J Sign Process Syst (2010) 60:183–210
DOI 10.1007/s11265-008-0304-5

Optimizing the H.264/AVC Video Encoder Application Structure for Reconfigurable and Application-Specific Platforms

Muhammad Shafique · Lars Bauer · Jörg Henkel

Received: 15 April 2008 / Accepted: 16 October 2008 / Published online: 21 November 2008

Abstract The H.264/AVC video coding standard features diverse computational hot spots that need to be accelerated to cope with the significantly increased complexity compared to previous standards. In this paper, we propose an optimized application structure (i.e. the arrangement of functional components of an application determining the data flow properties) for the H.264 encoder which is suitable for application-specific and reconfigurable hardware platforms. Our proposed application structural optimization for the computational reduction of the *Motion Compensated Interpolation* is independent of the actual hardware platform that is used for execution. For a MIPS processor we achieve an average speedup of approximately 60× for *Motion Compensated Interpolation*. Our proposed application structure reduces the overhead for *Reconfigurable Platforms* by distributing the actual hardware requirements amongst the functional blocks. This increases the amount of available reconfigurable hardware per *Special Instruction* (within a functional block) which leads to a 2.84× performance improvement of the complete encoder when compared to a *Benchmark Application* with standard optimizations. We evaluate our application structure by means of four different hardware platforms.

Keywords H.264 · MPEG-4 AVC · Motion compensation · Motion estimation · Rate distortion · In-loop de-blocking filter · ASIP · Reconfigurable platform · RISPP · Special instructions · Hardware accelerators

1 Introduction and Motivation

The growing complexity of next generation mobile multimedia applications and the increasing demand for advanced services stimulate the need for high-performance embedded systems. Typically, for real-time 30 fps (33 ms/frame) video communication at *Quarter Common Intermediate Format* (176×144) resolution, a video encoder has a 20 ms (60%) time budget. The remaining 40% of the time budget is given to video decoder, audio codec, and the multiplexer. Due to this tight timing constraint, a complex encoder requires high performance using both application structure and hardware platform while keeping high video quality, low transmission bandwidths, and low storage capacity. An application structure is defined as the organization of functional/processing components of an application that determine the properties of data flow.

H.264/MPEG-4 AVC [1] from the Joint Video Team (JVT) of the ITU-T VCEG and ISO/IEC MPEG is one of the latest video coding standards and aims to address these constraints. Various studies have shown that it provides a

This paper is an extended version of our ESTIMedia'07 paper. We have significantly extended (more than 50%) our ESTIMedia'07 paper by adding (a) detailed discussions of the proposed optimizations and a detailed diagram of the final optimized application structure, (b) a new section presenting a comprehensive data flow diagram and data structure formats with a memory-related discussion, (c) *Special Instruction* for *De-blocking Filter*, (d) extending the presented results with new figures and tables, (e) new section describing the optimization steps to create the *Benchmark Application*, (f) A new sub-section with *Functional Description* of all *Special Instructions* with constituting data paths, and (g) an extended overview of different hardware platforms used for benchmarking.

M. Shafique (✉) · L. Bauer · J. Henkel
Chair for Embedded Systems, University of Karlsruhe, Karlsruhe, Germany
e-mail: shafique@informatik.uni-karlsruhe.de

L. Bauer
e-mail: lars.bauer@informatik.uni-karlsruhe.de

J. Henkel
e-mail: henkel@informatik.uni-karlsruhe.de

bit-rate reduction of 50% as compared to MPEG-2 with the same subjective visual quality [5, 6] but at the cost of additional computational complexity (~10× relative to MPEG-4 simple profile encoding, ~2× for decoding [7]). The H.264 encoder uses a complex feature set to achieve better compression and better subjective quality [6, 7]. Each functional block of this complex feature set contains multiple (computational) hot spots hence a complete mobile multimedia application (e.g. H.324 video conferencing application) will result in many diverse hot spots (instead of only a few). However, H.264/AVC is the biggest component of the H.324 application and requires large amount of computation, which makes it difficult to achieve real-time processing in software implementation.

The main design challenges faced by embedded system designers for such mobile multimedia applications are: reducing chip area, increasing application performance, reducing power consumption, and shortening time-to-market. Traditional approaches e.g. *Digital Signal Processors* (DSPs), *Application Specific Integrated Circuits* (ASICs), *Application-Specific Instruction Set Processors* (ASIPs), and *Field Programmable Gate Arrays* (FPGAs) do not necessarily meet all design challenges. Each of these has its own advantages and disadvantages, hence fails to offer a comprehensive solution to next generation complex mobile applications' requirements. DSPs offer high flexibility and a lower design time but they may not satisfy the area, power, and performance challenges. On the other hand, ASICs target specific applications where the area and performance can be optimized specifically. However, the design process of ASICs is lengthy and is not an ideal approach considering short time-to-market. H.264 has a large set of tools to support a variety of applications (e.g. low bit-rate video conferencing, recording, surveillance, HDTV, etc.). A generic ASIC for all tools is impractical and will be huge in size. On the other hand, multiple ASICs for different applications have a longer design time and thus an increased *Non-Recurring Engineering* (NRE) cost. Moreover, when considering multiple applications (video encoder is just one application) running on one device, programmability is inevitable (e.g. to support task switching). ASIPs overcome the shortcomings of DSPs and ASICs, with an application-specific instruction set that offers a high flexibility (than ASICs) in conjunction with a far better efficiency in terms of performance per power, performance per area (compared to GPP and DSPs). Tool suites and architectural IP for embedded customizable processors with different attributes are available from major vendors like Tensilica [13], CoWare [15], and ARC [16]. ASIPs provide dedicated hardware for each hot spot hence resulting in a large area. While scrutinizing the behavior of H.264 video encoder, we have noticed that these hot spots are not active at same time. A more efficient approach to target such kind of applications that are quite large and have a changing flow within a tight timing constraint is a dynamically reconfigurable architecture with customized hardware (see Section 7).

In a typical ASIP development flow, first the H.264 reference software [2] is adapted to contain only the required tools for the targeted profile (*Baseline-Profile* in our case) and basic optimizations for data structures are performed. An optimized low-complexity *Motion Estimator* is used instead of the exhaustive *Full Search* of the reference software. Then, this application is profiled to find the computational hot spots. For each hot spot, *Special Instructions* (SIs) are designed and then integrated in the application. After all these enhancements, we take this optimized application as our *Benchmark Application* for discussion and comparison. Section 3 presents the details of optimizations steps to create the *Benchmark Application* and explains why comparing against reference software (instead of comparing with an optimized application) would result in unrealistically high speedups. The functional arrangement inside the *Benchmark Application* architecture is still not optimal and has several deficiencies. This paper targets these shortcomings and proposes architectural optimizations. Now we will motivate the need for these optimizations with the help of an analytical study.

Figure 1 shows the functional blocks (hot spots) of the Benchmark Application: Motion Compensation (MC), Motion Estimation (ME) using Sum of Absolute (Transformed) Differences (SA(T)D), Intra Prediction (IPRED), (Inverse) Discrete Cosine Transform ((I)DCT), (Inverse) Hadamard Transform ((I)HT), (Inverse) Quantization ((I)Q), Rate Distortion (RD), and Context Adaptive Variable Length Coding (CAVLC). These functional blocks operate at Macroblock-level (MB=16×16 pixel block) where an MB can be of type Intra (I-MB: uses IPRED for prediction) or Inter (P-MB: uses MC for prediction).

The H.264 encoder *Benchmark Application* interpolates MBs before entering the MB *Encoding Loop*. We have performed a statistical study on different mobile video sequences with low-to-medium motion. Figure 2 shows the observations for two representative video sequences. We have noticed that in each frame the number of MBs for

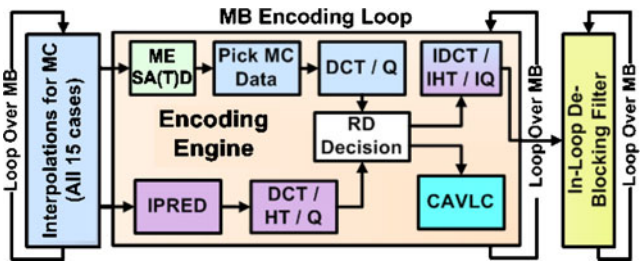

Figure 1 Arrangement of functional blocks inside the benchmark application of the H.264 video encoder.

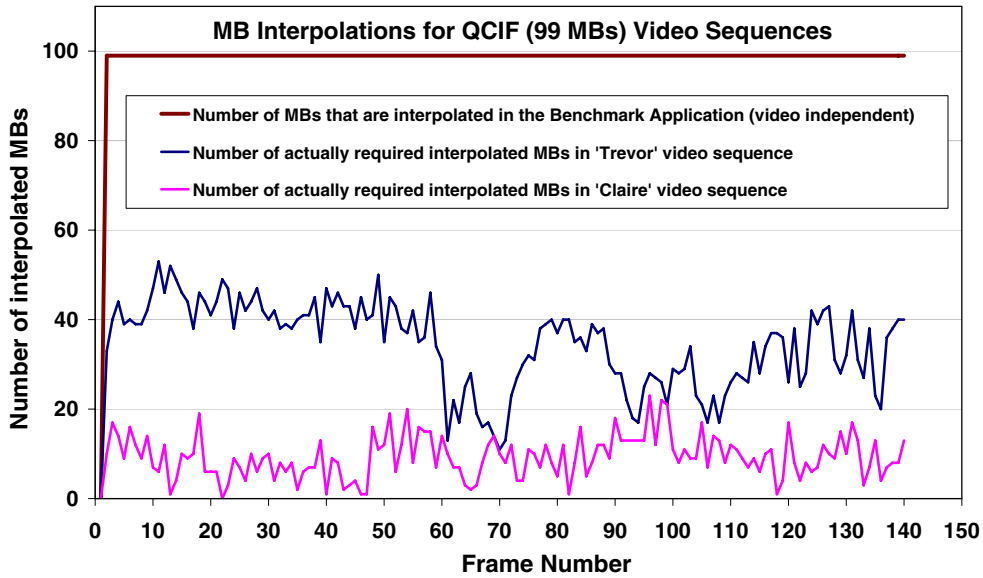

Figure 2 Number of computed vs. required interpolated MBs for two standard test sequences for mobile devices.

which an interpolation was actually required to process MC is much less than the number of MBs processed for interpolation by the *Benchmark Application*. After analysis, we found that the significant gap between the processed and the actually required interpolations is due to the stationary background, i.e. the motion vector (which determines the need for interpolations) is zero.

One of our contributions in this paper is to eradicate this problem by shifting the process of interpolation after the ME computation. This enables us to determine and process only the required interpolations, as it is explained in Section 4. It will save computational time for both *General Purpose Processor* and other hardware platforms with application accelerators without affecting the visual quality. However, as a side effect, this approach increases the number of functional blocks inside the MB *Encoding Loop*.

Altogether, we evaluate our proposed application structure by mapping it to the following four different processor types:

- GPP: General Purpose Processor, e.g. MIPS/ARM
- ASIP: Application-Specific Instruction Set Processor, e.g. Xtensa from Tensilica [13]
- Reconfigurable Platform: Run-time reconfigurable processor with static reconfiguration decisions, e.g. Molen [52]
- Self-adaptive reconfigurable processor (e.g. RISPP: Rotating Instruction Set Processing Platform, see Section 7).

Note: These hardware platforms are not the focus of this work. The focus of the work is the application structural optimizations for H.264 video encoder. An application structure running on a particular hardware platform is compared with different application structures on the same hardware platform. To keep the comparison fair all the application structures get the same set of *Special Instructions* and data paths.

ASIPs may offer a dedicated hardware implementation for *each* hot spot but this typically requires a large silicon footprint. Still, the sequential execution pattern of the application execution may only utilize a certain portion of the additionally provided hardware accelerators at any time. The *Reconfigurable Platforms* overcome this problem by re-using the hardware in time-multiplex. In Fig. 1, first, the hardware is reconfigured for *Interpolation* and while reconfiguring, the *Special Instructions* (SIs) for *Interpolation* are executed in software (i.e. similar to GPP). After the reconfiguration is finished, the *Interpolation* SIs execute in hardware. Once the *Interpolation* for the whole frame is done, the hardware is reconfigured for the MB *Encoding Loop* and subsequently it is reconfigured for the in-loop *Deblocking Filter* (see Fig. 1). It is important to note that due to a high reconfiguration time, we cannot reconfigure in between the processing of a hot spot i.e. within processing of each MB. We noticed that there are several functional blocks inside the MB *Encoding Loop*, and not all data paths for SIs of the MB *Encoding Loop* can be supported in the available reconfigurable hardware. The bigger number of data paths required to expedite a computational hot spot corresponds to a high *hardware pressure* inside this hot spot (i.e. a high amount of hardware that has to be provided to expedite the hot spot). A higher *hardware pressure* results in:

(a) more application-specific accelerators that might be required (for performance) within a computational hot spot than actually fit into the reconfigurable hardware. Therefore, not hot spots might be expedited by hardware but have to be executed in software (similar to GPP) instead, and

(b) increased reconfiguration overhead, as the reconfiguration time depends on the amount of hardware that needs to be reconfigured.

Both points lead to performance degradations for *Reconfigurable Platforms*, depending on the magnitude of *hardware pressure*. This is a drawback for the class of *Reconfigurable Platforms* and therefore we propose a further optimization in the application structure to counter this drawback.

Our novel contributions in a nutshell:

- Application structural optimizations for reduced processing by offering an on-demand MB interpolation (Section 4).
- Application structural optimizations to reduce the *hardware pressure* inside the MB *Encoding Loop* by decoupling the *Motion Estimation* and *Rate Distortion* (Section 5).
- Optimized data paths and the resulting *Special Instruction* for the main computational hot spots of the H.264 encoder that are implemented in hardware and used for benchmarking our optimized application structure (Section 6).

The rest of the paper is organized as follows: related work is presented in Section 2. Section 3 gives details for creating our *Benchmark Application*. We present our application structural optimization for reduced interpolation in Section 4 along with the impact on memory and cache accesses. Section 5 sheds light on the application structural optimizations for reduced *hardware pressure* considering *Motion Estimation* and *Rate Distortion* along with the discussion on its impact on the application data flow in Section 5.1. We discuss the optimized data paths and the resulting *Special Instructions* for the main computational hot spots of the H.264 encoder in Section 6. The details for *Special Instruction* and optimized data paths of *De-blocking Filter* are discussed in Section 6.1. The properties of different hardware platforms used for benchmarking are explained in Section 7. The detailed discussion and evaluation of our proposed optimizations are presented in Section 8 with an in-sight of how we achieve the overall benefit. We conclude our work in Section 9.

2 Related Work

In the following, we discuss three different types of prominent related work for the H.264 video codec: (a) hardware/(re)configurable solutions, (b) dedicated hardware for a particular component, and (c) algorithmic optimizations.

A hardware design methodology for H.264/AVC video coding system is described in [17]. In this methodology, five major functions are extracted and mapped onto a four stage *Macroblock* (MB) pipelining structure. Reduction in the internal memory size and bandwidth is also proposed using a hybrid task-pipelining scheme. However, some encoding hot spots (e.g. MC, DCT, and Quantization) are not considered for hardware mapping. An energy efficient, instruction cell based, dynamically reconfigurable fabric combined with ANSI-C programmability, is presented in [18, 19]. This architecture claims to combine the flexibility and programmability of DSP with the performance of FPGA and the energy requirements of ASIC. In [20] the authors have presented the XPP-S (Extreme Processing Platform-Samsung), an architecture that is enhanced and customized to suit the needs of multimedia application. It introduces a run-time reconfigurable architecture PACT-XPP that replaces the concept of instruction sequencing by configuration sequencing [21, 22]. In [23, 24], and [25] the authors have mapped an H.264 decoder onto the ADRES coarse-grained reconfigurable array. In [23] and [24] authors have targeted IDCT and in [25] MC optimizations are proposed using loop coalescing, loop merging, and loop unrolling, etc. However, at the encoder side the scenario is different from that in decoder, because the interpolation for Luma component is performed on frame-level. Although the proposed optimizations in [25] expedite the overall interpolation process, this approach does not avoid the excessive computations for those MBs that lie on integer-pixel boundary.

A hardware co-processor for real time H.264 video encoding is presented in [26]. It provides only *Context Adaptive Binary Arithmetic Coding* (CABAC) and ME in two different co-processors thus offers partial performance improvement. A major effort has been spent on individual blocks of the H.264 codec e.g. DCT ([33–35]) ME ([36–39]), and *De-blocking Filter* ([40–44]). Instead of targeting one specific component, we have implemented 12 hardware accelerators (see Section 3) for the major computational-intensive parts of the H.264 encoder and used them for evaluating our proposed application structure in the result section.

