

# Specialization of the Cell SPE for Media Applications

Cor Meenderinck and Ben Juurlink

Computer Engineering Laboratory

Faculty of Electrical Engineering, Mathematics, and Computer Science

Delft University of Technology, Delft, The Netherlands

{cor,benj}@ce.et.tudelft.nl

**Abstract**—There is a clear trend towards multi-cores to meet the performance requirements of emerging and future applications. A different way to scale performance is, however, to specialize the cores for specific application domains. This option is especially attractive for low-cost embedded systems where less silicon area directly translates to less cost. We propose architectural enhancements to specialize the Cell SPE for video decoding. Specifically, based on deficiencies we observed in the H.264 kernels, we propose a handful of application-specific instructions to improve performance. The speedups achieved are between 1.84 and 2.37.

**Index Terms**—architecture, media, domain specific accelerator, specialization.

## I. INTRODUCTION

We have entered the era of Chip MultiProcessors (CMPs), where performance gains are to be achieved by exploiting thread level parallelism. The ever growing transistor budget allows an increase in the number of cores. The impact of the power wall, however, is increasing over time [1], limiting performance growth. Therefore, increased parallelism has to be complemented with increased power efficiency. The latter can be achieved by specializing cores for specific application domains. We refer to such a core as an accelerator.

In this paper we propose a media accelerator based on the Cell SPE (Synergistic Processing Element) architecture. Through a thorough analysis of the H.264 video decoding kernels, we identify Instruction Set Architecture (ISA) deficiencies such as the inability to transpose a matrix efficiently, lack of saturating arithmetic, and the omission of scalar instructions. Based on these deficiencies we propose new application-specific instructions. We show that by adding a few (at most a dozen) application-specific instructions, significant (more than a factor of 2) performance improvements can be obtained. The Cell SPE was chosen as the baseline because it is already optimized for computation-intensive applications but not for H.264 video decoding.

This paper is organized as follows. An overview of the baseline SPE architecture is provided in Section II. The experimental setup is described in Section III. In Section IV we present the enhancements that specialize the SPE for media applications. Evaluation results of the media accelerator are presented in Section V. Related work is discussed in Section VI, while Section VII concludes the paper.

## II. OVERVIEW OF THE SPE ARCHITECTURE

The Cell processor consists of one PPE (Power Processing Element) and eight SPEs. The PPE is a general purpose PowerPC that runs the operating system and controls the SPEs. The SPEs function as accelerators; they operate autonomously but are depending on the PPE for receiving tasks to execute. The SPE consists of a Synergistic Processing Unit (SPU), a Local Store (LS), and a Memory Flow Controller (MFC) as depicted in Figure 1. The SPU has direct access to the LS, but global memory can only be accessed through the MFC by DMA commands. As the MFC is autonomous, a double buffering strategy can be used to hide the latency of global memory access. While the SPU is processing a task, the MFC is loading the data needed for the next task into the LS.

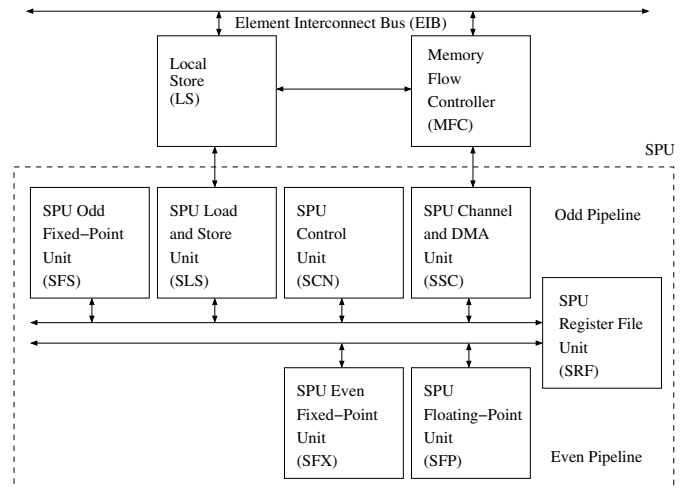


Fig. 1. Overview of the SPE architecture (based on [2]).

The SPU has 128 registers, each 128 bits wide. All data transfers between the SPU and the LS are 128-bit wide. Also the LS accesses are 128-bit aligned. The ISA of the SPU is completely SIMD and the 128-bit vectors can be treated as one quadword (128-bit), two doublewords (64-bit), four words (32-bit), eight halfwords (16-bit), or 16 bytes.

There is no scalar datapath in the SPU, but scalar operations are performed using the SIMD registers and pipelines. For each data type, a preferred slot is defined in which the scalar

value is maintained. Most of the issues of scalar computations are handled by the compiler.

