.

Slot 1		Slot 2		3, 4, 5
HW OPCODE	src. & dest.	NOP	src. & dest.	...

Fig. 4. Hardwired double-slot operation instruction format.

Slot 1		Slot 2		3, 4, 5
RFU OPCODE	src. & dest.	*RFU-OP ID*	src. & dest.	...

Fig. 5. RFU double-slot instruction format.

and four inputs and two outputs can be specified by the single-, and double-slot instructions, respectively:

$$\mathbf{EXECUTE\_1\ Rs1, Rs2 \rightarrow Rd1}$$
$$\mathbf{EXECUTE\_2\ Rs1, Rs2, Rs3, Rs4 \rightarrow Rd1, Rd2}$$

The EXECUTE instructions are generic, since their semantics can be redefined. By reconfiguring the raw hardware, followed by issuing an EXECUTE instruction, any new user-defined operation subject to FPGA size and TriMedia organization can be executed, while only a single entry in the opcode space is needed to encode the EXECUTE instruction. Since all the fields in the EXECUTE_1 instruction format except for the opcode field encode the input and output registers, there are no provisions for additional encoding. Thus, only a single operation per context can be encoded within EXECUTE_1. That is, if a different single-slot operation is to be launched, then a reconfiguration of the raw hardware must be carried out beforehand by SET/ ACTIVATE instructions. However, as we describe subsequently, more operations per context can be encoded within a multiple-slot EXECUTE instruction. This may reduce the number of reconfigurations when a large FPGA is available.

In the standard TriMedia-CPU64, only one of the *opcode* fields in a multiple-slot instruction defines the operation, all the others being set NOP s (Fig. 4). By using these unused fields as an argument for the RFU OPCODE (Fig. 5), a large number of RFU operations can be encoded per context, while only a single entry for the EXECUTE instruction needs to be allocated in the opcode space. Assuming a double-slot operation, for example, the 9-bit additional opcode (which is subsequently referred to as an RFU_OP IDENTIFIER or simple RFU_OP_ID) can specify 512 different operations.

We would like to mention that the two parts of the double-slot operation are decoded separately, and only when the first part specifies an EXECUTE_2 opcode, the second opcode is interpreted as an RFU_OP IDENTIFIER, and thus decoded locally at the RFU by the RFU controller. This way, an RFU super-operation does not create pressure on the instruction decoder, neatly fits in the existing instruction format, fits the existing connectivity structure to the register file, and hence requires very little hardware overhead.

The RFU controller itself can be a simple decoder for the RFU_OP_ID field, a finite-state machine having the RFU_OP_ID as argument, or even a *microcoded* engine for which the RFU_OP_ID points to the address of a micro-routine [18]. In the later case, the microcode within the RFU controller becomes part of the RFU configuration, and, therefore, subject to reconfiguration by means of SET_CONTEXT and ACTIVATE_CONTEXT instructions.

```
.rfu_op_id  IDCT   IDCT_OP_ID  ; specifies the IDCT OP_ID
.pattern    IDCT   2i+2o       ; it has 2 inputs and 2 outputs
.latency    IDCT   7           ; specifies the IDCT latency
.recovery   IDCT   2           ; specifies the IDCT recovery
```

Fig. 6. Syntax and annotation code for a user-defined IDCT operation.

In connection to the EXECUTE instructions, we would like to emphasize that their semantics, number of operands, latency, and recovery are all explicitly user-definable, while the slot-width is defined implicitly by the particular EXECUTE_1 or EXECUTE_2 opcode. Thus, it is the responsibility of the programmer to augment the machine description file with appropriate information [19]. Assuming a user-defined IDCT operation, a way to specify such information is to annotate the source code, as presented in Fig. 6. At the machine implementation level, these parameters are set by means of *Selectors*, which become part of the RFU configuration, as presented in Fig. 3. A different $\{\text{number of operands, latency, recovery}\}$ set can be defined for each RFU_OP_ID. With such mechanism, an EXECUTE instruction is truly generic, and the programmer is able to adjust its behavior as needed.

$$\mathbf{EXECUTE\_2 < IDCT > Rs1, Rs2 \rightarrow Rd1, Rd2.}$$

To give an indication of the programming complexity on FPGA-augmented TriMedia, we next outline the strategy to implement a reconfigurable design.

IV. FPGA-Augmented TriMedia-CPU64 Programming Model

To efficiently use the new reconfigurable processing facilities, the user needs a programming model. At C-level, we propose to call the RFU-based functions by defining new, so called, *custom operations*. Since custom operations are already widely used by the standard TriMedia, granting the C-level programmer a direct access to hardware operations [19], only minimal modifications of the standard compilation toolchain are required. As in the standard declaration of any custom operation, the type and number of operands, latency, and recovery have to be specified for each new RFU-based operation. However, the specification process has to be carried out by the programmer rather than by the manufacturer. In addition, the programmer has also to specify the RFU_OP_ID for multiple-slot operations. A way to specify these parameters is by using pragmas. A sample of a C-level code calling RFU is presented subsequently.

In this example, Rx_input, Ry_input, Rz_output, and Rw_output specify each a vector of four 16-bit signed integers. Thus, the IDCT operation reads in eight 16-bit signed integers and computes eight 16-bit signed integers. The standard TriMedia compiler does not recognize the pragma; therefore, the IDCT call is compiled into a function call, and the portability of the C-level code is ensured. Considering the FPGA-augmented TriMedia, the compiler does recognize the RFU_OP pragma, and generates a machine-level instruction having the EXECUTE_2 as opcode, and IDCT# as RFU_OP_ID. As specified by the *width* and *pattern* fields, this instruction designates a double-slot operation having two register inputs and two register outputs. Based on the *latency* and *recovery* fields, the

EXECUTE_2 < IDCT > operation is scheduled on slot pair 1+2 as any other hardwired operation having a latency of 16 and a recovery of 2.

```
An Example of a C-Level Code Calling the
RFU
#include < stdio.h >
#include "trimedia.h"
vec64sh Rx_input, Ry_input, Rz_output,
 Rw_output;
int main (void){
 …
 #pragma RFU_OP (width = 2 − slot, rfu_op_id =
 IDCT#, pattern = 2i + 2o, latency = 16,
 recovery = 2)
 IDCT (& Rz_output, & Rw_output, Rx_input,
 Ry_input);
 …
}
```

TriMedia is oriented to media-processing domain, in which large sets of data are manipulated in a repetitive fashion, basically around loops. Since complex long-latency operations are envisioned to be configured on the RFU, the instruction-level parallelism (ILP) within a single loop iteration containing RFU calls is expected to diminish. In order to expose to the compiler the ILP that is still available across the loop boundary, two strategies can be employed: 1) loop unrolling and/or grafting, and the more efficient 2) software pipelining (which can be regarded as infinite loop unrolling). While the first strategy tradeoffs code size (and, thus, the overhead associated to the additional instruction cache misses) for ILP [20], the second strategy significantly increases ILP while maintaining about the same code size. However, software pipelining has to tradeoff the overhead associated to firing-up and flushing the software pipeline for real time response [21]. Since an *event* (e.g., a pending interrupt) is handled only when an *interruptible jump* is decoded, the programmer is able to control the number of uninterrupted executions of the software pipeline loop by forcing a jump to be *interruptible* or *noninterruptible*. As shown in the subsequent code, this control can be managed by declaring the loop as *atomic*. The TriMedia compiler will recognize the pragma *TCS_atomic*, and generate *noninterruptible jumps* for all the jumps in the *IDCT_function*, with the exception of the return.