Different algorithmic optimizations are presented in [27–32] for the reduction of computational complexity. [27] introduces an early termination algorithm for variable block size ME in H.264 while giving a *Peak Signal to Noise Ratio* (PSNR: metric for video quality measurement) loss of 0.13 dB (note: a loss of 0.5 dB in PSNR results in a visual quality degradation corresponding to 10% reduced bit-rate [9]) for *Foreman* video sequence along with an increase of 4.6% in bit rate. Several algorithmic optimizations for the *Baseline-Profile* of H.264 are presented in [28]. These optimizations include adaptive diamond pattern based ME, sub-pixel ME, heuristic *Intra Prediction*, loop unrolling, "early out" thresholds, and adaptive inverse transforms. It gives a speedup of 4× at the cost of approximately 1 dB for *Carphone* CIF video sequence. A set of optimizations

related to transform and ME is given in [29]. This constitutes avoiding transform and inverse transform calculations depending upon the SAD value, calculation of *Intra Prediction* for four most probable modes, and fast ME with early termination of skipped MBs. This approach gives a PSNR loss of 0.15 dB. [30] detects all-zero coefficient blocks (i.e. all coefficients having value 'zero') before actual transform operation using SAD in an H.263 encoder. However, incorrect detection results in loss of visual quality. [31] suggests all-zero and zero-motion detection algorithms to reduce the computation of ME by jointly considering ME, DCT, and quantization. The value of SAD is compared against different thresholds to determine the stopping criterion. The computation reduction comes at the cost of 0.1 dB loss in PSNR. Two methods to reduce computation in DCT, quantization, and ME in H.263 are proposed in [32] that detect all-zero coefficients in MBs while giving a 0.5 dB PSNR loss and 8% increase in bit-rate. It uses the DC coefficient of DCT as an indicator.

In short, these optimizations concentrate on processing reduction in ME and/or DCT at the cost of proportional quality degradation (due to false early termination of the ME process or false early detection of blocks with non-zero quantized coefficient), but they did not consider other computational intensive parts. We instead reduce the processing load at functional level by avoiding advance and extra processing and a reduced *hardware pressure* in the MB *Encoding Loop* by decoupling processing blocks. In our test applications, we use an optimized low-complexity *Motion Estimator* to accentuate the optimization effect of other functional blocks. After ME load reduction, MC is the next bigger hot spot. Therefore, this paper rather considers MC and *hardware pressure*.

Additionally, we have proposed an optimized data path for *De-blocking Filter* and now we will discuss some related work for this. [40] uses a 2×4×4 internal buffer and 32×16 internal SRAM for buffering operation of *De-blocking Filter* with I/O bandwidth of 32-bits. All filtering options are calculated in parallel while the condition computation is done in a control unit. The paper uses 1-D reconfigurable FIR filter (8 pixels in and 8 pixels out) but does not target the optimizations of actual filter data path. It takes 232 cycles/MB. [41] introduces a five-stage pipelined filter using two local memories. This approach suffers with the overhead of multiplexers to avoid pipeline hazards. It costs 20.9 K *Gate Equivalents* for 180 nm technology and requires 214–246 cycles/MB. A fast *De-blocking Filter* is presented in [42] that uses a data path, a control unit, an address generator, one 384×8 register file, two dual port internal SRAMs to store partially filtered pixels, and two buffers (input and output filtered pixels). The filter data path is implemented as a two-stage pipeline. The first pipeline stage includes one 12-bit adder and two shifters to perform numerical calculations like multiplication and addition. The second pipeline stage includes one 12-bit comparator, several two's complementers and multiplexers to determine conditional branch results. In worst case, this technique takes 6,144 clock cycles to filter one MB. A pipelined architecture for the *De-blocking Filter* is illustrated in [43] that incorporates a modified processing order for filtering and simultaneously processes horizontal and vertical filtering. The performance improvement majorly comes from the reordering pattern. For 180 nm synthesis this approach costs 20.84 K *Gate Equivalents* and takes 192 (memory)+160 (processing) cycles. [44] maps the H.264 *De-blocking Filter* on the ADRES coarse-grained reconfigurable array ([23, 24]). It achieves 1.15×and 3× speedup for overall filtering and kernel processing respectively.

We are different from the above approaches because we target first the optimization of core filtering data paths in order to reduce the total number of primitive operations in one filtering. In addition to this, we collapse all conditions in one data path and calculate two generic conditions that decide the filtering output. We incorporate a parallel scheme for filtering one 4-pixel edge in eight cycles (see Section 6) where each MB has 48 (32 *Luma*+16 *Chroma*) 4-pixel edges. For high throughput, we use two 128-bit load/store units (as e.g. Tensilica [13] is offering for their cores [14]).

3 Optimization Steps to Create the Benchmark Application

The H.264/AVC reference software contains a large set of tools to support a variety of applications (video conferencing to HDTV) and uses complex data structures to facilitate all these tools. For that reason, the reference software is not a suitable base for optimizations. We therefore spent a tremendous effort to get a good starting point for our proposed optimizations. A systematic procedure to convert the reference C code of H.264 into a data flow model is presented in [45] that can be used for different design environments for DSP systems. We have handled this issue in a different way i.e. from the perspective of ASIPs and *Reconfigurable Platforms*. In order to achieve the *Benchmark Application*, the H.264 reference software [2] is passed through a series of optimization steps to improve the performance of the application on the targeted hardware platforms. The details of these optimizations are as follows:

(a) First, we adapted the reference software to contain only *Baseline-Profile* tools (Fig. 3a) considering multimedia applications with *Common Intermediate Format* (CIF=352×288) or *Quarter CIF* (QCIF=176×144) resolutions running on mobile devices. We

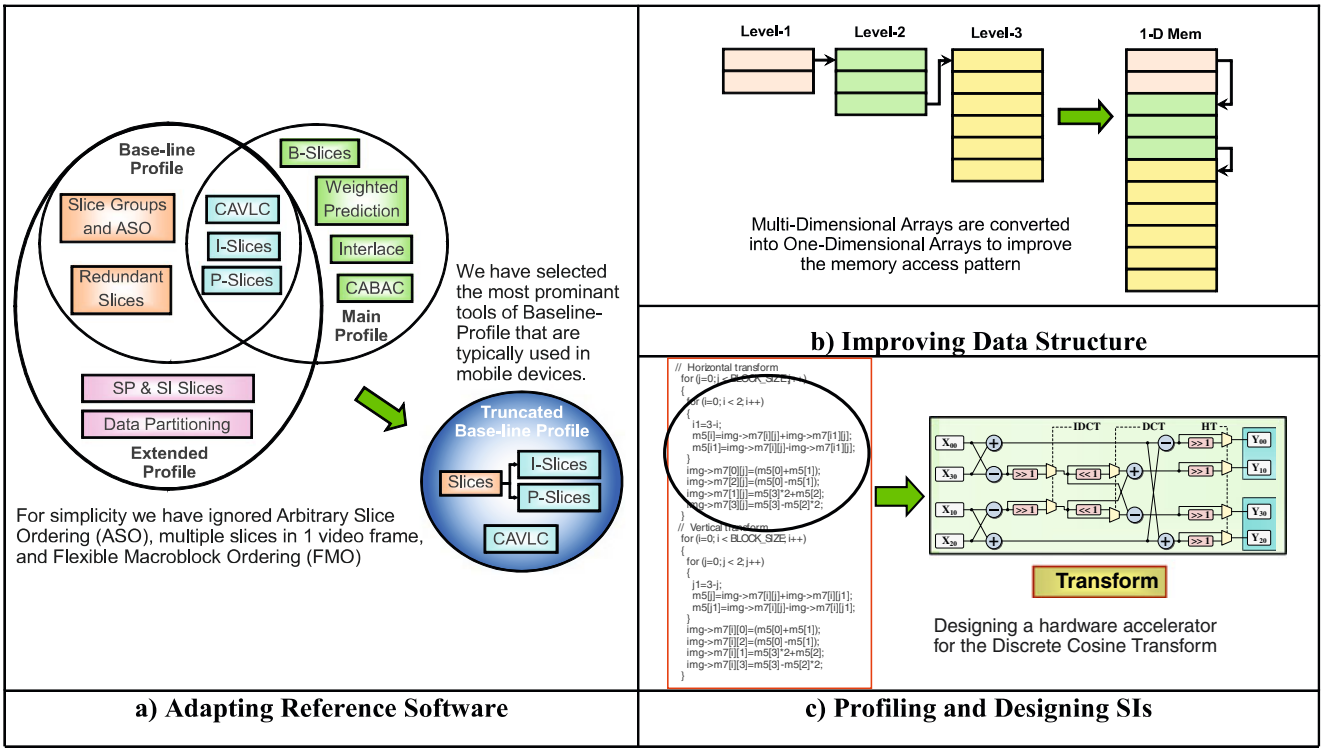

Figure 3 Steps to construct the benchmark application.

further truncated/curtailed the *Baseline-Profile* by excluding *Flexible Macroblock Ordering* (FMO) and multiple slice (i.e. complete video frame is one slice).

(b) Afterwards, we improved the data structure of this application by replacing e.g. multi-dimensional arrays with one-dimensional arrays to improve the memory accesses (Fig. 3b). We additionally improved the basic data flow of the application and unrolled the inner loops to enhance the compiler optimization space and to reduce the amount of jumps.

(c) Enhanced *Motion Estimation* (ME) in H.264 comprises variable block size ME, sub-pixel ME up to quarter pixel accuracy, and multiple reference frames. As a result, the ME process may consume up to 60% (one reference frame) and 80% (five reference frames) of the total encoding time [4]. Moreover, the reference software uses a *Full Search Motion Estimator* that is not practicable in real-world applications and is used only for quality comparison. Therefore, real-world applications necessitate a low-complexity *Motion Estimator*. We have used a low-complexity *Motion Estimator* called *UMHexagonS* [4] (also used in other publically available H.264 sources e.g. *x264* [3]) to reduce the processing loads of ME process while keeping the visual quality closer to that of *Full Search*. *Full Search* requires on average 107811 SADs/frame for *Carphone* QCIF video sequence (256 kbps, 16 *Search Range* and 16×16 *Mode*). On the contrary, *UMHexagonS* requires only 4424 SADs/frame. Note: *Special Instructions* (SIs) are designed to support this *Motion Estimator* in hardware (Section 6) and same SIs are used for all application structures to keep the comparison fair. After optimizing the *Motion Estimator*, other functional blocks become prominent candidates for optimizations.

(d) Afterwards, this application is profiled to detect the computational hot spots. We have designed and implemented several *Special Instructions* (composed of hardware accelerators as shown in Table 1) to expedite these computational hot spots (Fig. 3c). This adapted and optimized application then serves as our *Benchmark Application* for further proposed optimizations. We simulated it for a MIPS-*based processor* (GPP), an ASIP, a *Reconfigurable Platform*, and RISPP while offering the same SI implementations.

Note optimizations (a–c) are good for GPP, while optimizations (a–d) are good for all other hardware platforms i.e. ASIPs, *Reconfigurable platforms*, and RISPP.

Table 1 gives the description of implemented SIs of H.264 video encoder that are used to demonstrate our optimized application structure. Section 6 presents the functional description of these *Special Instructions* along with the constituting data paths. All hardware platforms use the same set of SIs to accentuate only the effect of application structural optimizations.

Table 1 Implemented special instructions and data paths for the major functional components of H.264 video encoder.

Functional component	Special instruction	Description of special instructions	Accelerating data paths
Motion estimation (ME)	SAD	Sum of absolute differences of a 16×16 macroblock	SAD_16
	SATD	Sum of absolute transformed differences of a 4×4 sub-block	QSub, Transform, Repack, SAV
Motion compensation (MC)	MC_Hz_4	Motion compensated interpolation for horizontal case for 4 pixels	PointFilter, BytePack, Clip3
Intra prediction (IPred)	IPred_HDC	16×16 intra prediction for horizontal and DC	PackLBytes, CollapseAdd
	IPred_VDC	16×16 intra prediction for vertical and DC	CollapseAdd
(Inverse) transform	(I)DCT	Residue calculation and (inverse) discrete cosine transform for 4×4 sub-block	Transform, Repack, (QSub)
	(I)HT_2×2	2×2 (inverse) Hadamard transform of Chroma DC coefficients	Transform
	(I)HT_4×4	4×4 (inverse) Hadamard transform of intra DC coefficients	Transform, Repack
Loop filter (LF)	LF_BS4	4-pixel edge filtering for in-loop de-blocking filter with boundary strength 4	Cond, LF_4

4 Application Structural Optimization for Interpolation

As motivated in Fig. 2, the H.264 encoder *Benchmark Application* performs the interpolation for all MBs, although it is only required for those with a certain motion vector value (given by the *Motion Estimation* ME). Additionally, even for those MBs that require an interpolation, only one of the 15 possible interpolation cases is actually required (indeed one interpolation case is needed per *Block*, potentially a sub-part of a MB), which shows the enormous saving potential. The last 2 bits of the motion vector hereby determine the required interpolation case.

Figure 4 shows the distribution of interpolation cases in 139 frames of the *Carphone* sequence, which is the standard videophone test sequence with the highest interpolation computation load in our test-suite (see results in Section 8.2). Figure 4 demonstrates that in total 48.78% of the total MBs require one of these interpolation cases (C-1 to C-15). The case C-16 is for those MBs where the last 2 bits of the motion vector are zero (i.e. integer pixel resolution or stationary) such that no interpolation is required. The I-MBs (for *Intra Prediction*) actually do not require an interpolation either.

Figure 5 shows our optimizations to reduce the overhead of excessive interpolations in the *Benchmark Application*. After performing the *Motion Estimation* (ME) (lines 3–5), we obtain the motion vector, which allows us to perform only the required interpolation (line 7). The Sub-Pixel ME

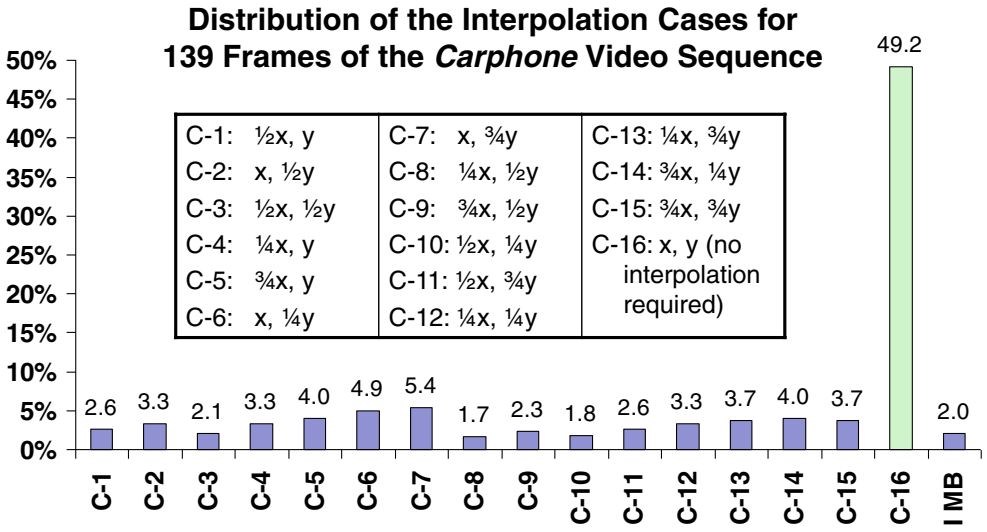

Figure 4 Distribution of interpolation cases.

```
1.   // for simplicity of demonstration we are considering only 16x16 Mode, but the concept is orthogonal to all Modes.
2.   FOR all MBs in frame DO                    // MB Encoding Loop
3.     Perform ME for all reference frames and all Modes
4.     a) Perform Integer ME
5.     b) Perform Sub-Pixel ME
6.     FOR LUMA and CHROMA DO
7.     Perform the required Interpolation for MC    // P-MB
8.     Perform DCT and Quantization for P-MB
9.     Perform IDCT and Inv. Quantization for P-MB
10.    Perform IPRED          // I-MB
11.    Perform DCT, HT and Quantization for I-MB
12.    Perform IDCT, IHT and Inv. Quantization for I-MB
13.    RD: Decide for I- or P-MB Type and its Mode
14.    END FOR
15.    Perform CAVLC // Both Luma and Chroma
16.  END FOR
17.  FOR all MBs in frame DO
18.    Perform Loop Filter// Both Luma and Chroma
19.  END FOR
```

Figure 5 Optimized application structure of the H.264 encoder for on-demand interpolation.