The SPU has six functional units, each assigned to either the even or odd pipeline. The SPU can issue two instructions concurrently if they are located in the same doubleword and if they execute in a different pipeline. Instructions within the same pipeline retire in order. For more details on the SPU architecture, the reader is referred to [3].

The SPE architecture is a good starting point for developing a media accelerator for the following reasons. First, the *DMA* style of accessing global memory suits video decoding very well as most required data is known in advance. The only exception to this is found in motion compensation. The data required for this kernel is determined by the motion vectors which are encoded in the bitstream and thus not known a priori. Second, the large *register file* allows to keep the most important data in registers, avoiding many loads and store. Producing code that exploits this opportunity is not trivial though. Third, the *local store* of the SPE is 256KB large and has to contain both data and instructions. As the media accelerator is designed to work in conjunction with a general purpose processor, it does not have to run the entire application. Typical parts to accelerate, e.g., decoding one macroblock, fit in the LS. Therefore, the current size of the LS is sufficient.

### III. EXPERIMENTAL SETUP

As benchmarks we used the kernels of an H.264 video decoding application. H.264 is currently the best video coding standard in terms of compression and quality [4]. However, it is also very computationally demanding. One of the best publicly available H.264 decoders is FFmpeg [5]. Versions of the kernels of this application ported to the Cell SPE and SIMDimized [6] were available to us. We used the following H.264 kernels for the analysis in this work: Inverse Discrete Cosine Transform (IDCT), Deblocking Filter (DF), and Luma interpolation (Luma). The chroma interpolation kernel is very similar to the Luma kernel and was therefore omitted. There is also an entropy decoding kernel in H.264. We do not consider this kernel in this research as it is highly bit serial. We are investigating an accelerator specifically for this kernel.

We used spu-gcc version 4.1.1. The architectural enhancements we implemented were made available to the programmer through intrinsics or inline assembly in case intrinsics would require complex modifications. The modifications assured that the compiler can handle the new instructions, but not necessarily in an optimized way. Optimizing the compiler for the new instructions is beyond the scope of this project.

As the media accelerator is based on the Cell SPE, we used CellSim [7] as the simulation platform. Before modifying the simulated architecture we validated the accuracy of the simulator. To match the real SPE some parameters were adjusted, like instruction latencies, fetch width, fetch buffer size. Furthermore, branch hint instructions and the branch miss penalty were implemented. Finally, the instruction fetch rate (IPC) of CellSim was adjusted. The real processor can fetch

two instructions per cycle, unless they execute both in the odd or even pipeline. CellSim does not distinguish between the odd and even pipeline and therefore initially fetched too many instructions per cycle on average compared to the IBM full system simulator SystemSim. Through experiments we observed that an instruction fetch rate of 1.4 in CellSim corresponds best with the behavior of the real processor.

To validate the configuration of CellSim, we compared the execution times measured in CellSim with those obtained with SystemSim. We did not compare with the real processor because the hardware counters have a precision of approximately one microsecond, which is not sufficient for our purposes.

Table I shows the execution times and the instruction count of all the kernels using both SystemSim and CellSim. It shows that the difference in execution time is at most 2.5%. Although one would expect the number of executed instructions to be the same in both simulators they are slightly different. This difference is caused by the different implementation of the profiler tool of SystemSim and that of CellSim. Altogether, from these results we conclude that CellSim is sufficiently accurate for evaluating the architectural enhancements described in this paper.

TABLE I  
COMPARISON OF EXECUTION TIMES AND INSTRUCTION COUNT IN  
SYSTEMSIM AND CELLSIM

Kernel	Cycles			Instructions		
	SystemSim	CellSim	Error	SystemSim	CellSim	Error
IDCT8	917	896	-2.3%	1099	1103	0.4%
IDCT4	2571	2524	-1.9%	1853	1858	0.3%
DF	121465	122122	0.5%	101111	101480	0.4%
Luma	2534	2598	2.5%	2557	2572	0.6%

### IV. THE MEDIA ACCELERATOR ENHANCEMENTS

To specialize the SPE for media applications we identified deficiencies in the execution of the H.264 kernels. A thorough analysis down to the assembly level was performed. Among others, this analysis comprised the following. We performed extensive profiling to determine the most time consuming parts. Analyzing trace files allowed to determine why those parts were time consuming. Also, we identified rather simple operations that involve a lot of overhead in the SPE ISA. Furthermore, both C and assembly code were analyzed to find opportunities for instruction collapsing.