As a final remark, we would like to mention that the current TriMedia scheduler uses the *decision tree* as a scheduling unit [22]. Thus, all operations return their computed values in the same decision tree that they are launched, even though the TriMedia architecture does not forbid the contrary. Since generating software pipeline loops essentially requires to return the computed values beyond the decision tree boundary, decision tree-based scheduling is the major limiting factor in generating tight software pipelined loops containing long-latency operations. However, the loop containing RFU operations may be very simple and symmetrical (see, for, example [23]); thus, programming in assembly is indeed feasible despite of the fact that the host is a complex VLIW processor. Conversely, one should use C-level loop unrolling where programming directly in assembly proves to be too complex for generating tight loops.

Section V addresses the computation of three important constituents of the pel reconstruction task: entropy decoding, IQ, and 8×8 IDCT. For each constituent, a reconfigurable design is proposed, and its performance is evaluated with respect to the pure software counterpart.

```
An Example of a Deep Software Pipeline
Calling Long-Latency RFU-Based Operation
#include < stdio.h >
#include"trimedia.h"
vec64sh Rx_input, Ry_input, Rz_output,
 Rw_output;
#pragma TCS_atomic
IDCT_function (vec64sh Rx_input, vec64sh
 Ry_input, vec64sh Rz_ouput, vec64sh
 Rw_output){
 for (i = 0; i < N; i++){
  …
  #pragma RFU_OP (width = 2 − slot, rfu_op_id =
 IDCT#, pattern = 2i + 2o, latency = 16, recovery =
 2)
  IDCT (& Rz_output, & Rw_output,
 Rx_input, Ry_input);
  …
}/*the looping jump is translated into a
 'noninterruptible jump'*/
}/*the 'return' is translated into an 'in-
 terruptible jump'*/
int main (void) {
 …
 IDCT_function (Rx_input, Ry_input,
 Rz_output, Rw_output);
 …
}
```

V. Reconfigurable Designs

An FPGA-mapped operation having a latency or recovery of 1 requires an FPGA clock frequency equal to TriMedia clock frequency. Nowadays, the upper limit of the clock frequency in TriMedia family is around 300 MHz, while the maximum clock frequency for ACEX 1K FPGA family is 180 MHz. Therefore, a hypothetical implementation having a latency and/or recovery of 1 is not a realistic scenario, and a latency and recovery of 2 or more are mandatory for the time being. Subsequently, we assume that the minimum latency and minimum recovery for all FPGA-mapped circuitries are equal to 2 cycles, which translates into an FPGA cycle time to TriMedia cycle time ratio of 2 or more. Our assumption does not violate the general accepted performance ratio figures for FPGA-mapped logic versus hardwired logic—see for example [12]. Thus, considering a TriMedia running at 200 MHz, the FPGA-mapped circuitry will work with a clock frequency of 100 MHz or less.

A. Entropy Decoder

As mentioned in Section II, entropy decoding consists of variable-length decoding (VLD) followed up by run-length decoding (RLD). The input string containing *variable-length codewords* is sent to VLD, which outputs *run-level* pairs. Conceptually, the RLD outputs the number of zeros specified by the *run* value and then passes the *level* through. In a programmable processor-based platform, a common strategy to optimize this process is to fill in an empty vector with *level* values at positions defined by *run* values [24], [25].

The entropy decoder implementation on standard TriMedia is an improved version of [24]. The VLD is performed by looking-up into the VLC table that is resident into the main memory. Each lookup decodes a chunk of bits and determines whether a valid code has been encountered or not. In case of a valid decode, i.e., *hit*, *a run-level* pair is generated, or an *end-of-block* flag is set. If a *miss* is detected, that is, more bits are needed for a valid decode, an offset into the VLC table and a chunk-size for the subsequent-level lookup are generated. The process of signaling an incomplete decode and generating a new offset may be repeated up to three times. To employ loop pipelining, the RLD is folded into VLD loop. Thus, RLD for the previously extracted *run-level* pair is performed in parallel with VLD of a new symbol. The inverse zig-zag function is also folded into VLD loop; therefore, it does not generate computational overhead. Experimental results show that a coefficient can be entropy decoded in 16.9 cycles on average [25]. The cumulated size of all VLC tables is 10 kB.

Due to data dependency, both VLD and RLD are sequential tasks. Thus, entropy decoding is an intricate function on TriMedia, since a VLIW architecture must benefit from instruction level parallelism in order to be efficient. For this reason, this function is an ideal candidate to benefit from reconfigurable hardware support. Since RLD is basically a memory operation, we propose to configure on FPGA only a VLD unit. For the variable-length decoder is a system with feedback, a new `VLD` iteration can be initiated only after the previous one completes. Consequently, a recovery lower than the latency gives no advantages. As a result, such implementation should not be sought.

VLD Implementation on FPGA: The basic idea of VLD in FPGA is to store the VLC tables into EAB's. Since an EAB is a lookup table of 8 inputs while the largest codeword length is 17 bits, the VLC tables should be partitioned such that the number of bits to be decoded for each and every codeword in a partition is 8 or less. The selection of the valid partition is done by circuitry mapped into the FPGA *logic cells*. A VLD unit decoding a single symbol per call (VLD-1) is described in [26]. Due to data dependencies, a VLD-2 unit that can decode two symbols per call is more difficult to design.

As depicted in Fig. 7, the basic idea of the VLD-2 implementation is to decode *run*, *level*, and *code-length* of the *current* symbol, and to determine only the *code-length* for the *next* symbol by means of advance computation techniques. The computation of the *run-level* pair of the *next* symbol is postponed to the subsequent VLD-2 call. In parallel, the *run-level* pair of the *previous* codeword is decoded. Thus, with the exception of a firing-up call which decodes only a single symbol, truly two-symbol decoding is achieved for all subsequent VLD-2 calls [27].

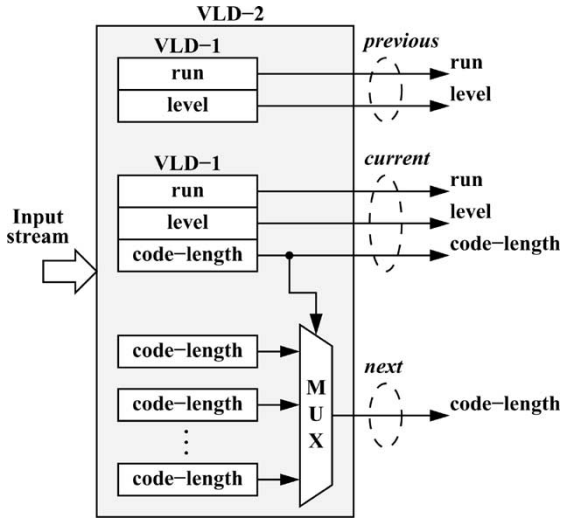

Fig. 7. The conceptual VLD-2 implementation on FPGA.

All 12 EAB's and 51% of the logic cells of an EP1K100 device have been used to implement the MPEG2-compliant variable-length decoder. By simulation with Altera tools, we found that the VLD-2 latency is equal to 8 TriMedia@200 MHz cycles.