(line 5) might additionally require interpolations, but it is avoided in most of the cases (C-16) due to the stationary nature of these MBs. Our proposed application structure maintains the flexibility for the designer to choose any low-complexity interpolation scheme for Sub-Pixel ME e.g. [39].

4.1 Memory-Related Discussion

Now we will discuss different memory related issues for cases of pre-computation and our on-demand interpolation.

(a) In case of on-demand interpolation scheme, we only need storage for 256 pixels (1 MB), as we exactly compute one interpolation case per MB (even for sub-pixel ME we can use an array of 256 interpolated pixels and then reuse it after calculation of each SATD). The same storage will be reused by other MBs as the interpolated values are no more required after reconstruction. Pre-computing all interpolated pixels up to quarter-pixel resolution instead needs to store all interpolated values of one video frame in a big memory of size 16×(size of one video frame) bytes. For QCIF (176×144) and CIF (352×288) resolution this corresponds to a 1,584 (176×144×16/256) and 6,336 (352×288×16/256) times bigger memory consumption, respectively, compared to our on-demand interpolation.

(b) Pre-computing all interpolation cases results in non-contiguous memory accesses. The interpolated frame is stored in one big memory, i.e. interpolated pixels are placed in between the integer pixel location. Due to this reason, when a particular interpolation case is called for *Motion Compensation*, the access to the pixels corresponding to this interpolation case is in a non-contiguous fashion (i.e. one 32-bit load will only give one useful 8-bit pixel value). This will ultimately lead to data cache misses as the data cache will soon be filled with the interpolated frame i.e. including those values too that were not required. On the other hand, our on-demand interpolation stores the interpolated pixels in an intermediate temporary storage using a contiguous fashion i.e. four interpolated pixels of a particular interpolation case are stored contiguously in one 32-bit register. This improves the overall memory access behavior.

(c) Our on-demand computation improves the data flow as it directly forwards the interpolated result for residual calculation (i.e. difference of current and prediction data) and then to DCT (as we will see in Section 5.1). Registers can be used to directly forward interpolated pixels. On the contrary, pre-computation requires big memory storage after interpolation and loading for residual calculation.

(d) Pre-computation is beneficial in-terms of instruction cache as it processes a similar set of instructions in one loop over all MBs. Conversely, on-demand interpolation is beneficial in-terms of data-cache which is more critical for data intensive applications (e.g. video encoder).

Our proposed optimization for on-demand interpolation is beneficial for all GPP, ASIPs, *Reconfigurable Platforms*, and RISPP.

5 Application Structural Optimization for Reducing the Hardware Pressure

As motivated in Fig. 1, there is a high *hardware pressure* in the MB *Encoding Loop* of the H.264 encoder *Benchmark Application*. Although the application structural optimization presented in Section 4 results in a significant reduction of performed interpolations, it further increases the *hardware pressure* of the MB *Encoding Loop*, as the hardware for the *Motion Compensated Interpolation* is now shifted inside this loop. A higher *hardware pressure* has a negative impact when the encoder application is executed on a *Reconfigurable Platform*. This is due to the fact that the amount of hardware required to expedite the MB *Encoding Loop* (i.e. the *hardware pressure*) is increased and not all data paths can be accommodated in the available reconfigurable hardware. Moreover, it takes longer until the reconfiguration is completed and the hardware is ready to execute. Therefore, in order to reduce the *hardware pressure* we decouple those functional blocks that may be processed independent of rest of the encoding process. Decoupling of these functional blocks is performed with the surety that the encoding process does not deviate from the standard specification and a standard compliant bitstream is generated. We decouple *Motion Estimation* and *Rate Distortion* as they are non-normative and standard does not fix their implementation. However, this decoupling of functional blocks affects the data flow of application, as we will discuss later in Section 5.1.

Motion Estimation (ME) is the process of finding the displacement (motion vector) of an MB in the reference frame. The accuracy of ME depends upon the search technique of the *Motion Estimator* and the motion characteristics of the input sequence. As *Motion Estimator* does not depend upon the reconstructed path of encoder, ME can be processed independently on the whole frame. Therefore, we take it out of the MB *Encoding Loop* (as shown in Fig. 6) which will decouple the hardware for both integer and sub-pixel ME. Moreover, it is also worthy to note that some accelerating data paths of SATD (i.e. *QSub*, *Repack*, *Transform*) are shared by *(I)DCT*, *(I)HT_4×4*, and *(I)HT_2×2 Special Instructions* (see Table 1). Therefore, after the *Motion Estimation* is completed for one frame and the subsequent MB *Encoding Loop* is started, these reusable data paths are already available which reduces the reconfiguration overhead. As motion vectors are already stored in a full-frame based memory data structure, no additional memory is required when ME is decoupled from the MB *Encoding Loop*. Decoupling ME will also improve the instruction cache usage as same instructions are now processed for long time in one loop. A much better data-arrangement (depending upon the search patterns) can be performed to improve the data cache usage (i.e. reduced number of misses) when processing ME on *Image-level* due to the increased chances of availability of data in the cache. However, when ME executes inside the MB *Encoding Loop* these data-arrangement techniques may not help. This is because subsequent functional blocks (MC, DCT, CAVLC etc.) typically replace the data that might be beneficial for the next execution of ME.,

Rate Distortion (RD) and *Rate controller* (RC) are the tools inside a video encoder that control the encoded quality and bit-rates. Eventually their task is to decide about the *Quantization Parameter* (QP) for each MB and the type of MB (*Intra* or *Inter*) to be encoded. Furthermore, the *Inter-/Intra-Mode* decision is also attached with this as an additional RD decision layer. The H.264 *Benchmark Application* computes both I- and P-MB encoding flows with all possible *Modes* and then chooses the one with the best trade-off between the required bits to encode the MB and the distortion (i.e. video quality) using a *Lagrange Scheme*, according to an adjustable optimization goal.

We additionally take RD outside the MB *Encoding Loop* (see Fig. 6) to perform an early decision on MB type (I or P) and *Mode* (for I or P). This will further reduce the *hardware pressure* in the MB *Encoding Loop* and the total processing load (either I or P computation instead of both). Shifting RD is less efficient in terms of bits/MB as compared to the reference RD scheme as the latter checks all possible combinations to make a decision. However, RD outside the MB *Encoding Loop* is capable to utilize intelligent schemes to achieve a near-optimal solution e.g. *Inter-Modes* can be predicted using homogeneous regions and edge map [46]. *Rate Controllers* are normally two-layered (*Image-level* and *MB-level*). The *Image-level* RC monitors the bits per frame and selects a good starting QP value for the current frame hence provides a better convergence of the *MB-level* RC. The *MB-level* RC takes decision on current and/or previous frame motion properties (SAD value and motion vectors) therefore can be integrated in the *Motion Estimation* loop. Furthermore, texture, brightness, and *Human Visual System* (HVS: which provides certain hints about psycho-visual behavior of humans) properties may be used to integrate the RD inside the *MB-level* RC to make early *Mode* decisions. The processing description of RD and RC are beyond the scope of this paper but details can be found in [8, 10–12].

Our optimized application structure provides the flexibility for integration of any fast mode decision logic e.g. [46, 47]. Techniques like [46] detect the homogenous regions for mode decision, which can additionally be used by *MB-level* RC as a decision parameter. Mode decision of P-MB will be decided in ME loop while the mode of I-MB will be decided in the actual *Intra Prediction* stage. Moreover, fast *Intra Prediction* schemes e.g. [48] can also be incorporated easily in our proposed architecture but the

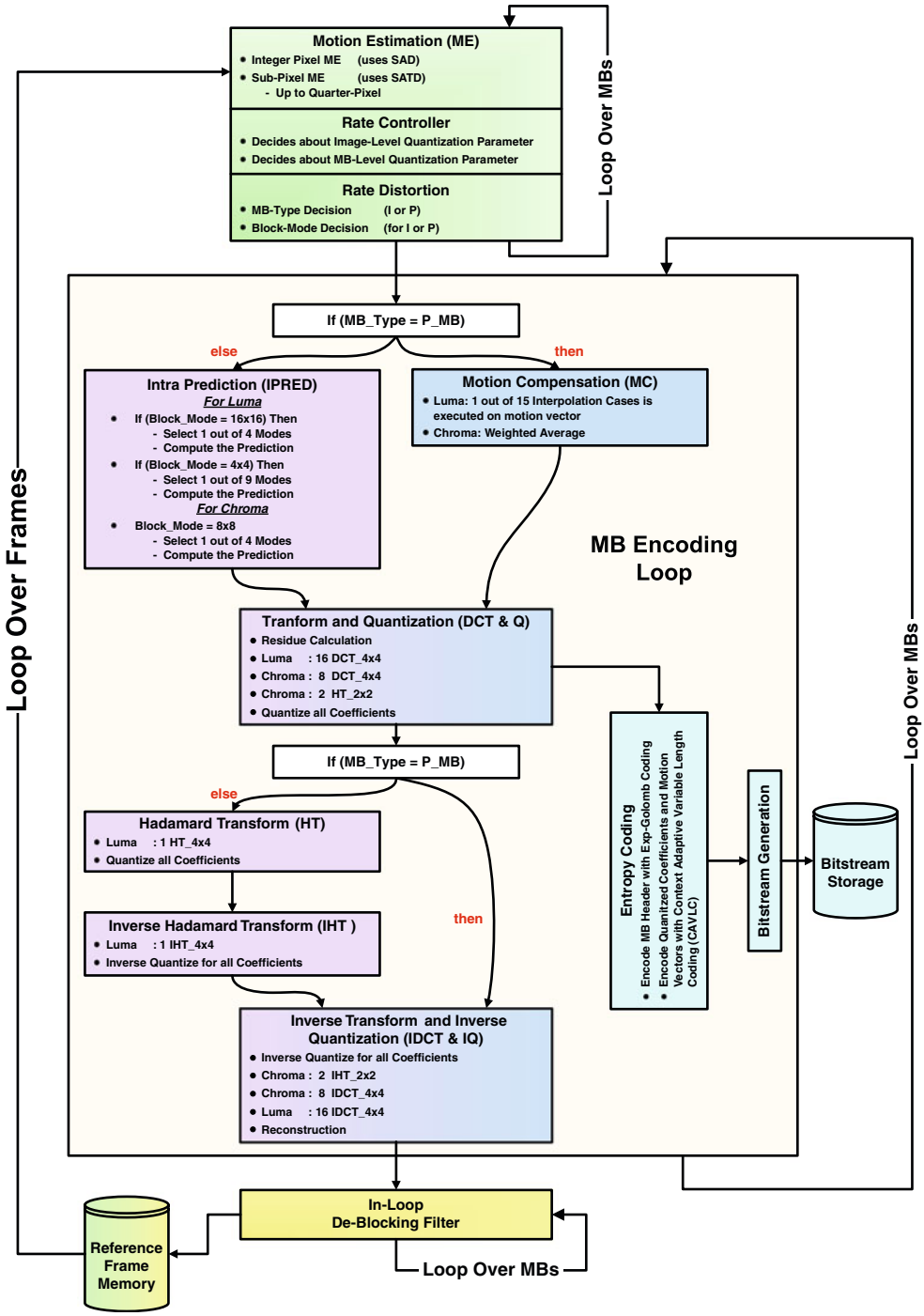

Figure 6 Optimized application structure of the H.264 encoder with reduced hardware pressure.

edge map will be calculated at *Image-level* and can also be used by RC.

We need an additional memory to store the type (i.e. I- or P-MB) of all MBs in a current frame after the RC decision which is equal to (Number of MBs in 1 Frame)× 1 bit. In actual implementation, we have considered 99×8= 792 bits (i.e. 99 bytes for QCIF) as we are using 'char' as the smallest storage type, but still it is a negligible overhead. Our optimized application structure with reduced *hardware pressure* provides a good arrangement of processing functions that facilitates an efficient data flow. For multimedia applications, data format/structure and data flow are very important as they greatly influence the resulting performance. Therefore, now we will discuss the complete data flow inside the encoder along with the impact of optimizations of the application structures.

5.1 Data Flow of the Optimized Application Structure of the H.264 Encoder with Reduced Hardware Pressure

Figure 7 shows the data flow diagram of our optimized application structure with reduced *hardware pressure*. The boxes show the process (i.e. the processing function of the encoder) and arrows represent the direction of the flow of data structure (i.e. text on these arrows). D2 and D2 are two data stores that contain the data structures for current and previous frames. E1 and E2 are two external entities to store the coding configuration and encoded bitstream, respectively. The format of these data structures is shown in Fig. 8 along with a short description.

Motion Estimation (1.0, 1.1) takes *Quantization Parameter* from *Rate Controller* (11.0), Luma components of current and previous frames (CurrY, PrevY) from two data stores D1 and D2 as input. It forwards the result i.e. MV and SAD arrays to the *Rate-Distortion* based *Mode-Decision*

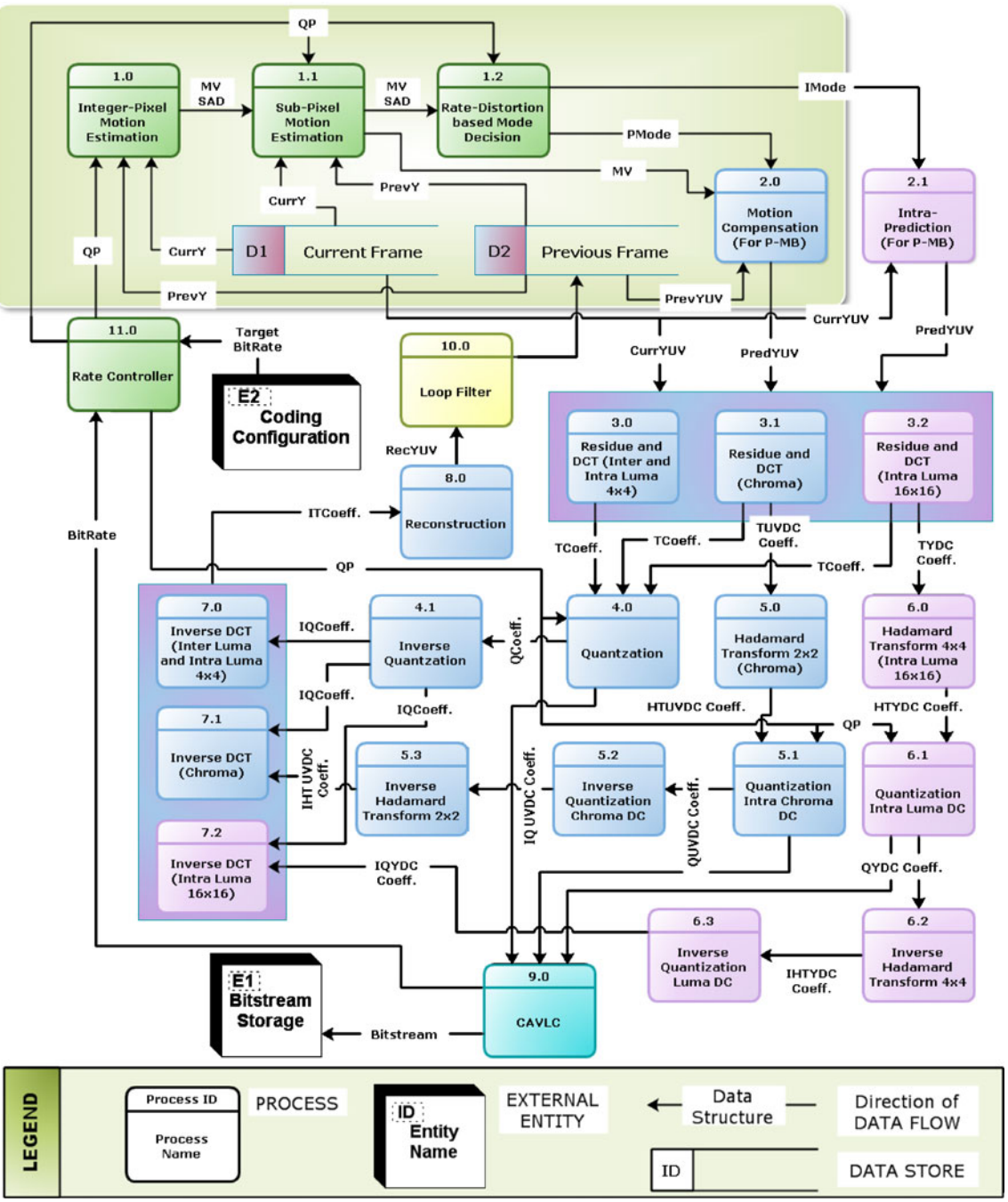

Figure 7 Data flow diagram of the optimized H.264 encoder application structure with reduced hardware pressure.