Due to space limitations, we cannot present all identified deficiencies nor can we describe all details of the ISA enhancements. Table II provides an overview of all the instruction classes that were added to the SPE. For each instruction an available opcode was chosen. For some instructions (marked with \* in Table II) a new instruction format was defined. The I17 format is equal to R17 but has two immediate values; the RRI3 format is explained in Section IV-D. A detailed description of the instructions can be found in [8].

TABLE II

OVERVIEW OF THE MEDIA SPECIFIC INSTRUCTION CLASSES. TWO INSTRUCTION FORMATS (MARKED WITH \*) WERE ADDED TO THE ISA.

Instruction class	Description	Latency	Instr. format
As2ve	Add scalar to specific vector element	3	RI7/II7*
Asp	Add two vectors, saturate to smaller data type, and pack	4	RR
Clip	Clip elements between min and max	4	RR
Idct4	Performs 2-dimensional 4×4 IDCT	10	RR
Idct8	Intra-vector one-dimensional 8-point IDCT	8	RR
Lds	Load scalar into preferred slot	7	RR
Mpytr	Multiply and truncate to original data size	7	RR/RI7/RRR
Mpy_byte	Multiply, madd, and msub for bytes	7	RR/RI7/RRR
Sat_pack	Saturate elements of two vectors to smaller data type and pack into one vector	2	RR
Sfxsh	Several combinations of add, sub, and shifts	5	RR13*
Swapoe	Swap odd and even elements of two vectors	4	RR
Unaligned L/S	Load or store quadword from unaligned memory address	7	RR/RI10

### A. Support for Scalar Operations

The `lds` (Load Scalar) and `as2ve` (Add Scalar to Vector Element) instructions optimize scalar processing. Since the SPE has a SIMD-only architecture, scalar operations are performed by placing the scalar in the so-called preferred slot and using the SIMD functional units. For operations that involve only scalar variables this works well; the compiler places the scalar variable in a memory location congruent with the preferred slot even if this wastes memory. But if the operand is a vector element a lot of rearrangement overhead is required to move the desired element to the preferred slot. Rather than providing a full scalar datapath, which would increase the silicon area significantly, we propose adding a few scalar instructions.

The `lds{b,hw,w}` instructions load a byte, halfword, or word from the local store into the preferred slot of a SIMD register. The effective address of `ldsw` is word-aligned, but not necessarily quadword-aligned, as in the SPE architecture. The addressing mode is base plus scaled index, i.e., the effective address of `ldsw $2,$6,$5` is  $\$6 + 4 \times \$5$ . Note that the base plus scaled index addressing mode is supported in several ISAs (e.g., x86), but not found to be very useful because the compiler can often eliminate the index variables. In this case, however, the datapath is completely SIMD and the index is used to select a specific element of the loaded quadword.

The `lds` instructions can, for example, be used for the table lookups in the Deblocking Filter (DF) kernel. The elements of the table are stored in consecutive memory locations. Loading a specific element of the table normally requires computing the address of the vector containing the element, loading the vector, and finally rotating the vector such that the desired element is moved to the preferred slot. The `lds` instructions perform these operations in one simple instruction.

The `as2ve{b,hw,w}[i]` instructions add a scalar to a specific element of a SIMD vector. For example, the instruction `as2vehw rt,ra,elt` adds the scalar halfword contained in the preferred slot of register `ra` to element `elt` of the SIMD vector contained in register `rt`, where `elt` is a value between 0 and 7. In the SPE architecture,

the elements are numbered from left to right, where the left position corresponds to the most significant byte and bit. In the immediate versions of these instructions, operand `ra` is replaced by an immediate value. Figure 2 depicts an example how the `as2ve` instructions can be used to speed up the IDCT kernels. Note that these instructions provide additional advantages over the `lds` instructions. If `lds` is used to load a specific element of a SIMD vector, the result of the addition still needs to be merged with the other elements of the SIMD vector, which incurs considerable overhead.

original			
<code>ai</code>	<code>\$2,\$4,14</code>	<code>; \$2: &amp;a[0] + 14</code>	
<code>lqd</code>	<code>\$8,0(\$4)</code>	<code>; \$8:  a[0]    a[1]   ... a[16]  </code>	
<code>chd</code>	<code>\$7,0(\$4)</code>	<code>; \$7: mask for halfword insertion</code>	
<code>rotqby</code>	<code>\$2,\$8,\$2</code>	<code>; \$2:  a[16]    a[0]   ... a[15]  </code>	
<code>ahi</code>	<code>\$2,\$2,32</code>	<code>; \$2:  a[16]+32    a[0]+32   ...  </code>	
<code>shufb</code>	<code>\$6,\$2,\$8,\$7</code>	<code>; \$6:  a[0]+32    a[1]   ... a[16]  </code>	
enhanced			
<code>lqd</code>	<code>\$8,0(\$4)</code>	<code>; \$8:  a[0]    a[1]   ... a[16]  </code>	
<code>as2vehwi</code>	<code>\$8,32,0</code>	<code>; \$8:  a[0]+32    a[1]   ... a[16]  </code>	