Entropy Decoding on Extended TriMedia: Two operations are needed to control the VLD-2 unit: one for reset, and the other to launch the proper VLD-2 operation, as follows:

$$\mathbf{EXECUTE\_2} < \mathbf{RESET - VLD - 2} > \rightarrow (\mathbf{void})$$
$$\mathbf{EXECUTE\_2} < \mathbf{VLD - 2} > \mathbf{Ry}, \mathbf{Ryy} \rightarrow \mathbf{Rz}, \mathbf{Rw}$$

The Ry and Ryy registers contain the incoming bit string which has been aligned with standard TriMedia shifting operations to point to the *previous* and *current* symbols, respectively. The *run*, *level*, code-length for both codewords, as well as control information are stored into the Rz and Rw registers.

The entropy decoding routine calling the FPGA-mapped VLD-2 is also organized as a software pipeline (the RLD is folded into the loop). After compiling and scheduling the entropy decoding routine, we determined that a single DCT coefficient can be decoded in 9 cycles. For more details we refer the reader to [27].

B. IQ Instruction and Computing Facility

After entropy decoding, the two-dimensional (2-D) array of coefficients, $QF[v][u]$, is inverse quantized to produce the reconstructed DCT coefficients, $F[v][u]$. In MPEG2, IQ consists of three stages: inverse quantization arithmetic, saturation, and mismatch control [1]. The inverse quantization arithmetic produces $F''[v][u]$ coefficients. For dc coefficients in intracoded blocks, (1) is used:

$$F''[0][0] = \text{intra\_dc\_mult} \times QF[0][0] \tag{1}$$

where the factor *intra_dc_mult* is derived from the data element *intra_dc_precision* according to Table 7-4 of the ITU-T Recommendation H.262 [1]. Basically, (1) specifies a scaling-up by a

factor of 8, 4, 2, or 1. For all other coefficients, the following equation should be used:

$$F''[v][u] = (2 \times QF[v][u] + k) \times W[v][u] \times \frac{\text{quantizer\_scale}}{32} \tag{2}$$

where $k = \text{sign}(QF[v][u])$ for nonintrablocks, and $k = 0$ for intrablocks. The factor *quantizer_scale*, which is derived from the data elements *quantizer_scale_code* and *quantizer_scale_type* according to Table 7-6 of the ITU-T Recommendation H.262 [1], is encoded as a 7-bit fixed length code. Each weighting coefficient, $W[v][u]$, $v = 0 \ldots 7$ $u = 0 \ldots 7$, is represented on an 8-bit unsigned integer, and extracted during the parsing of the picture header. The operator "/" represents the integer division with truncation of the result toward zero.

The coefficients resulting from the inverse quantization arithmetic are saturated to lie in the range $[-2048 \ldots +2047]$. Finally, the mismatch control operation toggles the least significant bit of $F[7][7]$ if the double sum $\sum_{v=0}^{7}\sum_{u=0}^{7} F[v][u]$ is even.

In standard TriMedia, the IQ implementation intensively uses four-way SIMD operations. The 8×8 matrix is stored in 16 four-element vectors, each element being a 16-bit signed integer. First, all 64 coefficients are inverse quantized with the general (2), and then saturated. Then, the mismatch sum is computed as if the block is nonintracoded. In parallel, the intra dc coefficient is processed with (1), and then saturated. Finally, the resulting dc coefficient, and, consequently, the mismatch sum are updated if the block is intracoded. An entire 8×8 matrix can be inverse quantized in 39 cycles for intra-, and 52 cycles for nonintracoded blocks, respectively (load and store operations are taken into account).

Since the IQ is mostly a symmetrical feed-forward task, the entire IQ computation can benefit from reconfigurable support. This case, the VLIW core has only to load new data from and write the computed data back to main memory. The IQ implementation details are outlined subsequently.

IQ Implementation on FPGA: The number of pixels which can simultaneously be inverse quantized on FPGA is subject to the raw hardware logic capacity. On an ACEX EP1K100 FPGA, we succeeded to map an IQ-4 unit that can process four coefficients per call. As shown in Fig. 8, the IQ-4 unit is structured as follows: the 7-stage pipeline implements the inverse quantization arithmetic and the subsequent saturation, while the finite state machine implements the mismatch control operation.

The reduction modules corresponding to multiplications by $W[v][u]$ (8-bit unsigned integer) and quantizer_scale (7-bit unsigned integer) have been splitted-up to fit into an 100 MHz pipeline. The factors intra_dc_mult and quantizer_scale are generated inside FPGA from the MPEG data elements intra_dc_precision, respectively quantizer_scale_code and q_scale_type. The finite state machine accumulates the mismatch information during successive IQ-4 calls, and updates the last DCT coefficient accordingly at the end of each 16th call. Thus, to ensure the correct response, a block should be completely processed before a new one is being considered. Furthermore, the 64-bit word containing the dc component should be processed firstly, and the 64-bit word containing the highest spatial frequency component should be processed lastly.

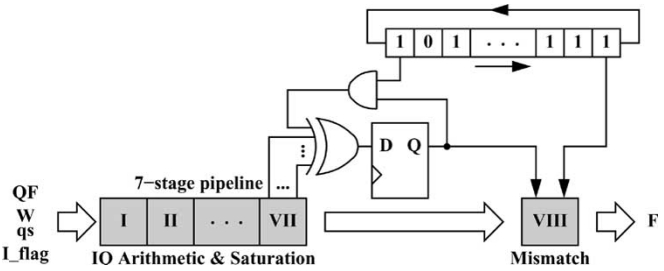

Fig. 8. The IQ Implementation on FPGA.

By writing and synthesizing VHDL code, we mapped an IQ-4 unit on an ACEX EP1K100 FPGA, and determined a latency of 18 and a recovery of 2 TriMedia@200 MHz cycles. It worth to mention that 43% of the logic cells are occupied by IQ-4.

IQ on Extended TriMedia: Two operations are needed to control the IQ-4 unit: one which resets the finite state machine, and the other to launch the proper IQ-4 operation, as follows:

$$\textbf{EXECUTE\_2} < \textbf{RESET} - \textbf{IQ} - \textbf{4} > \rightarrow (\textbf{void})$$
$$\textbf{EXECUTE\_2} < \textbf{IQ} - \textbf{4} > \textbf{R\_QF}, \textbf{R\_W}, \textbf{R\_qs}, \textbf{R\_param} \rightarrow \textbf{Rd}$$

To inverse quantize an 8×8 block of coefficients, sixteen `IQ-4` operations are launched in a row. Before and after the RFU calls, `LOAD` and `STORE` operations fetch the input operands from main memory into register file, and store the results back into memory, respectively. Since the code is very simple and symmetrical, generating a tight software-pipeline loop by programming directly in assembly is indeed feasible. With such implementation, a throughput close to 1/32 IQ/cycle can be achieved.

C. IDCT Instruction and Computing Facility

The N-point 1-D IDCT is defined by [28]:

$$x_i = \frac{2}{N}\sum_{u=0}^{N-1} K_u X_u \cos\frac{(2i+1)u\pi}{2N}$$

where X_u are the inputs, x_i are the outputs, and $K_u = \sqrt{1/2}$ for $u = 0$, otherwise is 1. For MPEG, a 2-D IDCT processes an 8×8 matrix X [5]:

$$x_{i,j} = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} K_u K_v X_{u,v} \cos\frac{(2i+1)u\pi}{16} \cos\frac{(2j+1)v\pi}{16}.$$

A standard way to compute the 2-D IDCT is the row-column separation: the 1-D transform is first applied to each row and then to each column of the 8×8 matrix. This strategy can be combined with the (modified) Loeffler 1-D IDCT algorithm to further reduce the computational complexity [29], [30].