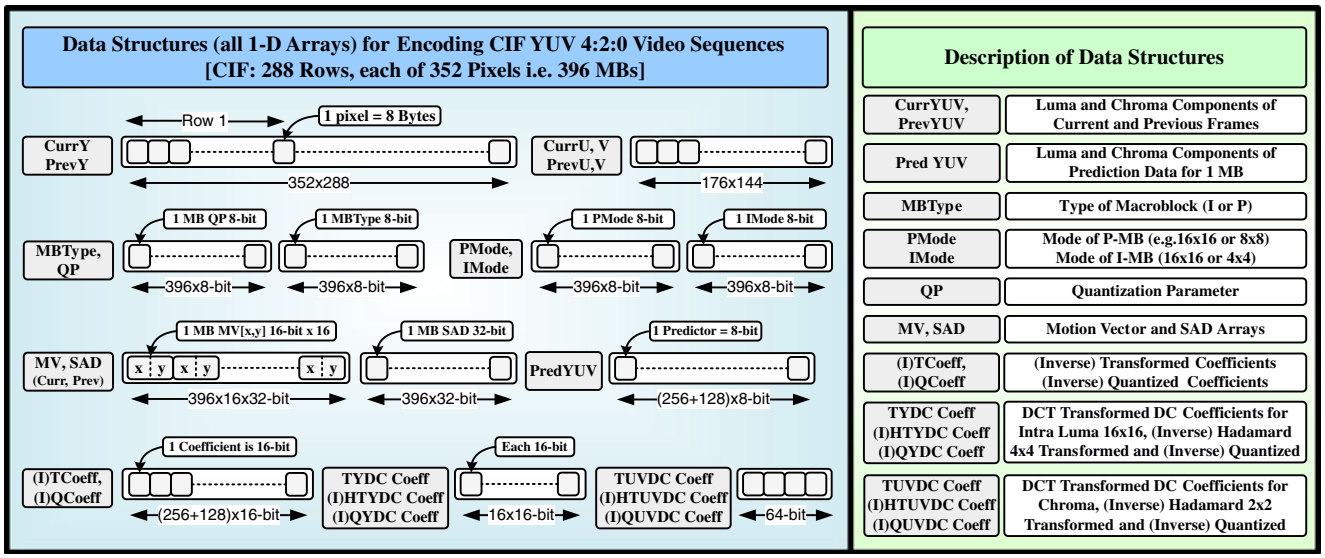

Figure 8 Description and organization of major data structures.

process (1.2) that selects the type of an MB and its expected coding mode. If the selected type of MB is *Intra* then the mode information (IMode) is forwarded to the *Intra Prediction* (2.1) block that computes the prediction using current frame (CurrYUV), otherwise PMode is forwarded to the *Motion Compensation* (2.0) that computes the prediction using previous frame (PrevYUV) and MV. The three transform processes (3.0–3.2) calculate the residue from using Luma and Chroma prediction results (PredYUV) and current frame data (CurrYUV) that is then transformed using 4×4 DCT. In case of *Intra Luma 16×16* the 16 DC coefficients (TYDC Coeff.) are further transformed using 4×4 *Hadamard Transform* (6.0) while in case of Chroma 4 DC coefficients (TUVDC Coeff.) are passed to 2×2 *Hadamard Transform* process (5.0). All the transformed coefficients (TCoeff, HTUVDC Coeff, HTYDC Coeff) are then quantized (4.0, 5.1, 6.1). The quantized result (QCoeff, QYDC Coeff, QUVDC Coeff) is forwarded to CAVLC (9.0) and to the backward path i.e. inverse quantization (6.3, 5.2, 4.1), inverse transform (6.2, 5.3, 7.0–7.2), and reconstruction (8.0). The reconstructed image is then processed with in-loop *De-blocking Filter* (10.0) while the output of CAVLC (i.e. bitstream) is stored in the *Bitstream Storage* (E1). Depending upon the achieved bit rate and coding configuration (E2) the Rate Controller (11.0) decides about the *Quantization Parameter.*

Optimizations of application structures change the data flow i.e. the flow of data structures from one processing function to the other. As the looping mechanism is changed, the data flow is changed. On the one hand performing on-demand interpolation increases the probability of instruction cache miss (as discussed in Section 4.1). However, on the other hand it improves the data cache by offering a smooth data flow between prediction calculation and transform process i.e. it improves the data flow as it directly forwards the interpolated result for residual calculation and then to DCT. After our proposed optimization, the size of data structure for interpolation result is much smaller than before optimization. The new PredYUV (Fig. 8) data structure requires only 384 bytes [(256+128)×8-bits] for CIF videos, as the prediction result for only one MB is required to be stored. On the contrary, pre-computation requires a big data structure (4×ImageSize i.e. 4×396×384 bytes) storage after interpolation and loading for residual calculation.

For reduced *hardware pressure* optimization, the *Motion Estimation* process is decoupled from the main MB *Encoding Loop*. Since now *Motion Estimation* executes in a one big loop, the instruction cache behavior is improved. The rectangular region in Fig. 7 shows the surrounded data structures whose flow is affected by this optimization of reduced *hardware pressure*. Before optimizing for reduced *hardware pressure*, *Motion Estimation* was processed on MB-level, therefore MV and SAD arrays were passed to the *Motion Compensation* process in each loop iteration. Since the encoder uses MVs of the spatially neighboring MBs for *Motion Estimation*, the data structure provides the storage for MVs of complete video frame (e.g. 396×32-bits for a CIF frame). After optimizing for reduced *hardware pressure*, there is no change in the size of MV and SAD data structures. The MV and SAD arrays of the complete video frame are forwarded just once to the *Motion Compensation* process.

Additionally now *Rate-Distortion* based *Mode-Decision* can be performed by analyzing the neighboring MVs and SADs. The type of MB and its prediction mode is stored at frame-level and is passed to the prediction processes.

Without our proposed optimizations i.e. when processing *Motion Estimation* and *Rate-Distortion* at MB-level, mode decision algorithms cannot use the information of MVs and SADs of the spatially next MBs. On the contrary our proposed optimized application structure facilitates much intelligent *Rate-Distortion* schemes where modes can be predicted using the motion properties of spatially next MBs too. Summarizing: our proposed optimizations not only save the excessive computations using on-demand interpolation for *Motion Compensation* and relax the *hardware pressure* in case of *Reconfigurable Platforms* by decoupling the *Motion Estimation* and *Rate-Distortion* processes but also improves the data flows and instruction cache behavior.

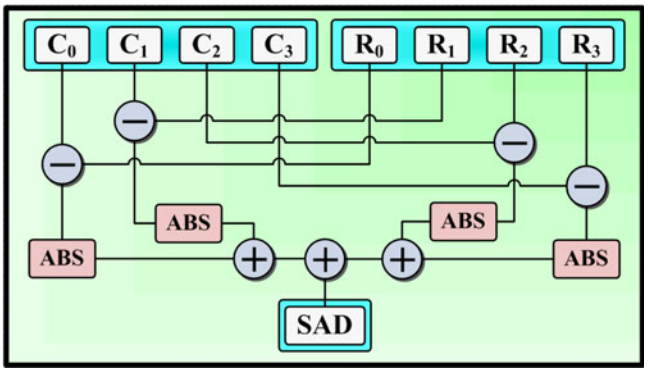

Figure 9 SAD_16 data path for SAD special instruction.

6 Functional Description of Special Instructions and Fast Data Paths

a. Motion estimation: sum of absolute differences and sum of absolute transformed differences [SA(T)D]

The ME process consists of two stages: *Integer-Pixel* search and *Fractional-Pixel* search. *Integer-pixel* ME uses *Sum of Absolute Differences* (SAD) to calculate the block distortion for a *Macroblock* (MB=16×16-pixel block) in the current video frame w.r.t. a MB in the reference frame at integer pixel resolution. For one MB in the current frame (F_t), various SADs are computed using MBs from the reference frame (e.g. immediately previous F_{t-1}). Equation 1 shows the SAD formula:

$$\mathrm{SAD} = \sum_{y=0}^{15}\sum_{x=0}^{15} |C(x,y) - R(x,y)| \tag{1}$$

where C is the current MB and R is the reference MB. One 16×16 SAD computation requires 256 subtractions, 256 absolute operations, 255 additions along with loading of 256 current and 256 reference MB pixels from memory. We have designed and implemented an SI that computes SAD of the complete MB. It constitutes two instances of the data path SAD_16 (as shown in Fig. 9) that computes SAD of 4 pixels of current MB w.r.t. 4 pixels of reference MB.

Once the best *Integer-pixel Motion Vector* (MV) is found, the *Sub-Pixel* ME process refines the search to fractional pixel accuracy. At this stage, due to high correlation between surrounding candidate MVs, the accuracy of minimum block distortion matters a lot. Therefore, H.264 proposes *Sum of Absolute Transformed Differences* (SATD) as the cost function to calculate the block distortion. It performs a 2-D *Hadamard Transform* on a 4×4 array of difference values to give a closer representation to DCT that is performed later in the encoding process. Due to this reason, SATD provides a better MV compared with that calculated using SAD. However, because of high computational load, SATD is only used in *Sub-Pixel* ME and not in *Integer-pixel* ME. The SATD operation is defined as:

$$\mathrm{SATD} = \sum_{y=0}^{4}\sum_{x=0}^{4} |\mathrm{HT}_{4\times4}\{C(x,y) - R(x,y)\}| \tag{2}$$

where C is the current and R is the reference pixel value of the 4×4 sub-block, and $\mathrm{HT}_{4\times4}$ is the 2-D 4×4 *Hadamard Transform* on a matrix D (the differences between current and reference pixel values) and it is defined as:

$$\mathrm{HT}_{4\times4} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} [D] \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \Big/ 2. \tag{3}$$

Our SATD SI uses four different types of data paths to perform a complete 4×4 SATD operation. All data paths have two 32-bit inputs and two 32-bit outputs. *QSub* (Fig. 10a) performs four subtractions; it takes eight unsigned 8-bit pixels P_i, Q_i, i=0…3 and returns four 16-bit signed residue outputs i.e. $R_i=P_i-Q_i$; for i=0…3. *Repack* (Fig. 10b) rearranges the 16-bit half-words of its 32-bit inputs by packing two 16-bit LSBs and two 16-bit MSBs in two 32-bit outputs. If $\mathrm{input}_1=X_1 \circ X_2$ and $\mathrm{input}_2=X_3 \circ X_4$, then $\mathrm{output}_1=X_1 \circ X_3$ and $\mathrm{output}_2=X_2 \circ X_4$. *Transform* (Fig. 10c) performs a four-point butterfly of *(Inverse) Discrete Cosine Transform* or *(Inverse) Hadamard Transform*. Four *Transform* data paths are used to perform a *Hadamard Transform* along one axis using only additions and subtractions. The second stage of this operation performs an additional arithmetical right-shift on the four results. SAV (Fig. 10d) computes the absolute values of its four 16-bit inputs and returns their sum. After the SAV data path, the four results are accumulated with three additions to complete the SATD SI.

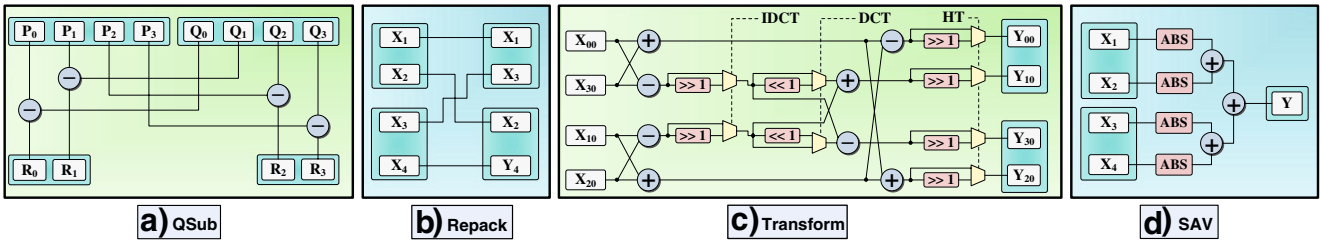

Figure 10 Data paths for SATD_4×4 special instruction.

b. Motion compensation (MC_Hz_4)

Each MB in a video frame is predicted either by the neighboring MBs in the same frame i.e. *Intra-Predicted* (I-MB) or by an MB in the previous frame i.e. *Inter-Predicted* (P-MB). This prediction is subtracted from the current block to calculate the residue that is then transformed, quantized, and entropy coded. The decoder creates an identical prediction and adds this to the decoded residual. Inter prediction uses block-based *Motion Compensation*. First, the samples at half-pixel location (i.e. between integer-position samples) in the *Luminance* (Luma: Y) component of the reference frame are generated (Fig. 11: blue boxes for horizontal and green boxes for vertical) using a six tap *Finite Impulse Response* (FIR) filter with weights [1/32, −5/32, 20/32, 20/32, −5/32, 1/32]. For example, half-pixel sample 'b' (Fig. 11) is computed as $b = ((E - 5F + 20G + 20H - 5I + J + 16)/32)$. The samples at quarter-pixel positions are generated by *Bilinear Interpolation* using two horizontally and/or vertically adjacent half- or integer-position samples e.g. $Gb = (G + b + 1)/2$, where 'Gb' is the pixel at quarter-pixel position between 'G' and 'b'.

The MC_Hz_4 SI is used to compute the half-pixel interpolated values. It takes two 32-bit input values containing 8 pixels and applies a six-tap filter using *SHIFT* and *ADD* operations. In case of aligned memory access, *BytePack* aligns the data for the filtering operation. Then the *PointFilter* data path performs the actual six-tap filtering operation. Afterwards *Clip3* data path performs the rounding and shift operation followed by a clipping between 0 and 255. Figure 12 shows the three constituting data paths for MC_Hz_4 SI.

c. Intra prediction: horizontal DC (IPred_HDC) and vertical DC (IPred_VDC)

In case of high motion scenes, the *Motion Estimator* normally fails to provide a good match (i.e. MB with minimum block distortion) thus resulting in a high residue and for a given bit rate this deteriorates the encoded quality. In this case, *Intra Prediction* serves as an alternate by providing a better prediction i.e. reduced amount of residues. Our two SIs *Ipred_HDC* and *Ipred_VDC* implement three modes of *Luma 16×16* i.e. *Horizontal*, *Vertical*, and *DC*. *Horizontal Prediction* is given by $p[-1, y]$, with $x, y = 0 \ldots 15$ and *Vertical Prediction* is given by $p[x, -1]$, with $x, y = 0 \ldots 15$. *DC Prediction* is the average of top and left neighboring pixels and is computed as follows:

- If all left and top neighboring samples are available, then $\mathrm{DC} = \left(\sum_{x'=0}^{15} p[x', -1] + \sum_{y'=0}^{15} p[-1, y'] + 16 \right) >> 5$.
- Otherwise, if any of the top neighboring samples are marked as not available and all of the left neighboring samples are marked as available, then $\mathrm{DC} = \left(\sum_{y'=0}^{15} p[-1, y'] + 8 \right) >> 4$.
- Otherwise, if any of the left neighboring samples are not available and all of the top neighboring samples are marked as available, then $\mathrm{DC} = \left(\sum_{x'=0}^{15} p[x', -1] + 8 \right) >> 4$.
- Otherwise, $\mathrm{DC} = (1 \ll (\mathrm{BitDepth_Y} - 1)) = 128$, for 8-bit pixels.

IPred_HDC computes the *Horizontal Prediction* and the sum of left neighbors for *DC Prediction*. IPred_VDC computes the *Vertical Prediction* and the sum of top neighbors for *DC Prediction*. Figure 13 shows the *CollapseAdd* and *PackLBytes* data paths, which constitute both of the *Intra Prediction* SIs.

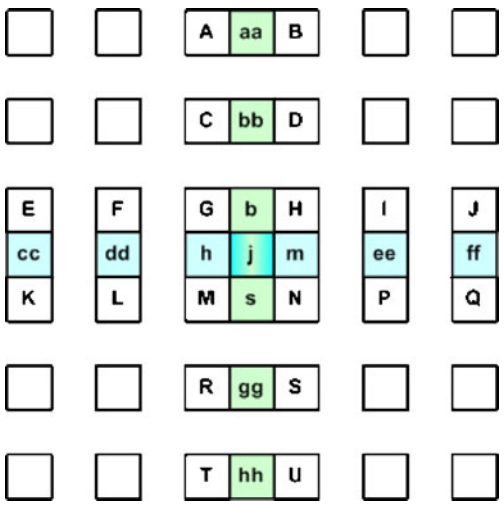

Figure 11 Interpolation of Luma half-pixel positions.