In standard TriMedia, four 16-bit coefficients, which are stored in a 64-bit word, are processed in parallel by a single four-way SIMD operation. Eight 1-D IDCT's are first computed as two (4-way) SIMD 1-D IDCT's using the modified

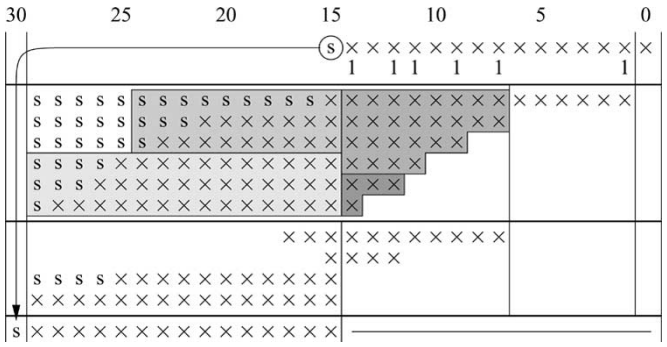

Fig. 9. The partial product matrix and the selected reduction steps for multiplication by the constant $C_0' = 0x3a82$.

"Loeffler" algorithm. Then, the matrix transposition is performed by the hardwired transpose unit. Finally, eight 1-D IDCT's (two SIMD 1-D IDCT's) are computed. This way, a 2-D IDCT including `LOAD` and `STORE` operations can be performed in 56 cycles [2].

Since the standard TriMedia provides a good support for transposition and matrix storage, there is a little benefit to configure the entire 2-D IDCT on FPGA. Consequently, only an 1-D IDCT unit is configured on the RFU. Some FPGA implementation details of 1-D IDCT are presented subsequently.

1-D IDCT Implementation on FPGA: All operations are implemented using 16-bit fixed-point arithmetic [30]. For all multiplications, the multiplicand is a 16-bit signed number represented in 2's complement notation, while the multiplier is a 15-bit or less positive constant [23]. To reduce the implementation costs, we used a *multiplication-by-constant* scheme, which is optimized against the multiplier, as follows. First, we built a partial product matrix, where only the rows corresponding to a '1' in the multiplier are filled in. Then, reduction schemes which fit into 100 MHz pipeline stages are sought. For example, the multiplication by the constant $C_0' = 0x3a82$ is performed in two pipeline stages [23], as shown in Fig. 9.

After several optimization iterations, we succeeded to fit the 1-D IDCT into a 7-stage pipeline running at 100 MHz. As described in [23], this figure translates to an 1-D IDCT operation with a latency of 8 and recovery of 2 TriMedia cycles.

2-D IDCT on Extended TriMedia: By launching an 1-D IDCT double-slot operation having two 64-bit inputs (Rx and Ry) and two 64-bit outputs (Rz and Rw), an 8-point IDCT is computed:

$$\mathbf{EXECUTE\_2} < \mathbf{1-D-IDCT} > \mathbf{Rx}, \mathbf{Ry} \rightarrow \mathbf{Rz}, \mathbf{Rw}$$

To calculate the 2-D IDCT, eight 1-D IDCT's are firstly computed. Then, the 8×8 matrix is transposed using the hardwired `TRANSPOSE` operation. Finally, eight 1-D IDCT's complete the 2-D IDCT. Before and after each 2-D IDCT, `LOAD` and `STORE` operations fetch the input operands from main memory into register file, and store the results back into memory, respectively.

The 2-D IDCT is organized as a software pipeline loop inside which FPGA-based 1-D IDCT operations are launched. As described in [23], the code is very simple; thus, programming in assembly is indeed possible. With such implementation, a throughput of $1/32$ IDCT/cycle has been achieved. For more details we refer the reader to [23].

VI. Experimental Results

In order to determine the performance gain provided by the multiple-context FPGA, we consider the pel reconstruction as benchmark. As mentioned, it consists of entropy decoding, IQ, 2-D IDCT, and some extra tasts (header parsing, decoding of motion vectors, etc.). Our experiment includes two approaches: *pure software* and *FPGA-based*. For the first approach, we would like to remind that a DCT coefficient can be decoded in 16.9 cycles [25]. A pure software 2-D IDCT can be scheduled in 56 cycles [2]. IQ takes 39 cycles per intrablock and 52 cycles per nonintrablock.

In the FPGA-based approach, the VLD, IQ, and IDCT functions benefit from reconfigurable hardware support. For each of the mentioned function, the corresponding code is replaced by a group of three instructions: `SET_CONTEXT`, `ACTIVATE_CONTEXT`, and `EXECUTE`. Due to the large off-chip reconfiguration penality, all the RFU contexts are configured at application load-time, i.e., the `SET_CONTEXT` instructions are all scheduled on the top of the program code. During the program execution, the VLD-2, IQ-4, and 1-D IDCT units are activated by `ACTIVATE_CONTEXT` instructions as needed. As mentioned in Section II, the context switching penality is 100 cycles.

The testing database consists of five MPEG-2 conformance scenes. For all experiments, the incoming string is assumed to be entirely resident into the main memory. This way, side effects associated with string acquisition (such as asynchronous interrupts or other operating system related tasks) do not have to be counted. With this assumption, the relevant metric is the number of the instruction cycles required to perform pel reconstruction.

In Section V, we outlined a reconfigurable entropy decoder that decodes a DCT coefficient in 9 cycles, a reconfigurable inverse quantizer and a reconfigurable 2-D IDCT, each of them having a throughput of one 8×8 block every 32 cycles. Although each of these reconfigurable designs has been highly optimized, the problem is not completed yet. There are different ways to embed the reconfigurable ingredients into the pel reconstruction task, and the programmer has to find the fastest compound implementation. Basically, each implementation has to be evaluated against the overhead of firing-up and flushing the IQ and IDCT pipelines, the compulsory cache misses, and the FPGA context-switching penalty. Processing large batches of blocks translates into a low pipeline firing-up/flushing overhead and low context-switching penalty but high cache-miss penalty. A way to vary the batch size while the MPEG decoding synchronization is ensured is to carry out the decoding process at different levels: **macroblock**, **slice**, and **picture/frame**. Thus, to assess the performance for different batch sizes, the pel reconstruction task has been analyzed according to three computing scenarios, as depicted in Fig. 10. That is, the VLD is performed till an entire macroblock/slice/picture is extracted, and only then the IQ and IDCT are carried out for all blocks in a macroblock/slice/picture, respectively.

First, pel reconstruction at macroblock level is analyzed. According to Table I, the average number of blocks per macroblock

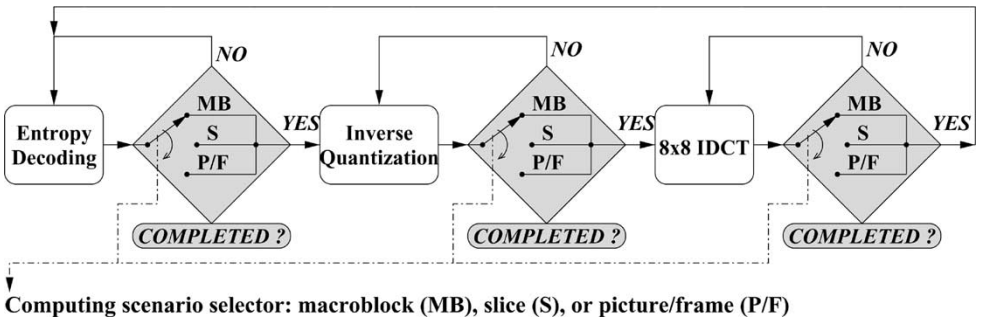

Fig. 10. Three possible computing scenarios of pel reconstruction.