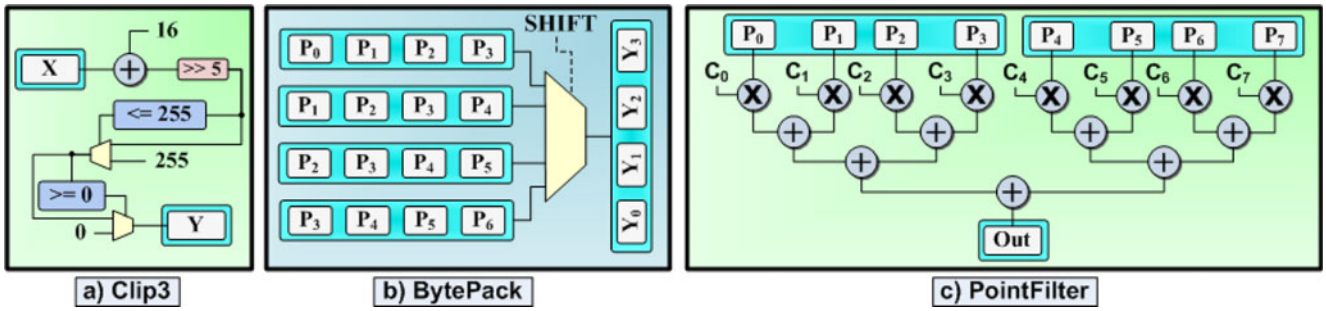

Figure 12 Data paths for MC_Hz_4 special instruction.

d. (Inverse) discrete cosine transform ((I)DCT)

H.264 uses three different transforms depending on the data to be coded. A 4×4 *Hadamard Transform* for the 16 *Luma DC Coefficients* in I-MBs predicted in 16×16 mode, a 2×2 *Hadamard Transform* for the 4 *Chroma DC Coefficients* (in both I- and P-MBs) and a 4×4 *Discrete Cosine Transform* that operates on 4×4 sub-blocks of residual data after *Motion Compensation* or *Intra Prediction*. The H.264 DCT is an integer transform (all operations can be carried out using integer arithmetic), therefore, it ensures zero mismatches between encoder and decoder. Equation 4 shows the core part of the 2-D DCT on a 4×4 sub-block X that can be implemented using only additions and shifts:

$$\mathrm{DCT}_{4\times4} = \mathrm{CXC}^T = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \quad [X] \quad \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right). \tag{4}$$

The inverse transform is given by Eq. 5 and it is orthogonal to the forward transform i.e. $T^{-1}(T(X))=X$:

$$\mathrm{IDCT}_{4\times4} = C_I Y' C_I^T = \left(\begin{bmatrix} 1 & 1 & 1 & 1/2 \\ 1 & 1/2 & -1 & -1 \\ 1 & -1/2 & -1 & 1 \\ 1 & -1 & 1 & -1/2 \end{bmatrix} \quad [Y'] \quad \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1/2 & -1/2 & -1 \\ 1 & -1 & -1 & 1 \\ 1/2 & -1 & 1 & -1/2 \end{bmatrix} \right) \tag{5}$$

where $Y'=Y\otimes E_t$ given E_t as the matrix of weighting factor. The *DCT* and *IDCT* SIs use *QSub*, *Transform*, and *Repack* (Fig. 10a–c) data paths to compute the 2-D *(Inverse) Discrete Cosine Transform* of 4×4 array.

e. (Inverse) Hadamard transform 4×4 ((I)HT_4×4)

If an MB is encoded as I-MB in 16×16 mode, each 4×4 residual block is first transformed using Eq. 4. Then, the

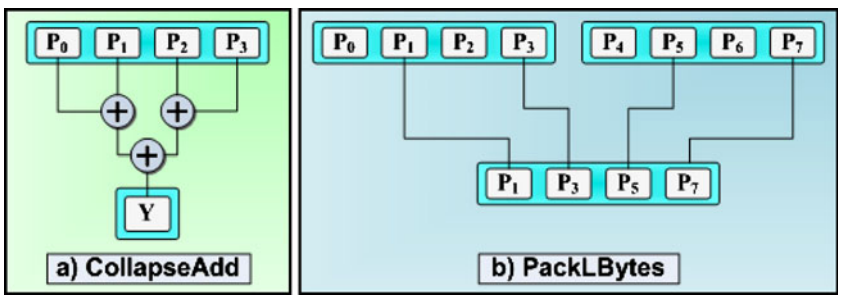

Figure 13 Data paths for IPred_HDC and IPred_VDC special instructions.

DC coefficients 4×4 blocks are transformed using a 4×4 *Hadamard Transform* (see Eq. 3). The inverse *Hadamard Transform* is identical to the forward transform (Eq. 3). The SIs for *HT_4×4* and *IHT_4×4* use *Transform* and *Repack* (Fig. 10b, c) data paths to compute the 2-D *(Inverse) Hadamard Transform* of 4×4 *Intra Luma DC* array.

f. (Inverse) Hadamard transform 2×2 ((I)HT_2×2)

Each 4×4 block in the Chroma components is transformed using Eq. 4. The DC coefficients of each 4×4 block of Chroma coefficients are grouped in a 2×2 block (W_{DC}) and are further transformed using a 2×2 *Hadamard Transform* as shown in Eq. 6. Note: the forward and inverse transforms are identical:

$$\mathrm{HT}_{2x2} = \left(\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} [W_{\mathrm{DC}}] \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \right). \quad (6)$$

The SIs for *HT_2×2* and *IHT_2×2* use one *Transform* (Fig. 10c) data path and computes the 2-D *(Inverse) Hadamard Transform* of 2×2 *Chroma DC* array.

g. Loop filter (LF_BS4)

The *De-blocking Filter* is applied after the reconstruction stage in the encoder to reduce blocking distortion by smoothing the block edges. The filtered image is used for motion-compensated prediction of future frames. The following section describes the loop filter *Special Instruction* and the constituting data paths in detail along with our proposed optimizations.

6.1 Fast Data Paths and Special Instruction for In-Loop De-blocking Filter

The H.264 codec has an in-loop adaptive *De-blocking Filter* for removing the blocking artifacts at 4×4 sub-block boundaries. Each boundary of a 4×4 sub-block is called one 4-pixel edge onwards as shown in Fig. 14. Each MB has 48 (32 for *Luma* and 16 for *Chroma*) 4-pixel edges. The standard specific details of the filtering operation can be found in [1]. Figure 15 shows the filtering conditions and filtering equations for *Boundary Strength*=4 (as specified in [1]) where P_i and Q_i (i=0, 1, 2, 3) are the pixel values across the block horizontal or vertical boundary as shown in Fig. 16.

We have designed a *Special Instruction* (SI) for in-loop *De-blocking Filter* (as shown in Fig. 17a) that targets the processing flow of Fig. 16. This SI filters one 4-pixel edge, which corresponds to the filtering of four rows each with 8 pixels. The *LF_BS4* SI constitutes two types of data paths: the first data path computes all the conditions (Fig. 17c) and the second data path performs the actual filtering operation (Fig. 18). The *LF_BS4* SI requires four data paths of each type to filter four rows of an edge. Threshold values α and β are packed with P (4-pixel group on left side of the edge; see Fig. 16) and Q (4-pixel group on right side of the edge) type pixels and passed as input to the control data path. The UV and BS act as control signals to determine the case of *Luma-Chroma* and *Boundary Strength* respectively. The condition data path outputs two 1-bit flags X_1 (for filtering P-type i.e. P_i pixels) and X_2 (for filtering Q-type i.e. Q_i pixels) that act as the control signals of the filter data path. The two sets of pixels (P and Q type) are passed as input to this data path and appropriate filtered pixels are chosen depending upon the two control signals.

Figure 17b shows the processing schedule of the *LF_BS4* SI. In first two cycles, two rows are loaded (P and Q of one row are loaded in one *LOAD* command). In cycle 3, two control data paths are executed in parallel followed by two parallel filter data paths in the cycle 4 to get the filtered pixels for first and second row of the edge. In the mean time, next two loads are executed. In cycle 5 and 6, the filtered pixels of first and second rows are stored while control and filter data paths are processed in parallel for third and fourth rows. In cycle 7 and 8, the filtered pixels of third and fourth rows are stored. Now we will discuss the two proposed data paths.

We have collapsed all the if–else equation in one condition data path that calculates two outputs to determine the final filtered values for the pixel edge. In hardware, all the conditions are processed in parallel and our hardware implementation is 130× faster than the software implementation (i.e. running on GPP).

Figure 18 shows our optimized data path to compute the filtered pixels for *Luma* and *Chroma* and selects the appropriate filtered values depending upon X_1 and X_2 flags. This new proposed data path needs fewer operations to filter the pixels on block boundaries as compared to the standard equations. The proposed data path exploits the redundancies in the operation sequence, re-arranges the operation pattern, and reuses the intermediate results as much as possible. Furthermore, this data path is not only good for hardware (ASIPs, *Reconfigurable Platforms*, RISPP) implementations, but also beneficial when implemented in software (GPP). In the standard equations, only one part is processed depending upon which condition is chosen at run time. For the software implementation of our proposed data path, the *else*-part can be detached in order to save the extra processing overhead. For the hardware implementation, we always process both paths in parallel. Note that the filtering data path is made more reusable

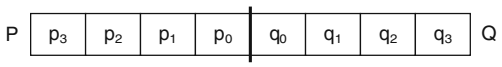

Figure 14 4-pixel edges in one macroblock.

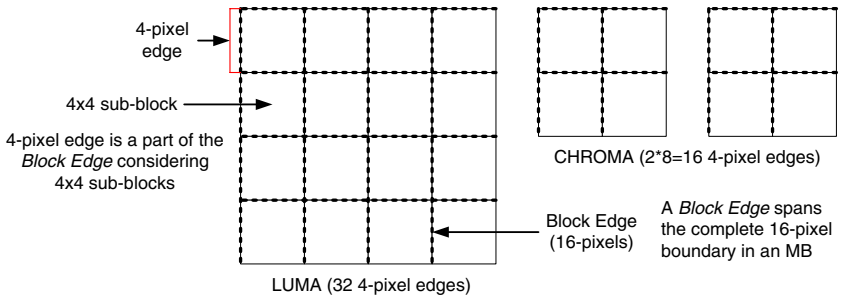

Figure 15 The filtering process for boundary strength=4.

using multiplexers. It is used to process two cases of *Luma* and one case of *Chroma* filtering depending upon the filtering conditions. The filtering of a four-pixel edge in software (i.e. running on GPP) takes 960 cycles for *Boundary Strength*=4 case. Our proposed *Special Instruction* (Fig. 17a) using these two optimized data paths (Figs. 17c and 18) requires only eight cycles (Fig. 17b) i.e. a speedup of 120×.

We have implemented the presented data paths for computing the conditions (Fig. 17c) and the filtering operations (Fig. 18) for a Virtex-II FPGA. The filtering operation was implemented in two different versions. The first one (*original*) was implemented as indicated by the pseudo-code in Fig. 16 and the second one (*optimized*) was implemented in our optimized data path (Fig. 18). The results are shown in Table 2. The optimized loop filter operation reduces the number of required slices to 67.8% (i.e. 1.47× reduction). At the same time, the critical path increases by 1.17× to 9.69 ns (103 MHz), which does not increase the critical path for our hardware prototype (see Section 7).

7 Properties of Hardware Platforms used for Benchmarking

We evaluate our proposed application structural optimizations with diverse hardware platforms in the following

```
1.   // Compute Filtering Conditions and Filtered Pixels for Boundary Strength=4
2.   IF (abs(q0–p0) < α) THEN
3.     IF (abs(q0-q1) < β) & (abs(p0-p1) < β) THEN
4.       IF (chromaEdgeFlag==0) THEN
5.       aq = abs(q0-q2) < β
6.       ap = abs(p0-p2) < β
7.       END IF
8.       IF (Boundary_Strength==4) THEN
9.       IF (chromaEdgeFlag==0)&(ap < β && Abs( p0 – q0 ) < ( ( α >> 2 ) + 2 )) THEN
10.        p'0 = ( p2 + 2*p1 + 2*p0 + 2*q0 + q1 + 4 ) >> 3
11.        p'1 = ( p2 + p1 + p0 + q0 + 2 ) >> 2
12.        p'2 = ( 2*p3 + 3*p2 + p1 + p0 + q0 + 4 ) >> 3
13.      ELSE
14.            p'0 = ( 2*p1 + p0 + q1 + 2 ) >> 2
15.            p'1 = p1
16.            p'2 = p2
17.      END IF
18.      IF (chromaEdgeFlag==0)&(aq < β && Abs( p0 – q0 ) < ( ( α >> 2 ) + 2 )) THEN
19.        q'0 = ( p1 + 2*p0 + 2*q0 + 2*q1 + q2 + 4 ) >> 3
20.        q'1 = ( p0 + q0 + q1 + q2 + 2 ) >> 2
21.        q'2 = ( 2*q3 + 3*q2 + q1 + q0 + p0 + 4 ) >> 3
22.      ELSE
23.            q'0 = ( 2*q1 + q0 + p1 + 2 ) >> 2
24.            q'1 = q1
25.            q'2 = q2
26.      END IF
27.      END IF
28.    END IF
29.  END IF
```

Figure 16 Pixel samples across a 4×4 block horizontal or vertical boundary.

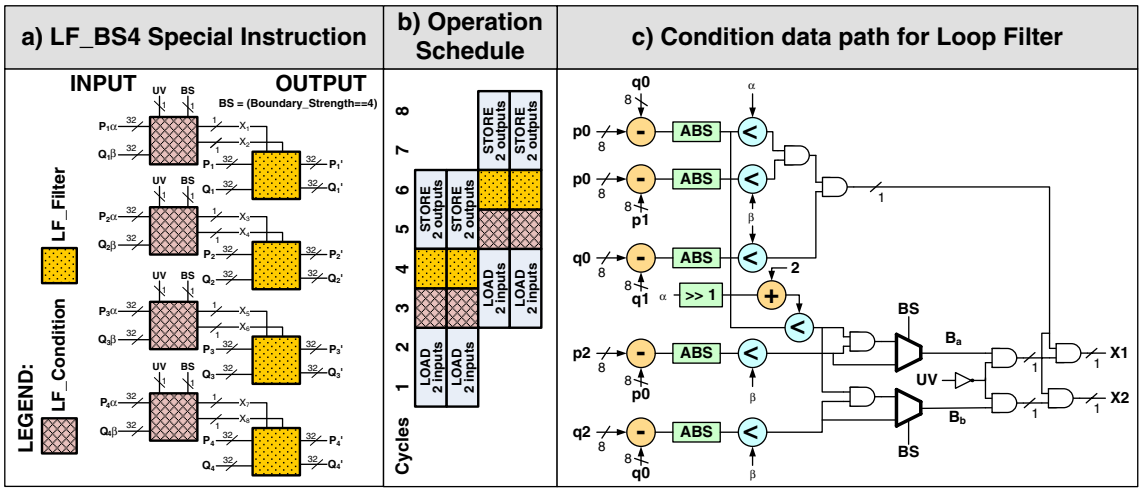

Figure 17 Special instruction with constituting optimized data path of filtering conditions for in-loop de-blocking filter and the processing schedule using condition and filtering operations.

result section, i.e. GPP (*General Purpose Processor*), ASIP (*Application-Specific Instruction set Processor*), *Reconfigurable Platform*, and RISPP (*Rotating Instruction Set Processing Platform*). Although the focus of this paper is on our proposed application structural optimizations, we will briefly describe the differences between ASIPs, *Reconfigurable Platforms*, and RISPP that are used as hardware platforms for benchmarking these optimizations.

The term ASIP comprises nowadays a far larger variety of embedded processors allowing for customization in various ways including (a) instruction set extensions, (b) parameterization and (c) inclusion/exclusion of predefined blocks tailored to specific applications (like, for example, an MPEG-4 decoder) [54]. A generic design flow of an embedded processor can be described as follows: (1) an application is analyzed/profiled, (2) an extensible instruction set is defined, (3) extensible instruction set is synthesized together with the core instruction set, (4) retargetable tools for compilation, instruction set simulation etc. are (often automatically) created and application characteristics are analyzed, (5) the process might be iterated several times until design constraints comply. However, for large applications that feature many computational hotspots and not just a few exposed ones, current ASIP concepts struggle. In fact, customization for many hotspots bloats the initial small processor core to consider-

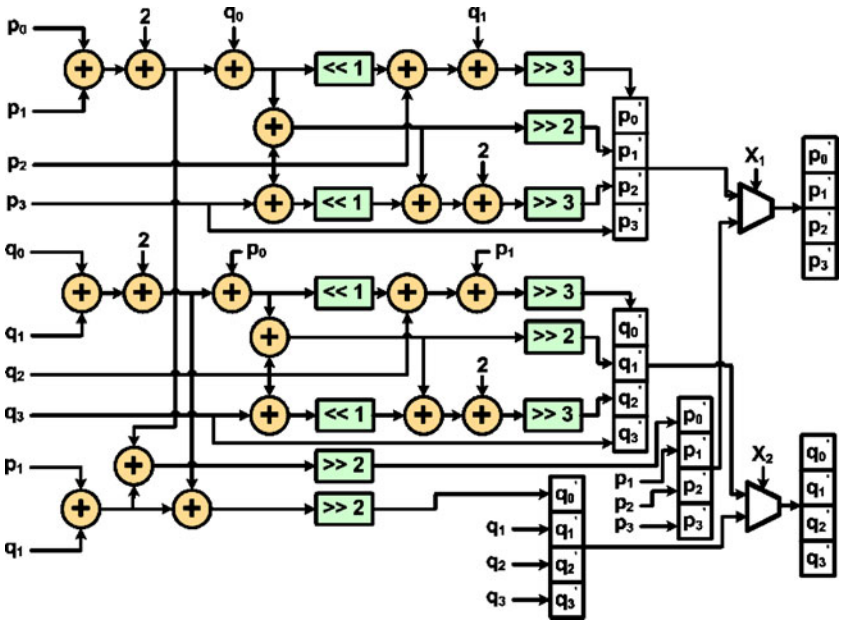

Figure 18 Optimized data path of boundary strength=4 filtering operation for in-loop de-blocking filter with parallel processing of Luma and Chroma paths.

Table 2 Results for hardware implementation of individual data paths.

Characteristics	Data paths		
	Condition	Loop filter (original)	Loop filter (optimized)
No. of slices	72	174	118
No. of LUTs	138	306	204
Gate equivalents	1,207	2,754	2,074
Latency [ns]	6.49	8.25	9.69

ably larger sizes (factors of the original base processor core). Moreover, to develop an ASIP, a design-space exploration is performed and the decisions (which *Special Instructions* (SIs), which level of parallelism, etc.) are fixed at design time.

Reconfigurable processors instead combine e.g. a pipeline with reconfigurable parts. For instance, fine-grained reconfigurable hardware (based on lookup tables i.e. similar to FPGAs) or coarse-grained reconfigurable hardware (e.g. arrays of ALUs) can be connected to a fixed processor as functional unit or as co-processor [53]. This reconfigurable hardware can change its functionality at run time, i.e. while the application is executing. Therefore, the SIs no longer need to be fixed at design time (like for ASIPs), but only the amount of reconfigurable hardware is fixed. The SIs are fixed at compile time and then reconfigured at run time. Fixing the SIs at compile time has to consider constraints like their maximal size (to fit into the reconfigurable hardware) and typically includes the decisions *when* to reconfigure *which* part of the reconfigurable hardware. Determining the decision *when* to start reconfiguring the hardware has to consider the rather long reconfiguration time (in the range of milliseconds, depending on the size of the SI).

Compared to typical ASIPs or *Reconfigurable Platforms* RISPP is based on offering SIs in a modular manner. The modular SIs are connected to the core pipeline in the execute stage, i.e. the parameters are prepared in the decode stage and are then passed to the specific SI implementation in the execute stage. The interface to the core pipeline is identical for ASIPs, *Reconfigurable Platform*s, and RISPP. The main difference is the way in which the SIs are implemented. Instead of implementing full SIs independently, data paths are implemented as elementary re-usable reconfigurable units and then combined to build an SI implementation. The resulting hierarchy of modular SIs (i.e. SIs, SI implementations, and data paths) is shown in Fig. 17. Figure 17a shows the composition of the *Loop Filter* SI, while Fig. 17c shows one of its elementary data paths. The schedule in Fig. 17b corresponds to one certain implementation of this SI, for the case that two instances of both required data paths are available and can be used in parallel. However, the same SI can also be implemented with only one instance of each data path by executing the data paths sequentially (note that parallelism is still exploited within the implementation of a data path). On the one hand, this saves area but on the other hand, it costs performance. Additionally, the *Loop Filter* SI can be implemented without any data paths (executing it like a GPP) and it can be implemented if e.g. only the filtering data paths (Fig. 18) is available in hardware (then the conditions are evaluated like on a GPP and then forwarded to the filtering data paths). These different implementation trade-offs are available for all extensible processor platforms, i.e. ASIP, *Reconfigurable Platform*, and RISPP. ASIPs select a certain SI implementation at design time and *Reconfigurable Platforms* select an implementation at compile time. RISPP instead selects an implementation at run time out of multiple compile-time prepared alternatives [49]. While performing this selection, RISPP considers the current situation, e.g. the expected SI execution frequency (determined by an online monitoring [50]) to specifically fulfill the dynamically changing requirements of the application.

To efficiently support different implementation alternatives of one SI, RISPP offers the data paths as elementary reconfigurable units. This means that the configuration of a single data path is loaded into one out of multiple reconfigurable regions. As soon as sufficient data paths for an SI implementation are available, the data paths are connected (via busses, using a compile-time prepared connection scheme) to implement this SI. As soon as sufficient further data paths finished reconfiguration, these connections are changed to use the further data paths and thus to realize a faster SI implementation that exploits more parallelism. Reconfiguring at the level of data paths not only allows the concept of *upgrading* SI implementations at run time but it furthermore allows *sharing*, i.e. one reconfigured data path can be used for the implementations of different SIs. A run-time system manages the currently available SI implementation and controls the required connections between the core pipeline and the data paths. This run-time system also controls in which sequence the SIs shall be *upgraded*. The feature to gradually *upgrade* SI implementations diminishes the above-mentioned problem of rather long reconfiguration times, but it does not diminish the conceptual problem of high *hardware pressure* like presented in Section 5. Therefore, either more reconfigurable hardware has to be added or the *hardware pressure* itself has to be distributed, e.g. by our optimized application structure.

For evaluation purpose, we have implemented a RISPP prototype on an FPGA board (see Fig. 19). We have implemented the core pipeline and the reconfigurable

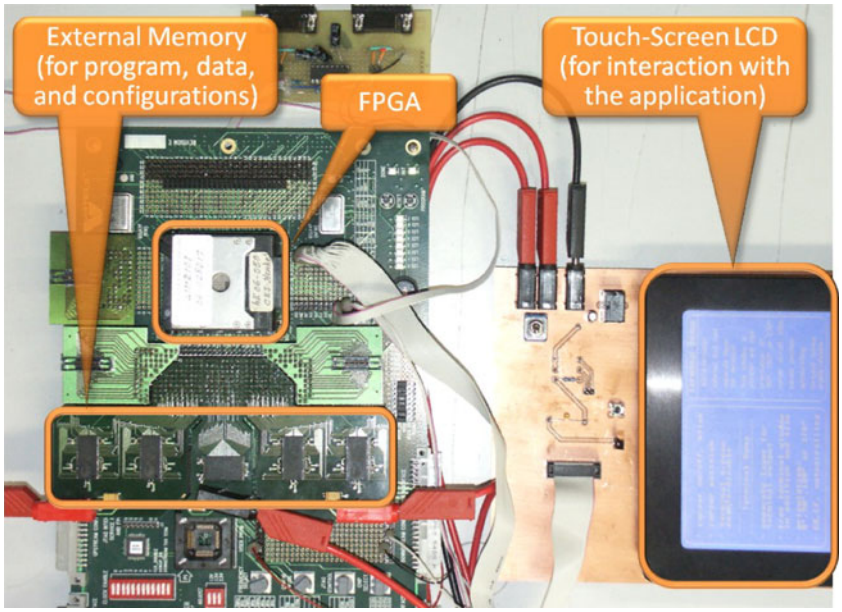

Figure 19 FPGA-based RISPP prototyping platform.

regions on the FPGA and tightly coupled the reconfigurable parts (as functional units) with the execution stage of the pipeline. Our core pipeline is based on a MIPS processor that was extended for our requirements (e.g. four read ports in the register file) and runs at 50 MHz. The processor is connected to external SRAM for instruction and data memory of the application as well as to a fast external EEPROM for the configuration data. The realized peripheral connections comprise a touch-screen LCD that allows interaction with the running application (not used while benchmarking).

8 Evaluation, Discussion and Results

8.1 Test Conditions and Fairness of Comparison

In the motivation (Section 1), we have introduced and analyzed the problem of excessive *Interpolation* and *hardware pressure*. To solve this issue, we have proposed a set of application structural optimizations. At first, we have created an optimized *Benchmark Application* of H.264 video encoder [2] (Section 3) considering multimedia applications with *Common Intermediate Format* (CIF=352×288) or *Quarter CIF* (QCIF=176×144) resolutions running on mobile devices. The same *Benchmark Application* is used for all hardware platforms, i.e. GPP (General Purpose Processor; in our case a MIPS), ASIP (*Application-Specific Instruction set Processor*), *Reconfigurable Platform*, and RISPP). Detailed results are presented for the *Carphone* (a typical videophone sequence) under the test conditions *Quantization Parameter* (QP)=30, *Group of Pictures* (GOP)=IPPP…, and *Search Range*=16. Before proceeding to the discussion of results, we will present the fairness of comparison in this section.

This paper focuses on application structural optimizations for H.264 video encoder. An application structure running on a particular hardware platform is compared with different application structures on the same hardware platform. Therefore, the performance in terms of cycles is considered, as the test conditions with respect to the underlying hardware platforms are always same for different application structures. In order to study the effect of the proposed optimizations and to validate our concept, we have carried out this comparison on several hardware platforms keeping in mind that the different application structures are only compared within one hardware platform at a time. We will now enumerate the similarities and test conditions that justify the fairness of comparison.

1. All the *Application Structures* use the same *Benchmark Application* as the starting point. Same set of optimized data paths and *Special Instruction* (SIs) is given to all application structures. Each data path takes one cycle for execution.
2. All the application structures were given the same hardware resources, i.e. register file, number of read/write ports, memory accesses, etc. when executing on a particular hardware platform.
3. Only the application structures running on the same hardware platforms are compared with each other i.e. *Benchmark Application* executing on ASIP is compared with *Interpolation-Optimized Application* and *Hardware-Pressure-Optimized Application* running on ASIP.

8.2 Results and Discussion

Our proposed optimized filtering operation for the in-loop *De-blocking Filter* (Fig. 18) needs (18+5)×48×#MBs additions (CIF=399,168; QCIF=99,792 additions) while the standard equations need (30+6)×48×#MBs additions (CIF=684,288; QCIF=171,072 additions) for both *Luma* and *Chroma*. It shows 41.67% operation reduction. The standard equations suffer performance degradation due to six conditional jumps but our optimized implementation has only two conditions. For a hardware implementation, this results in simple multiplexers. The optimized filtering operation reduces the number of required slices to 67.8% (i.e. 1.47× reduction, see Table 2). At the same time the critical path increases (by 1.17×) to 9.69 ns (103 MHz), which does not increase the critical path for our hardware prototype that is running at 50 MHz (20 ns). In addition to the filtering operation, the hardware implementation of our condition data path (Fig. 17c) is 130× faster than the corresponding software implementation (i.e. running on GPP) as we process all conditions in parallel. The proposed condition data path requires only 72 slices as the logic after comparing the absolute differences is done on bit-level (see Fig. 17c). Combining two instances of our both optimized data paths to implement the proposed *Special Instruction* (Fig. 17a) takes only eight cycles (see Fig. 17b) to filter a 4-pixel edge for the *Boundary Strength*=4 case. Performing the same computation on the GPP requires 960 cycles, which corresponds to a 120× faster hardware execution. However, this speedup only considers the execution of our proposed In-loop *De-Blocking Filter*, not the whole H.264 encoder. Therefore, we subsequently concentrate on the execution time of specific hot spots and the whole application (showing the benefits of our proposed optimizations of the application structure). Eventually, we will present the detailed analysis of the SI execution pattern within one frame to clarify and evaluate the conceptual differences of our both proposed application structures.

Figure 20 shows a comparison between the *Benchmark Application* and our on-demand interpolation processing (Section 4) running on GPP for four typical QCIF video sequences that cover most of the motion characteristics of videos for mobile devices. *Carphone* represents the video call scenario in a moving car where the motion is due to the body of the caller and the background from the side-window. *Miss America* is a typical head–shoulder movement sequence and exhibits relatively low motion characteristics. *Trevor* is a relatively complex scene where several persons

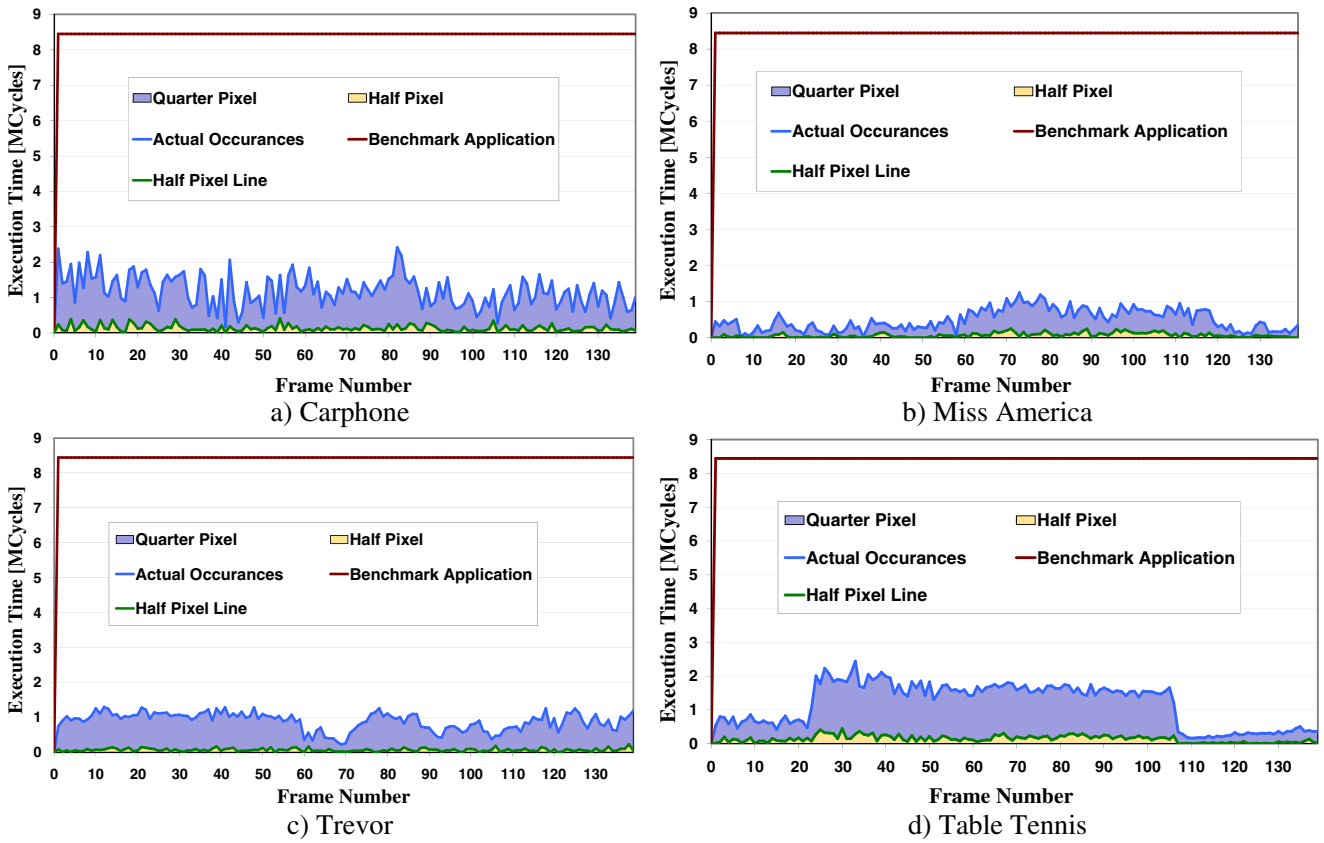

Figure 20 Processing time of interpolation for the benchmark vs. our proposed application structure when running on GPP (MIPS).

are moving in one video sequence simulating several movements occurring in video conferencing. *Table Tennis* represents camera pan motion and the motion of game objects (persons and ball). In case of camera panning the background is not stationary, rather all the MBs of background move with a same amount, i.e. motion vectors are approximately equal for MBs of background.

The processing time of the interpolation in the *Benchmark Application* is always 8.444 MCycles as it processes all cases. Using our optimized application structure from Section 4, the interpolations for 1 *QCIF* frame requires at maximum 2.417 (average 1.191), 1.244 (average 0.491), 1.298 (average 0.893), and 2.441 (average 1.177) MCycles for *Carphone*, *Miss America*, *Trevor*, and *Table Tennis* respectively. Analyzing the behavior of these four typical video sequences we find that the maximum and average processing time for interpolation is much smaller than that of the *Benchmark Application* (due to the stationary background). Therefore, the application structural optimization for on-demand interpolation gives a huge saving in terms of processing time (compared to the *Benchmark Application*) as we will now evaluate with further video sequences (Table 3).