TABLE I
MPEG-2 STATISTICS FOR SEVERAL CONFORMANCE BIT-STRINGS

Scene	Blocks/macroblock						Blocks/slice						Blocks/{picture,frame}						Macroblocks with blocks/slice						Coded macroblocks/slice						Slices/{picture,frame}		
	I		P		B		I		P		B		I		P		B		I		P		B		I		P		B		I	P	B
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	μ	μ
bat_327_334	–	–	5.7	0.2	5.2	0.3	–	–	257	9	234	14	–	–	9236	236	8432	258	–	–	45	0	45	0	–	–	45	0	45	0	–	36	36
popplen	6.0	0	3.7	0.9	3.1	1.4	264	0	80	33	38	27	3960	0	1202	198	573	93	44	0	22	8	11	5	44	0	44	0	42	5	15	15	15
sarnoff2	6.0	0	4.1	0.5	2.4	0.4	270	0	171	26	61	23	8100	0	5120	0	1823	26	45	0	42	3	24	7	45	0	44	1	45	1	30	30	30
tennis	6.0	0	4.0	0.4	2.2	0.5	264	0	167	27	71	32	9504	0	5997	221	2563	718	44	0	41	5	31	10	44	0	43	3	44	1	36	36	36
ti1cheer	6.0	0	4.1	0.7	2.8	0.7	264	0	155	56	88	45	7920	0	4481	0	1324	62	44	0	36	11	29	11	44	0	38	10	34	11	30	30	15
Average	6.0	–	4.3	–	3.1	–	266	–	166	–	98	–	n/a	–	n/a	–	n/a	–	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

TABLE II
TOTAL NUMBER OF BLOCKS, MACROBLOCKS, SLICES, AND PICTURES/FRAMES FOR SEVERAL MPEG-2 CONFORMANCE BIT-STRINGS

Scene	Blocks				Macroblocks with blocks				Coded macroblocks				Slices	{Pictures,Frames}
	I	P	B	Total	I	P	B	Total	I	P	B	Total	IPB	IPB
bat_327_334	0	36,943	25,295	62,238	0	6,473	4,847	11,320	0	6,480	4,860	11,340	252	7 (7+0)
popplen	3,960	3,606	1,145	8,711	660	970	338	1,968	660	1,980	1,247	3,887	90	3 (0+6)
sarnoff2	8,100	5,120	3,645	16,865	1,350	1,258	1,468	4,076	1,350	1,332	2,671	5,353	120	4 (4+0)
tennis	9,504	17,992	10,250	37,746	1,584	4,457	4,450	10,491	1,584	4,625	6,305	12,514	288	8 (8+0)
ti1cheer	7,920	4,481	5,275	17,676	1,320	1,049	1,722	4,091	1,320	1,106	2,045	4,471	120	4 (2+4)

TABLE III
COMPULSORY/TRASHING CACHE MISS PENALTY, FPGA CONTEXT SWITCHING OVERHEAD, AND PIPELINE FIRE-UP + FLUSHING OVERHEAD FOR DIFFERENT COMPUTING SCENARIOS

Scene	Macroblock level				Slice level – 16KB Data cache (D$)				Slice level – 32KB Data cache (D$)				Picture/Frame level			
	D$ misses (cycles)	FPGA (cycles)	pipeline (cycles)	**Total** (cycles)	D$ misses (cycles)	FPGA (cycles)	pipeline (cycles)	**Total** (cycles)	D$ misses (cycles)	FPGA (cycles)	pipeline (cycles)	**Total** (cycles)	D$ misses (cycles)	FPGA (cycles)	pipeline (cycles)	**Total** (cycles)
bat_327_334	1,369,236	3,396,000	412,624	**5,177,860**	2,688,444	75,600	8,880	**2,772,924**	1,395,020	75,600	8,880	**1,479,500**	4,107,708	2,100	268	**4,110,076**
popplen	191,642	590,400	58,406	**840,448**	285,054	27,000	3,368	**315,422**	196,922	27,000	3,368	**227,290**	574,926	1,800	216	**576,942**
sarnoff2	371,030	1,222,800	104,336	**1,698,166**	617,166	36,000	3,776	**656,942**	389,510	36,000	3,776	**429,286**	1,113,090	1,200	112	**1,114,402**
tennis	830,412	3,147,300	238,300	**3,385,600**	1,238,864	86,400	9,600	**1,334,864**	843,084	86,400	9,600	**939,084**	2,491,236	2,400	288	**2,493,924**
ti1cheer	388,872	1,227,300	129,260	**1,356,560**	625,988	36,000	3,600	**665,588**	399,432	36,000	3,600	**439,032**	1,166,616	1,800	184	**389,236**

is small, being 3.1 for B-type pictures, 4.3 for P-type pictures, and 6.0 for I-type pictures. Given the fact that the overhead to fire-up the 2-D IDCT software pipeline is 20 cycles [23], $6 \times 32 + 20 = 212$ cycles are needed to compute 2-D IDCT for an entire intracoded macroblock, which translates to an average of $212/6 = 35.5$ cycles/block (11% performance degradation versus the ideal 32 cycles/block). The performance degrades even more for nonintracoded macroblocks. For the worst case when the number of blocks is odd, the last 2-D IDCT will complete in 45 cycles instead of 32, giving a total of $32 + 32 + 45 + 20 = 129$ cycles for the average of 3 blocks in a B-coded macroblock. This translates to 43 cycles/block (34% performance degradation versus the ideal 32 cycles/block).

On the other side, processing a very large number of blocks in order to minimize the pipeline firing-up and flushing overhead implies that *trashing* cache misses are generated when data is read from and written back to main memory. Assuming frame-level pel reconstruction, the smallest average number of blocks per frame is equal to 573 (Table I—popplen scene). Thus, a data cache of $573 \times 64 \times 2 = 72$ kB is needed to avoid trashing cache misses after entropy decoding. Such a data cache is larger than the current TriMedia cache (16 kB or 32 kB). Assuming a penalty of 11 cycles per cache miss, an overhead of $16\ \text{words} \times 11\ \text{cycles} = 176\ \text{cycles}$ per block is generated. Thus, all the performance gain provided by the reconfigurable design is lost.

The FPGA context switching occurs three times per pel reconstruction iteration to successively activate VLD-2, IQ-4, and 1-D IDCT (each switch takes 100 cycles). Based on the total number of macroblocks, slices, and pictures/frames in an MPEG string (Table II), we can determine the FPGA context-switching penalty for all three computing scenarios. For example, there are 3 context switches per slice for slice-level decoding, which translates to $288\ \text{slices} \times 3 \times 100 = 86{,}400$ cycles for the *tennis* scene. A complete list of penalties is presented in Table III.

Two data cache sizes have been considered: 16 kB and 32 kB. As it can be observed, the lowest total penalty is achieved for decoding at slice level. This winning computing scenario, which will be used for comparison with the pure-software solu-

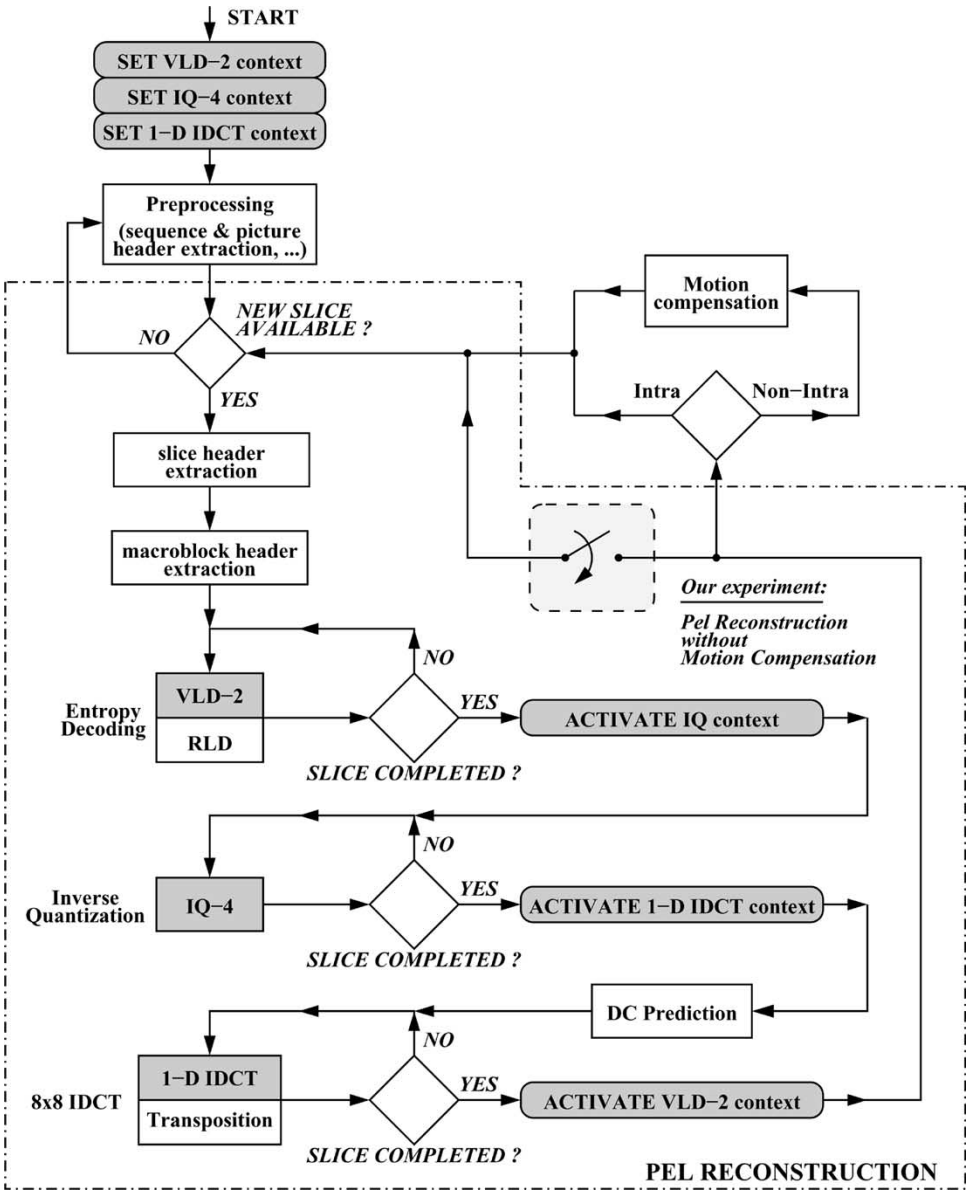

Fig. 11. The winning computing scenario of pel reconstruction.

tion, is presented *in extenso* in Fig. 11. For each slice, the variable-length decoding of all macroblocks (headers and DCT coefficients extraction) is first performed. By software pipelining, run-length decoding is carried out in parallel to recreate the 8×8 matrices. Then, all the blocks in the slice are inverse quantized, and dc coefficient prediction for intracoded macroblocks is carried out. Finally, a burst of 2-D IDCT's is launched in order to complete the reconstruction of the initial matrices of pels.

In connection with the cache miss penalty that is encountered at slice-level decoding, we would like to mention that the number of trashing cache misses approaches to zero if a 32 kB data cache is available. The remaining penalty, e.g., 1 395 020 cycles for the bat_327_334 scene, is mostly due to the compulsory cache misses that are encountered during entropy decoding. Assuming a data cache of only 16 kB, a possible strategy to keep the number of trashing cache misses at low level is to split the slice-level processing into subparts. This way, the decoding will be carried out at *sub-slice level*. Since this issue is somehow beyond the paper scope, we will not go into details.

The winning reconfigurable design is compared to the pure-software solution. All three FPGA contexts are setup at application load time. Since long MPEG strings of minutes or hours (much longer that the conformance scenes) are to be decoded in practice, the overhead for setting-up the FPGA contexts ($3 \times 1\,671\,360 = 5\,014\,080$ cycles, which corresponds to 25 msec on a TriMedia@200 MHz) is not taken into consideration. The experimental results are presented in Table IV. The figures indicate the number of instruction cycles needed to process the MPEG string and the overhead. For example, 401 492 cycles are needed to perform IQ for the popplen scene. For the same scene, there is an overhead of 196 922 cycles due to cache misses, which is taken into consideration for both FPGA-based and pure-software solutions. For the FPGA-based solution only, there is an additional overhead of 30 368 cycles

TABLE IV
PEL RECONSTRUCTION EXPERIMENTAL RESULTS FOR THE WINNING COMPUTING SCENARIO

Routine	Extra tasks: header parsing, motion vectors decoding, ...	2-D IDCT (1-D IDCT on FPGA)		Entropy Decoding (VLD-2 on FPGA)		Inverse Quantization (IQ on FPGA)		Overhead		**Pel reconstruction**	
								cache	FPGA		
Scene	TM & TM+FPGA (cycles)	TM (cycles)	TM+FPGA (cycles)	TM (cycles)	TM+FPGA (cycles)	TM (cycles)	TM+FPGA (cycles)	TM & TM+FPGA (cycles)	TM+FPGA (cycles)	TM (cycles)	TM+FPGA (cycles)
bat_327_334	7,122,569	3,485,328	2,000,496	7,435,734	3,553,462	3,236,376	1,997,664	1,395,020	84,480	**22,675,027**	**16,153,691**
popplen	1,328,094	487,816	282,120	1,252,879	609,614	401,492	280,912	196,922	30,368	**3,667,203**	**2,728,030**
sarnoff2	2,153,508	944,440	543,456	1,964,877	973,039	771,680	542,560	389,510	39,776	**6,224,015**	**4,641,849**
tennis	4,495,546	2,113,776	1,217,472	4,675,214	2,287,978	1,839,240	1,214,784	843,084	96,000	**13,966,860**	**10,154,864**
ti1cheer	1,708,082	989,856	569,292	2,147,769	1,088,928	816,192	568,488	399,432	39,600	**6,061,331**	**4,373,822**

TABLE V
PEL RECONSTRUCTION RELATIVE IMPROVEMENT

Scene	**bat_327_334**	**popplen**	**sarnoff2**	**tennis**	**ti1cheer**
Entropy Decoding (VLD-2 on FPGA)	52.2%	51.3%	50.5%	51.1%	49.3%
Inverse Quantization (IQ-4 on FPGA)	38.3%	30.0%	29.7%	34.0%	30.3%
2-D IDCT (1-D IDCT on FPGA)	42.6%	42.2%	42.5%	42.4%	42.5%
Pel reconstruction (VLD-2, IQ-4, 1-D IDCT on FPGA)	**28.8%**	**25.6%**	**25.4%**	**27.3%**	**27.8%**

due to FPGA context switching (27,000 cycles) and pipeline firing-up and flushing (3,368 cycles).