We have considered seven different QCIF video sequences [51] with different motion properties to simulate various mobile video scenarios. In the *Benchmark Application*, the interpolation of 139 frames takes 1,174 MCycles (139 frames×8.444 MCycles; same for all video sequences). Table 3 shows the processing time and the speedup (compared to the *Benchmark Application* architecture) of the interpolation process in our optimized application structure with on-demand interpolation on GPP. Our presented approach to reduce the number of performed interpolations is independent of any specific processor platform as it conceptually reduces the amount of required computations, independent whether this computation is done in software (GPP) or in hardware (e.g. ASIP). It comes at the cost of increased *hardware pressure*.

While the above presented optimizations (i.e. optimized filtering operation and reduced interpolations) are beneficial for a GPP as well, we will now focus on hardware implementations (especially on *Reconfigurable Platforms*). At design time, a particular amount of hardware is given to ASIPs, *Reconfigurable Platforms*, and RISPP. The amount of hardware determines how many *Special Instructions* (SIs) can be expedited to exploit the inherent parallelism thus directly corresponding to the achieved performance. In case of ASIPs, the data paths that can be accommodated in this hardware are fixed, while for *Reconfigurable Platforms* and RISPP the data path are loaded at run time and only the amount of data path that can be present on the reconfigurable hardware at the same time is fixed. Figure 21 shows the execution times of the *Benchmark Application* (onwards called *BApp*), our *Interpolation-Optimized Architecture* (as explained in Section 4; onwards called *InOpt*), and our *Hardware-Pressure-Optimized Architecture* (as explained in Section 5, onwards called *HPOpt*) running on a *Reconfigurable Platform*. The *x*-axis in this plot shows to the amount of data paths that fit to the hardware at the same time. The more reconfigurable hardware is available the faster the execution becomes. The application execution without data paths i.e. always using the instruction set of the core-pipeline/base processor corresponds to a GPP. As the performance without data paths is much slower, we do not show it in the Fig. 21.

We will now shed light on the region between two important points marked as *Label A* and *Label B* in Fig. 21 where (contrary to the other regions) *InOpt* is not significantly faster than *BApp*. For one and two data paths, the *Motion Compensation Special Instruction* (i.e. *MC_Hz4*) executes in software for both *BApp* and *InOpt*. In this case, *InOpt* achieves its performance improvement due to the reduced number of interpolations. *HPOpt* is further better than *InOpt* due to the reduced *hardware pressure*, as we will see. At *Label A*, *BApp* has almost the same processing time as of *InOpt* because in case of *BApp*, the *MC_Hz4* SI (along with e.g. SAD and SATD) now executes in hardware. However, for *InOpt* the *MC_Hz4* SI still executes in software. This is due to the high *hardware*

Table 3 Processing time and speedup of interpolation (half- and quarter-pixel) for 139 frames compared to the benchmark application (1,173.77 MCycles) for GPP.

Video sequence	Half-pixel (MCycles)	Quarter-pixel (MCycles)	Total (MCycles)	Speedup
Bridge_Close	0.02	3.46	3.48	333.33
Carphone	18.23	147.37	165.60	7.09
Claire	3.08	26.50	29.58	39.68
Grandma	3.46	31.93	35.39	33.22
Miss America	8.98	59.33	68.30	17.18
Salesman	1.72	34.60	36.32	32.36
Table_Tennis	18.98	144.69	163.67	7.17
Trevor	8.89	115.27	124.16	9.45
Average	7.92	70.39	78.31	59.96

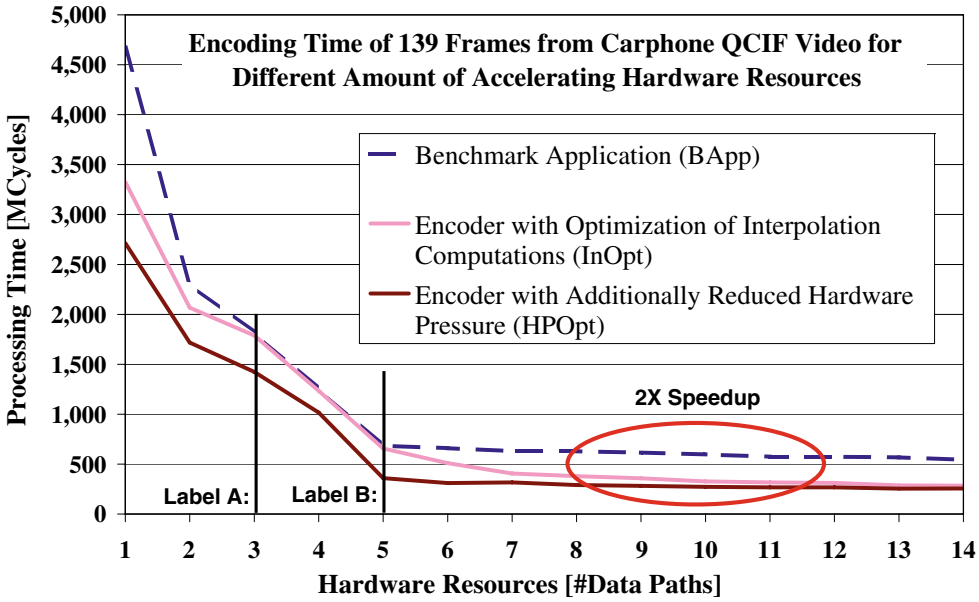

Figure 21 Comparison of the benchmark application and our proposed optimized application structures for reconfigurable platforms.

pressure that is introduced by shifting the interpolations inside the MB *Encoding Loop* (necessary to reduce the number of performed interpolations, see Section 4). The available reconfigurable hardware has to be shared among the SIs of the MB *Encoding Loop*, as reconfiguring within the loop is not beneficial (due to the long reconfiguration time as discussed in Section 7). As SAD and SATD are significantly more often executed than the *MC_Hz4* SI, it is more beneficial (concerning overall performance) to implement SAD and SATD in hardware while executing *MC_Hz4* in software. However, *BApp* can implement SAD and SATD as well as *MC_Hz4* in hardware, as it executes them in different loops and reconfiguration between the loops is feasible. Due to this reason, the gain of reduced interpolations is cancelled out by the difference of hardware and software executions of *MC_Hz4*. This situation exists between three (*Label A*) and five (*Label B*) available data paths. Afterwards, the main requirements (i.e. the execution of the most important SIs in hardware), of the MB *Encoding Loop* are fulfilled and the *MC_Hz4* SI is executed in hardware for *InOpt* too.

HPOpt resolves the discussed problem of *InOpt* by decoupling the SAD and SATD SIs from the MB *Encoding Loop*. The benefit of the reduced *hardware pressure* becomes especially noticeable in the region between *Label A* and *Label B*. When more reconfigurable hardware is available *InOpt* comes closer to *HPOpt* as the hardware needs of each SI are fulfilled. Due to high reconfiguration overhead, after eight data paths the *Reconfigurable Platform* does not give significant benefits per additional data path. However, the specific amount of data paths (where further performance benefits are no longer significant) highly depends on the application as well as the specific reconfiguration bandwidth.

Table 4 shows the maximum and average speedups (using one to 14 data paths; GPP does not use application-specific data paths therefore average=maximum) when executing the H.264 video encoder with the *BApp*, *InOpt*, and *HPOpt* application structures for all four hardware platforms. It is worthy to note that all these application structures use the same *Special Instructions* therefore the table purely represents the improvement of application structural optimizations. The table shows that *HPOpt* is up to 2.84× (average 2.36) and 1.94× (average 1.58×) faster than *BApp* for ASIP and *Reconfigurable Platform* respectively. For RISPP, *HPOpt* is at maximum 2.24× (average 1.91×) and 1.82× (average 1.29×) faster than *BApp* and *InOpt* respectively. Note, higher speedup for ASIP does not necessarily mean that ASIP is faster than *Reconfigurable Platform* or RISPP. They have different '1×' performance (i.e. execution time of *BAPP* on ASIP is different from that on *Reconfigurable Platform* or RISPP) and we do not intend to compare them. Therefore, the speedup in Table 4 only provides the comparison of different application structures on a particular hardware platform.

We now analyze the detailed differences in the execution patterns of *InOpt* (Fig. 5) vs. *HPOpt* (Fig. 6), on RISPP with four, five, and six data paths when encoding one *QCIF* video frame of *Carphone* sequence to explain how we get the performance benefit by reducing the *hardware pressure*. We show the number of executions of important SIs (*y*-axis) for timeframes of 100 KCycles time-slots (*x*-axis). For clarity, we restrict to those SIs that show the difference of the application structures. *InOpt* runs the SIs for ME and

Table 4 Maximum and average speedups of optimized vs. benchmark application architectures for different amount (i.e. one to 14) of accelerating hardware resources.

	InOpt vs. BApp		HPOpt vs. BApp		HPOpt vs. InOpt	
	Maximum speedup	Average speedup	Maximum speedup	Average speedup	Maximum speedup	Average speedup
GPP	1.38		1.46		1.06	
ASIP	2.34	2.01	2.84	2.36	1.30	1.17
Reconf.	2.05	1.54	1.94	1.58	1.67	1.09
RISPP	1.98	1.51	2.24	1.91	1.82	1.29

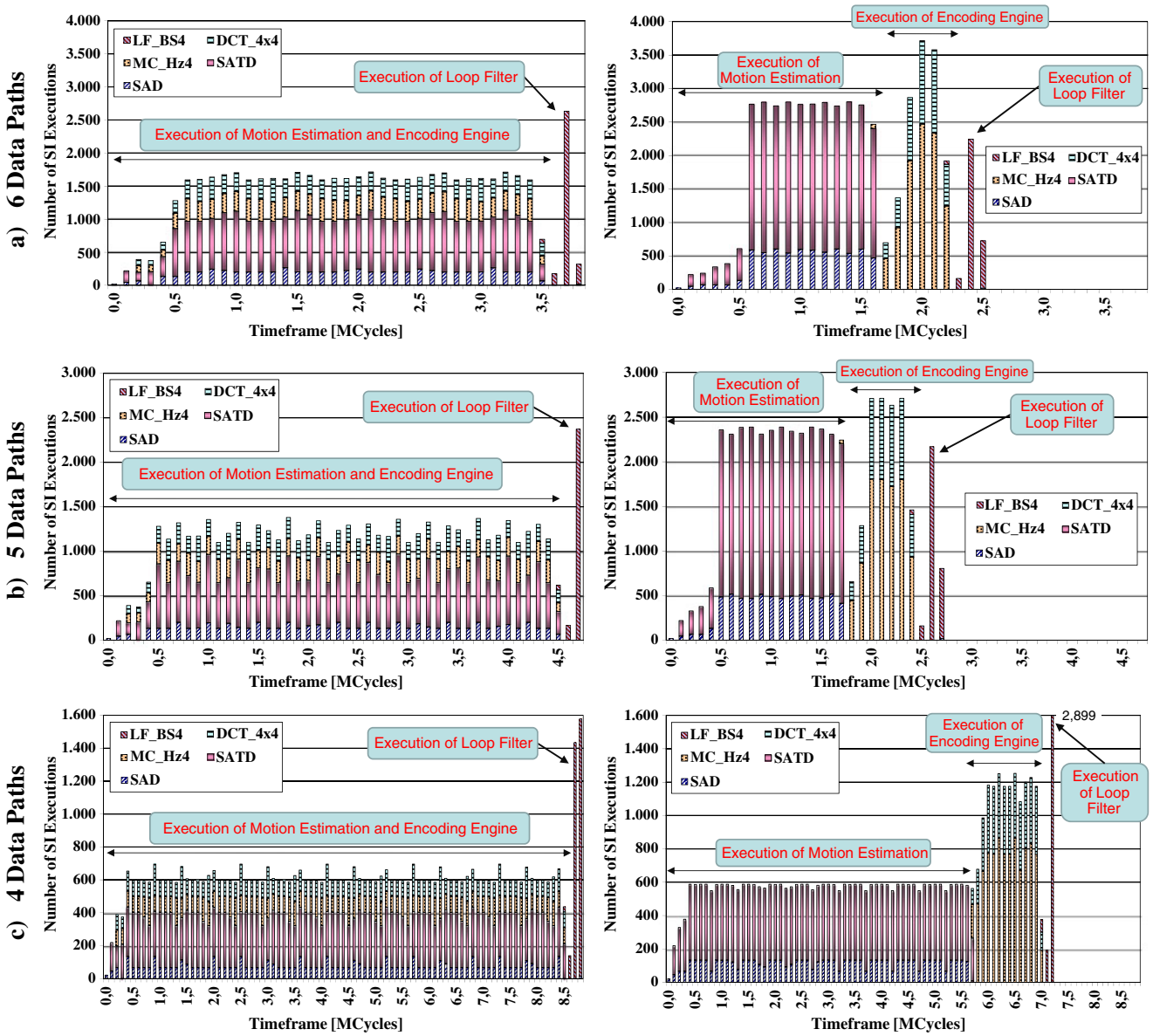

Figure 22 Execution time profile of major functions (five SIs) in our proposed application structures *InOpt* and *HPOpt* for 1 *QCIF* video frame. Detailed utilization variations for four, five, and six available data paths, used by *InOpt* and *HPOpt* respectively for RISPP. All figures show the encoding of 1 video image. The *x*-axis shows the timeframes of 100 KCycles, the *y*-axis denotes the number of SI executions. For a given number of data paths, the *x*-axis for *InOpt* and *HPOpt* was selected as the maximum number of timeframes that one of the application structures required to encode one video image.

MB *Encoding Loop* (*DCT*, SATD, …) together in one big loop over all MBs in a frame while *HPOpt* processes SIs of ME and MB *Encoding Loop* in two separate loops.

Figure 22a shows the detailed SI execution pattern for six available (reconfigurable) data paths using *InOpt* and *HPOpt* for RISPP. As ME (*Motion Estimation*) and MC (*Motion Compensation*) are processed in the same loop for *InOpt*, the first five data paths are allocated to SAD and SATD to expedite ME which is the most compute-intensive part of the video encoder. For *InOpt*, in the first 100 KCycles SAD is executed (running in software). After 100 KCycles (approximately one reconfiguration time, depending on which data path is loaded [49]), the first accelerating data path SAD_16 is loaded that will expedite the *SAD Special Instruction* (SI). Additionally, SATD starts execution, first in software and between 100 and 500 KCycles the corresponding data paths are loaded sequentially, therefore the SATD execution *upgrades* gradually (as explained in Section 7). The amount of SATD, *DCT*, and *MC* executions in the first 500 KCycle shows that not all accelerating data paths are loaded yet and therefore RISPP is not yet executing SIs at full performance. As more data paths finish loading, the single SI executions become faster and thus more SIs are executed per timeframe. Table 5 shows the six selected data paths in this case for the MB *Encoding Loop*. It is worthy to note that five data paths are given to the ME SIs while the MC SI gets only one data path. The optimization target thereby is to reduce the execution time of the complete application and not only of one functional block. Additionally, the *(I)DCT*, *(I)HT_2×2*, and *(I)HT_4×4* SIs are expedited due to reusable data paths (i.e. *Repack*, *Transform*, and *QSub* are shared between SATD and these SIs). Due to high *hardware pressure* not all SIs can be completely supported in hardware therefore the MB *Encoding Loop* takes longer to finish its execution, i.e. 3.6 MCycles. Afterwards, the data paths for in-loop *De-blocking Filter* are loaded. *InOpt* requires 3.9 MCycles to process the encoding of one *QCIF* video frame.

In contradiction to the presented *InOpt* processing flow, *HPOpt* first processes the ME SIs and therefore all the data paths are allocated to SAD (two data paths) and SATD (four data paths). It is noticeable in Fig. 22a that after all six data paths (2×SAD_16, *Repack*, *Transform*, *QSub*, SAV) are loaded, the total number of SI executions in the ME hot spot increases significantly as SAD and SATD are now executed in a faster hardware implementation (exploiting more parallelism). These faster hardware implementations were not selected for *InOpt*, as then no data path would have been left to accelerate MC. Therefore, MC would have been the bottleneck, leading to a reduced performance of this loop. After *HPOpt* finished the ME hot spot, the three data paths 2×SAD_16 and SAV are replaced by *PointFilter*, *Clip3*, and *BytePack* to expedite the *MC_Hz4* SI. The loop decoupling in *HPOpt* allows offering more data paths per SI as less SIs are needed per timeframe (SAD now additionally gets another data path of SAD_16 while *MC_Hz4* additional gets *Clip3* and *BytePack*). Therefore, all SIs can be offered in a better performance (exploiting more parallelism) which results in more executions per timeframe. Additionally, the number of *DCT* and *MC_Hz4* executions is reduced in *HPOpt* due to the early decision between I- and P-MB types by *Rate Distortion* (RD). In the result, *HPOpt* requires 2.6 MCycles to process the encoding of one *QCIF* video frame and is finished much earlier as compared to *InOpt*, showing the saving of 1.3 MCycles.