A digest of the Table IV, which reports the relative improvement of the FPGA-augmented TriMedia versus standard TriMedia with respect to the number of cycles, is presented in Table V. For each function that benefit from FPGA support, i.e., entropy decoding, IQ, and IDCT, only the number of instruction cycles needed to perform strictly that particular function are considered (which is similar to assume zero overhead). For the entire pel reconstruction task, all the overhead assuming a 32 kB data cache is included. As it can be observed, the FPGA-augmented TriMedia can perform MPEG2-compliant pel reconstruction with the average improvement of 27% in terms of cycles over the standard TriMedia.

The speed-up achieved on FPGA-augmented TriMedia is presented in Fig. 12. When VLD-2, IQ-4, and 1-D IDCT are each configured on a different FPGA context, pel reconstruction can be computed with the average speed-up of 1.4 $\times$. Assuming that only a single-context FPGA is available, then only one of the VLD-2, IQ-4, and 1-D IDCT functions can benefit from reconfigurable hardware support. The speed-up decreases to 1.1 $\times$ when either IQ-4 or 1-D IDCT is configured on the RFU, and to 1.2 $\times$ when only VLD-2 is configured on the RFU.

The significant speed-up of 1.4 $\times$ obtained for a 5-issue slot VLIW processor with a 64-bit datapath may generate the idea of providing more hardwired functional units instead of the reconfigurable array to achieve at least the same processing speed. However, this approach seems not to be feasible. As we mentioned, 4 out of 5 issue slots are filled in with operations in the pure-software entropy decoder [25], and more that 4.5 out of 5 issue slots are occupied in the pure-software IQ and 2-D IDCT. Since more operations per cycle cannot be issued, the processor cannot run faster whatever additional (fine-grain) hardwired functional units are provided.

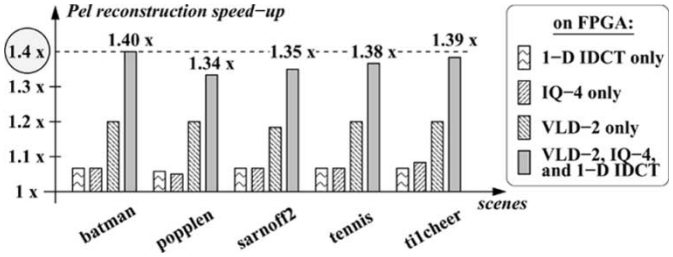

Fig. 12. FPGA-augmented TriMedia versus standard TriMedia speed-up.

The final remark addresses the area penalty induced by the FPGA core. According to DeHon [12], an additional area of 500 000 λ^2 per LUT is needed. For the ACEX EP1K100 device that includes about 5,000 LUT's, this figure translates to a total area of 20 mm^2 in a 0.18 $\mu\mathrm{m}$ technology. Giving the fact that TriMedia-CPU64 occupies about 35 mm^2 in the same technology, the FPGA augmentation with an EP1K100 device leads to a relative area increase of 43%. However, this increase is not a major concern. Indeed, TriMedia is envisioned to be embedded on the same die with a set of coprocessors, which also require additional area. For example, in the Viper chip [31], the MPEG video decoder and video coprocessor occupy together an area of about 18 mm^2. Replacing such coprocessors (which cannot be used for different tasks they have been designed for anyhow) with a reconfigurable core will keep the total die area at the same level. However, the physical realization of the FPGA-augmented TriMedia is subject for further work.

VII. CONCLUSIONS

We have described an architectural extension for TriMedia-CPU64, which encompasses a multiple-context FPGA-based RFU, a hardwired Configuration Unit managing the reconfiguration of the RFU, and their associated instructions. On the FPGA-augmented TriMedia-CPU64, we determined a speed-up of 1.4 $\times$ over a standard TriMedia-CPU64 for an MPEG2-compliant pel reconstruction task, at the expense of three new instructions: `SET_CONTEXT`, `ACTIVATE_CONTEXT`, `EXECUTE`, and a 5,000-cell FPGA. Given the fact that the experimental TriMedia instance is a 5-issue slot VLIW processor with a 64-bit datapath and a very rich multimedia-oriented instruction set, such an improvement within its target media processing domain indicates that TriMedia + FPGA hybrid is a promising approach.

REFERENCES

[1] "Information Technology—Generic Coding of Moving Pictures and Associated Audio Information: Video," International Telecommunication Unit, ITU-T Recommendation H.262, 2000.

[2] J. T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E. -J. D. Pol, M. J. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. Vranken, "Trimedia CPU64 architecture," in *Proc. Int. Conf. Computer Design (ICCD)*, Austin, TX, Oct. 1999, pp. 586–592.
[3] A. K. Riemens, K. A. Vissers, R. J. schutten, F. W. Sijstermans, G. J. Hekstra, and G. D. La Hei, "TriMedia CPU64 Application Domain and Benchmark Suite," in *Proc. Int. Conf. Computer Design (ICCD)*, Austin, TX, pp. 580–585.
[4] G. J. Hekstra, G. D. La Hei, P. Bingley, and F. W. Sijstermans, "TriMedia CPU64 Design Space Exploration," in *Proc. Int. Conf. Computer Design (ICCD)*, Austin, TX, Oct. 1999, pp. 599–600.
[5] J. L. Mitchell, W. B. Pennebaker, C. E. Fogg, and D. J. LeGall, *MPEG Video Compression Standard*. New York: Chapman & Hall, 1996.
[6] M.-T. Sun, "VLSI implementations for image communications," in *Chapter Design of High-Throughput Entropy Codec*. Amsterdam, The Netherlands: Elsevier, 1993, vol. 2, pp. 345–364.
[7] B. G. Haskell, A. Puri, and A. N. Netravali, *Digital Video: An Introduction to MPEG-2*. Norwell, Massachusetts: Kluwer Academic Publishers, 1996.
[8] S. Brown and J. Rose, "Architecture of FPGAs and CPLDs: A tutorial," *IEEE Trans. Des. Test Comput.*, vol. 13, pp. 42–57, Dec. 1996.
[9] A. DeHon, "DPGA-coupled microprocessors: Commodity IC's for the early 21st century," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, Napa Valley, CA, Apr. 1994, pp. 31–39.
[10] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *Proc. IEEE Symp. FPGAs Custom Computing Machines (FCCM)*, Napa Valley, CA, Apr. 1997, pp. 22–28.
[11] "Configuring APEX 20K, FLEX 10K & FLEX 6000 Devices, Datasheet," Altera Corporation, San Jose, CA, 1999.
[12] A. DeHon, "Reconfigurable architectures for general-purpose computing," in *A. I. 1586*. Cambridge, MA: Massachusetts Inst. Technology, 1996.
[13] "ACEX 1K Programmable Logic Family, Datasheet," Altera Corporation, San Jose, California, 2000.
[14] J. T. J. van Eijndhoven, G. A. Slavenburg, and S. Rathnam, "VLIW Processor has Different Functional Units Operating on Commands of Different Widths," U.S. Patent 076 154, 2000.
[15] M. Sima, S. Vassiliadis, S. Cotofana, J. T. J. van Eijndhoven, and K. A. Vissers, "Field-programmable custom computing machines. A taxonomy," in *Proc. Field-Programmable Logic and Applications (FPL)*, Montpellier, France, 2002, pp. 79–88.
[16] "XC6200 Field Programmable Gate Arrays, Datasheet," Xilinx Corporation, San Jose, California, 1996.
[17] "AT6000 Series Configuration, Application Note," Atmel Corporation, San Jose, CA, 1999.
[18] S. Vassiliadis, S. Wong, and S. Cotofana, "The MOLEN rm-coded processor," in *Field-Programmable Logic and Applications (FPL 2001)*, Belfast, Northern Ireland, Aug. 2001, pp. 275–285.
[19] E. -J. D. Pol, B. J. M. Aarts, J. T. J. van Eijndhoven, P. Struik, F. W. Sijstermans, M. J. A. Tromp, J. W. van de Waerdt, and P. W. van der Wolf, "Trimedia CPU64 application development environment," in *Proc. Int. Conf. Computer Design (ICCD '99)*, Austin, TX, Oct. 1999, pp. 593–598.
[20] ——, *Book 4—Software Tools. Part A: C Language Users Guide*. Milpitas, California, U.S.A: TriMedia Technologies, Inc., 2000.
[21] ——, *Book 2—Cookbook. Part D: Optimizing TriMedia Applications*, M. Boulevard, Ed. Milpitas, CA: TriMedia Technologies, Inc., 2000.
[22] J. Hoogerbrugge and L. Augusteijn, "Instruction scheduling for TriMedia," *J. Instruction-Level Parallelism*, vol. 1, no. 1, Feb. 1999.
[23] M. Sima, S. Cotofana, J. T. J. van Eijndhoven, S. Vassiliadis, and K. A. Vissers, "8 × 8 IDCT implementation on an FPGA-augmented TriMedia," in *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM 2001)*, Rohnert Park, CA, Apr. 2001.
[24] E.-J. D. Pol, *VLD Performance on Trimedia-CPU64*: private communication, 2000.
[25] M. Sima *et al.*, "Entropy decoding on TriMedia-CPU64," in *Proc. System Architecture Modeling and Simulation Workshop (SAMOS 2002)*, Samos, Greece, 2002.
[26] M. Sima, S. Cotofana, S. Vassiliadis, J. T. J. van Eijndhoven, and K. A. Vissers, "MPEG macroblock parsing and pel reconstruction on an FPGA-augmented TriMedia processor," in *Proc. IEEE Int. Conf. Computer Design (ICCD 2001)*, Austin, TX, Sept. 2001, pp. 425–430.
[27] ——, "MPEG-compliant entropy decoding on FPGA-augmented TriMedia-CPU64," in *IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM 2002)*, Napa Valley, CA, April 2002, pp. 261–270.
[28] K. R. Rao and P. Yip, *Discrete Cosine Transform. Algorithms, Advantages, Applications*. San Diego, CA: Academic, 1990.
[29] C. Loeffler, A. Ligtenberg, and G. S. Moschytz, "Practical fast 1-D DCT algorithms with 11 multiplications," in *Intl. Conference on Acoustics, Speech, and Signal Processing (ICASSP '89)*, 1989, pp. 988–991.
[30] J. T. J. van Eijndhoven and F. W. Sijstermans, "Data Processing Device and Method of Computing the Cosine Transform of a Matrix," U.S. Patent 397 235, 2002.
[31] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A multiprocessor SOC for advanced set-top box and digital TV systems," *IEEE Des. Test Comput.*, vol. 18, pp. 21–31, Sept.-Oct. 2001.

Mihai Sima (M'01) was born in Bucharest, Romania. He received the M.S. degree in Electrical Engineering from "Politehnica" University of Bucharest, and the Ph.D. degree in Electrical Engineering from Delft University of Technology, The Netherlands.

He had been with the "Microelectronics" Company in Bucharest for 3 years, where he was involved in instrumentation electronics for integrated circuit testing. Subsequently, he joined the Telecommunications Department of "Politehnica" University of Bucharest, where he had been involved in digital signal processing and speech recognition for 6 years. More recently, he had been with the Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, where he worked on reconfigurable architectures for media-processing domain. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of Victoria, B.C., Canada. His research interests include computer architecture, reconfigurable computing, embedded systems, digital signal processing, and speech recognition.

Sorin D. Coţofană (SM'00) was born in Mizil, Romania. He received the M.S. degree in Computer Science from the "Politehnica" University of Bucharest, Romania, and the Ph.D. degree in Electrical Engineering from Delft University of Technology, The Netherlands.

He worked with the Research & Development Institute for Electronic Components (ICCE) in Bucharest for ten years, involved in structured design of digital systems, design rule checking of IC's layout, logic and mixed-mode simulation of electronic circuits, testability analysis, and image processing. He is currently an Associate Professor with the Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, The Netherlands. His research interests include computer arithmetic, parallel architectures, embedded systems, reconfigurable computing, nano-electronics, neural networks, computational geometry, and computer aided design.

Stamatis Vassiliadis (F'97) was born in Manolates, Samos, Greece.

He is a Professor with the Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, The Netherlands. He has also served in the faculties of Cornell University, Ithaca, NY, and the State University of New York (S.U.N.Y.), Binghamton, NY. He worked for a decade with IBM in the Advanced Workstations and Systems laboratory in Austin TX, the Mid-Hudson Valley Laboratory in Poughkeepsie, NY, and the Glendale Laboratory in Endicott, NY. At IBM he was involved in a number of projects regarding computer design, organizations, and architectures and the leadership to advanced research projects. A number of his design and implementation proposals have been implemented in commercially-available systems and processors including the IBM 9370 model 60 computer system, the IBM POWER II, the IBM AS/400 Models 400, 500, and 510, Server Models 40S and 50S, the IBM AS/400 Advanced 36, and the IBM S/390 G4 and G5 computer systems.

Dr. Vassiliadis has received numerous awards including 23 levels of Publication Achievement Awards, 15 levels of Invention Achievement Awards and an Outstanding Innovation Award for Engineering/Scientific Hardware Design in 1989. In 1990, he was awarded the highest number of USA patents in IBM, six of his 70 USA patents being rated with the highest patent ranking in IBM.

Jos T. J. van Eijndhoven was born in Roosendaal, The Netherlands. He received the M.Sc. and Ph.D. degrees in electrical engineering from the Eindhoven University of Technology, The Netherlands, in 1981 and 1984, respectively.

He became a Senior Research Member in the design automation group of the Eindhoven University of Technology in 1985. In 1986, he spent a sabbatical period at the IBM Thomas J. Watson Research Laboratory, Yorktown Heights, NY, for research on high level synthesis. In 1998, he joined Philips Research Laboratories in Eindhoven, The Netherlands, to work on the architectural design of programmable multimedia hardware and the associated mapping of media processing applications.

Kees A. Vissers received the M.Sc. degree from Delft University of Technology, The Netherlands, in 1980.

He worked with Philips Research Laboratories, Eindhoven, The Netherlands, where he was involved in high-level simulation and high-level synthesis. He had been heading the research on hardware/software codesign and system level design for many years, and had a significant contribution to the TriMedia VLIW processor. From 1987 to 1988, he was a Visiting Researcher at Carnegie Mellon University, Pittsburgh, PA, with the group of D. Thomas. He is currently a Research Fellow with , Department of Electrical Engineering and Computer Sciences, University of California at Berkeley. His research interests include video processing, embedded media processing systems, and reconfigurable computing.