Compared to the presented details for six data paths, Fig. 22b and c present the results for five and four data paths respectively. It is noticeable in Fig. 22c that for four data paths *InOpt* requires 8.9 MCycles to process the encoding of one *QCIF* video frame which is significantly slower when compared with the performance of six data paths. This big difference comes due to the MB *Encoding*

Table 5 Data paths supported in hardware for InOpt and HPOpt for four, five, and six data paths.

No. of data paths	Data paths selected for acceleration				
	InOpt (data paths used in each hot spot)		HPOpt (data paths used in each hot spot)		
	MB encoding loop	In-loop de-blocking filter	Motion estimation	MB encoding loop	In-loop de-blocking filter
4	SAD_16, Repack, Transform, QSub	Cond, LF_4	SAD_16, Repack, Transform, QSub	Repack, Transform, QSub, PointFilter, Clip3	Cond, LF_4
5	SAD_16, Repack, Transform, QSub, SAV	Cond, LF_4	SAD_16, Repack, Transform, QSub, SAV	Repack, Transform, QSub, SAV, PointFilter, Clip3	Cond, LF_4
6	SAD_16, Repack, Transform, QSub, SAV, PointFilter	Cond, LF_4	2×SAD_16, Repack, Transform, QSub, SAV	Repack, Transform, QSub, PointFilter, Clip3, BytePack	Cond, LF_4

Loop that requires 8.6 MCycles to complete. Due to only four available data paths, *MC_Hz4* as well as SATD cannot be supported in a fast hardware implementation (e.g. SAV is missing for SATD, see Tables 1 and 5). *HPOpt* improves the performance (saving 1.6 MCycles compared to *InOpt*) but due to the absence of SAV the ME loop is still slow (requires 5.7 MCycles). However, for five data paths (Fig. 22b) SAV is available (see Table 5), therefore SATD executes in hardware but *MC_Hz4* is still in software. As SATD has a bigger contribution in the performance gains and we target reducing the overall execution time of the whole application, the data path for SATD is given preference on that of *MC_Hz4*. As a result the execution of *InOpt* for five data paths is faster than that of five data paths (Fig. 22b) but still slower than that of six data paths. *HPOpt* here gets a bigger benefit as now it complete the ME hot spot much earlier (1.8 MCycles) giving an overall saving of 2 MCycles for one frame encoding. Summarizing, because of faster SI implementations (more data paths per SI) and reduced computations, *HPOpt* is finished much earlier as compared to *InOpt*, showing the maximum speedup of 1.82× for RISPP (compared to the *Benchmark Application* running on RISPP), as shown in Table 4.

9 Conclusion

We have presented optimizations for the H.264 encoder application structure for reduced processing and reduced *hardware pressure* along with several novel data paths and the resulting *Special Instruction* for the main computational hot spots of the H.264 encoder. For in-loop *De-blocking Filter*, the optimized filtering data path reduces the number of required slices to 67.8% (i.e. 1.47× reduction, see Table 2). The *Special Instruction* is 120× faster than the *General Purpose Processor* Implementation. Our approach of reduced processing is good for all four hardware platforms (i.e. GPP, ASIP, *Reconfigurable Platform*, and RISPP), while the optimization of reduced *hardware pressure* target the *Reconfigurable Platform* and RISPP specifically. As compared to the optimized *Benchmark Application*, we achieve an average speedup of approximately 60× for Motion Compensated Interpolation for MIPS. For RISPP, our optimized application structure for interpolation achieves up to 1.98× improvement (see Table 4) over the *Benchmark Application* architecture. The reduced *hardware pressure* improves the performance of our optimized application architecture up to 2.24× compared to the *Benchmark Application*.

References

1. ITU-T Rec. H.264 and ISO/IEC 14496-10:2005 (E) (MPEG-4 AVC) “Advanced video coding for generic audiovisual services”, 2005.
2. ITU-T H.264 reference software version JM 13.2. Retrieved from http://iphome.hhi.de/suehring/tml/index.htm.
3. X264—a free H.264/AVC encoder. Retrieved from http://www.videolan.org/developers/x264.html.
4. Chen, Z., Zhou, P., & He, Y. (2002). Fast integer pel and fractional pel motion estimation for JVT, JVT-F017, 6th JVT Meeting, Awaji, December.
5. Raja, G., & Mirza, M. J. (2004). Performance comparison of advanced video coding H.264 standard with baseline H.263 and H.263+ standards. *IEEE International Symposium on Communications and Information Technology (ISCIT)*, *2*, 743–746.
6. Wiegand, T., Sullivan, G. J., Bjntegaard, G., & Luthra, A. (2003). Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, *13* (7), 560–576. doi:10.1109/TCSVT.2003.815165 (CSVT).
7. Ostermann, J., et al. (2004). Video coding with H.264/AVC: tools, performance, and complexity. *IEEE Circuits and Systems Magzine*, *4*(1), 7–28. doi:10.1109/MCAS.2004.1286980.
8. Wiegand, T., et al. (2003). Rate-constrained coder control and comparison of video coding standards. *IEEE Transactions on Circuits and Systems for Video Technology*, *13*(7), 688–703. doi:10.1109/TCSVT.2003.815168 (CSVT).
9. Bjontegaard, G. (2001). Calculation of average PSNR differences between RD-curves. ITU-T SG16 Doc. VCEG-M33.
10. Ziauddin, S. M., ul-Haq, I., Nadeem, M., & Shafique, M. Methods and systems for providing low cost robust operational control for video encoders, Pub. Date: Sept. 6, 2007; Patent Pub. No. US-2007-0206674-A1, Class: 375240050 (USPTO).
11. Yuan, W., Lin, S., Zhang, Y., Yuan, W., & Luo, H. (2006). Optimum bit allocation and rate control for H. 264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, *16*(6), 705–715. doi:10.1109/TCSVT.2006.875215 (CSVT).
12. Milani, S., et al. (2003). A rate control algorithm for the H.264 encoder. Baiona Workshop on Signal Processing in Communications.
13. Xtensa, L.X.: 2 processor, Tensilica Inc. Retrieved from http://www.tensilica.com.
14. Xtensa, L.X.: 2 I/O Bandwidth. Retrieved from http://www.tensilica.com/products/io_bandwidth.htm.
15. CoWare Inc: LISATek. Retrieved from http://www.coware.com/.
16. Arctangent processor. Retrieved from http://www.arc.com/configurablecores/.
17. Chen, T. C., Lian, C. J., & Chen, L. G. (2006). Hardware architecture design of an H.264/AVC video codec, Asia and South Pacific Conference on Design Automation (ASP-DAC), pp. 750–757.
18. Reconfigurable Instruction Cell Array, U.K. Patent Application Number 0508589.9.
19. Major, A., Yi, Y., Nousias, I., Milward, M., Khawam, S., & Arslan, T. (2006). H.264 Decoder implementation on a dynamically reconfigurable instruction cell based architecture. IEEE International SOC Conference, pp. 49–52.
20. Lee, W. H., & Kim, J. H. (2006). “H.264 Implementation with Embedded Reconfigurable Architecture”, IEEE International Conference on Computer and Information Technology (CIT), pp. 247–251.

21. The XPP team. (2002). The XPP White Paper, PACT Corporation, Release 2.1, pp. 1–4.
22. May, F. (2004). "PACT XPP virtual platform based on AXYS maxSim 5.0", PACT Corporation, Revision 0.3, pp. 12.
23. Berekovic, M., Kanstein, A., Desmet, D., Bartic, A., Mei, B., & Mignolet, J. (2005). Mapping of video compression algorithms on the ADRES coarse-grain reconfigurable array. Workshop on Multimedia and Stream Processors, Barcelona, November 12.
24. Veredas, F. J., Scheppler, M., Moffat, W., & Mei, B. (2005). Custom implementation of the coarse-grained reconfigurable ADRES Architecture for multimedia purposes. IEEE International Conference on Field Programmable Logic and Applications (FPL), pp. 106–111.
25. Mei, B., Veredas, F. J., & Masschelein, B. (2005). Mapping an H.264/AVC decoder onto the ADRES reconfigurable architecture. IEEE International Conference on Field Programmable Logic and Applications (FPL), pp. 622–625.
26. Martina, M., Masera, G., Fanucci, L., & Saponara, S. (2006). Hardware co-processors for real-time and high-quality H.264/avc video coding, 14th European Signal Processing Conference (EUSIPCO), pp. 200–204.
27. Yang, L., et al. (2005). An effective variable block-size early termination algorithm for H.264 video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, *15*(6), 784–788. doi:10.1109/TCSVT.2005.848306 (CSVT).
28. Lahti, J., et al. (2005). Algorithmic optimization of H.264/AVC encoder. *IEEE International Symposium on Circuits and Systems (ISCAS)*, *4*, 3463–3466.
29. Kant, S., Mithun, U., & Gupta, P. (2006). Real time H.264 video encoder implementation on a programmable DSP processor for videophone applications. International Conference on Consumer Electronics (ICCE), pp. 93–94.
30. Zhou, X., Yu, Z. H., & Yu, S. Y. (1998). Method for detecting all-zero DCT coefficients ahead of discrete cosine transform and quantization. *Electronics Letters*, *34*(19), 1839–1840. doi:10.1049/el:19981308.
31. Yang, J. F., Chang, S. H., & Chen, C. Y. (2002). Computation reduction for motion search in low rate video coders. *IEEE Transactions on Circuits and Systems for Video Technology*, *12* (10), 948–951. doi:10.1109/TCSVT.2002.804892 (CSVT).
32. Yu, A., Lee, R., & Flynn, M. (1997). Performance enhancement of H.263 encoder based on zero coefficient prediction. ACM International Conference on Multimedia, pp. 21–29.
33. Suh, K. B., Park, S. M., & Cho, H. J. (2005). An efficient hardware architecture of intra prediction and TQ/IQIT module for H.264 encoder. *ETRI Journal*, *27*(5), 511–524.
34. Agostini, L., et al. (2006). High throughput architecture for H.264/AVC forward transforms block. ACM Great Lakes symposium on VLSI (GLSVLSI), pp. 320–323.
35. Luczak, A., & Garstecki, P. (2005). A flexible architecture for *image* reconstruction in H.264/AVC decoders (vol. 1). European Conference Circuit Theory and Design, pp. I/217–I/220.
36. Deng, L., Gao, W., Hu, M. Z., & Ji, Z. Z. (2005). An efficient hardware implementation for motion estimation of AVC standard. *IEEE Transactions on Consumer Electronics*, *51*(4), 1360–1366. doi:10.1109/TCE.2005.1561868.
37. Yap, S. Y., et al. (2005). A fast VLSI architecture for full-search variable block size motion estimation in MPEG-4 AVC/H.264. Asia and South Pacific Conference on Design Automation (ASP-DAC), pp. 631–634.
38. Ou, C.-M., Le, C.-F., & Hwang, W.-J. (2005). An efficient VLSI architecture for H.264 variable block size motion estimation. *IEEE Transactions on Consumer Electronics*, *51*(4), 1291–1299. doi:10.1109/TCE.2005.1561858.
39. Suh, J. W., & Jeong, J. (2004). Fast sub-pixel motion estimation techniques having lower computational complexity. *IEEE Transactions on Consumer Electronics*, *50*(3), 968–973. doi:10.1109/TCE.2004.1341708.
40. Min, K. Y., & Chong, J. W. (2007). A memory and performance optimized architecture of deblocking filter in H.264/AVC. International Conference on Multimedia and Ubiquitous Engineering (MUE), pp. 220–225.
41. Shih, S. Y., Chang, C. R., & Lin, Y. L. (2006). A near optimal deblocking filter for H.264 advanced video coding. Asia and South Pacific Conference on Design Automation (ASP-DAC), pp. 170–175.
42. Parlak, M., & Hamzaoglu, I. (2006). An efficient hardware architecture for H.264 adaptive deblocking filter algorithm. First NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp. 381–385.
43. Chen, C.-M., & Chen, C.-H. (2007). An efficient pipeline architecture for deblocking filter in H.264/AVC. *IEICE Transactions on Information and Systems*, E 90–D(1), 99–107.
44. Arbelo, C., Kanstein, A., Lopez, S., Lopez, J. F., Berekovic, M., Sarmiento, R., et al. (2007). Mapping control-intensive video kernels onto a coarse-grain reconfigurable architecture: the H.264/AVC deblocking filter. Design, Automation, and Test in Europe (DATE), pp. 1–6.
45. Hwang, H., Oh, T., Jung, H., & Ha, S. (2006). Conversion of reference C code to dataflow model H.264 encoder case study. Asia and South Pacific Conference on Design Automation (ASP-DAC), pp. 152–157.
46. Lim, K. P., Wu, S., Wu, D. J., Rahardja, S., Lin, X., Pan, F., et al. (2003). Fast Inter Mode Selection, JVT-I020, 9th JVT Meeting, San Diego, United States, September.
47. Hu, Y., Li, Q., Ma, S., & Kuo, C.-C.J. (2007). Fast H.264/AVC inter-mode decision with RDC optimization. International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP), pp. 511–516.
48. Pan, F., Lin, X., Rahardja, S., Lim, K. P., Li, Z. G., Feng, G.N., Wu, D., & Wu, S. (2003). "Fast Mode Decision for Intra Prediction", JVT-G013, 7th JVT Meeting, Pattaya, Thailand, March.
49. Bauer, L., Shafique, M., Kramer, S., & Henkel, J. (2007). RISPP: rotating instruction set processing platform, 44th Design Automation Conference (DAC), pp. 791–796.
50. Bauer, L., Shafique, M., Teufel, D., & Henkel, J. (2007). A self-adaptive extensible embedded processor. International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pp. 344–347.
51. Xiph.org Test Media. Retrieved from http://media.xiph.org/video/derf/.
52. Vassiliadis, S., et al. (2004). The MOLEN polymorphic processor. *IEEE Transactions on Computers*, *53*(11), 1363–1375. doi:10.1109/TC.2004.104.
53. Vassiliadis, S., & Soudris, D. (2007). Fine- and coarse-grain reconfigurable computing. Berlin: Springer.
54. Henkel, J. (2003). Closing the SoC design gap. IEEE Computer, *36*(9), 119–121 (September).

Muhammad Shafique (S'07) received the M.Sc. degree in information technology from the Pakistan Institute of Engineering and Applied Sciences in 2003. He is currently working toward the Ph.D. degree at the Chair for Embedded Systems, University of Karlsruhe, Karlsruhe, Germany, under the supervision of Prof. Dr. Jörg Henkel. Prior to beginning his doctoral work, he was a Senior Embedded Systems Engineer with Streaming Networks, where he was involved in the development and optimization of H.264 Encoder on VLIW processors. His main research interests are extensible and reconfigurable processors for low-power mobile multimedia with a focus on dynamically varying run-time situations. He holds one U.S. patent. Mr. Shafique was the recipient of six Gold Medals and the Best Master Thesis Award.

Lars Bauer (S'05) received the M.Sc. degree in information technology from the University of Karlsruhe, Karlsruhe, Germany, in 2004, where he is currently working toward the Ph.D. degree at the Chair for Embedded Systems (CES) under Prof. Dr. Jörg Henkel. His main research interests are extensible processors and reconfigurable computing systems with a focus on dynamically varying run-time situations and concepts that allow systems to adapt to changing requirements.

Jörg Henkel (M'95-SM'01) is currently with Karlsruhe University (TH), Germany, where he is directing the Chair for Embedded Systems CES. Before, he was with NEC Laboratories in Princeton, NJ. His current research is focused on design and architectures for embedded systems with focus on low power and reliability. Dr. Henkel has organized various embedded systems and low power ACM/IEEE conferences/symposia as General Chair and Program Chair and was a Guest Editor on these topics in various Journals like the IEEE Computer Magazine. He is/has been a steering committee member of major conferences in the embedded systems field like at ICCAD and is also an editorial board member of various journals like the IEEE TVLSI. He has given full/half-day tutorials at leading conferences like DAC, ICCAD, DATE etc. In 2007 his co-edited book "Designing Embedded Processors - A low Power Perspective" appeared with Springer. Dr. Henkel is the Editor-in-Chief of the ACM Transactions on Embedded Computing Systems (ACM TECS) and holds nine US patents.