Fig. 2. Example of how the `as2ve` instructions speed up scalar operations, in this case: `a[0] += 32`; . On the top is the original assembly code, while the bottom depicts the assembly code using the `as2vehwi` instruction.

### B. Support for Clip & Pack Operations

In this category we provide three instruction classes: `clip`, `sat_pack` (saturate and pack), and `asp` (add, saturate, and pack).

The `clip{hw,w}` instructions clip the elements of a vector between minimum and maximum values defined in the second and third operand. These instructions can be used to clip the index of the table lookups in the deblocking filter kernel. They can also be used to emulate saturating arithmetic. Unlike many other SIMD ISAs, the SPE ISA does not support saturating arithmetic. All considered H.264 kernels, however, produce 8-bit unsigned values while all computations are performed using 16-bit signed values in order not to lose precision. Therefore, all results need to be clipped to a value between 0 and 255 before packing the 16-bit signed values to 8-bit unsigned values. This can be performed using the `cliphw` instruction. The `clipw` instruction allows to saturate a vector of signed words to signed halfwords. In the SPE architecture

clipping is costly as it has to be performed using multiple compare and select instructions.

Often saturation is directly followed by a pack. The `sat_pack` instructions (`sp[il]2{ub,hw}`) combine these two as they saturate the elements of two vectors to a smaller data type and pack the result into one vector. The two vectors can be packed in consecutive (`sp2{ub,hw}`) or interleaved (`spil2{ub,hw}`) order into the result. The extension of the instruction (`ub` or `hw`) denotes the data types of the output vector (unsigned byte or halfword). The Luma kernel benefits best from these instructions.

For the IDCT kernels we provide the `asp2ub` instruction that combines addition, saturation, and packing. First, it adds two vectors of halfwords. Next, the sum is saturated to unsigned bytes. Finally, the resulting bytes are packed into the first doubleword of the vector.

### C. Support for Matrix Transposition

The `swapoe{b,hw,w,dw}` instructions swap the odd-numbered elements of the first source/destination SIMD register with the even-numbered elements of the second source/destination register, as illustrated in Figure 3. As mentioned before, SIMD vector elements are numbered from left to right, starting with 0. Using the `swapoe` instructions matrix transposition can be performed with half as many instructions as a conventional SIMD implementation that uses permute instructions. Figure 4 depicts the matrix transposition procedure for an  $8 \times 8$  matrix. First, four `swapowedw` instructions swap the left-bottom and right-top  $4 \times 4$  blocks. In the subsequent two steps the sub-blocks are swapped with the `swapoew` and `swapoehw` instructions, respectively. Note that the transpose is performed in-place, using half as many registers as a normal transpose.

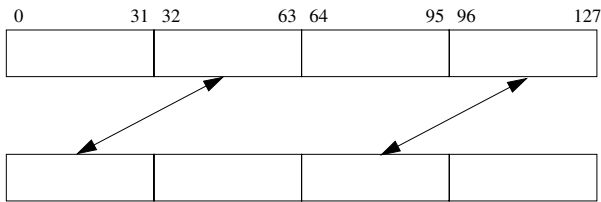


Fig. 3. The `swapoew` instruction swaps the odd and even words of the two vector operands.

Matrix transposition is a very common operation in media applications. This is because many media applications process two dimensional blocks of pixels. In H.264, for example, the elementary data is the macroblock, which is a matrix of  $16 \times 16$  pixels. Most matrix based computations allow to exploit data level parallelism using SIMD instructions. Often, however, computations have to be performed both row wise and column wise. Therefore, matrix transposes are required to rearrange the data for SIMD processing. Indeed, in all considered kernels, except Luma, the matrix is transposed twice. Note that the `swapoe` instructions have two destination registers, while the SPE register file has six read ports and two write ports.

This implies that the `swapoe` instructions cannot be scheduled together with another instruction that has a destination register. We account for this limitation by adding a cycle to the latency of the `swapoe` instructions. Specifically, the latency consists of three cycles for the predefined permutations and one cycle for the two writes, adding up to a total latency of four cycles (see also Section IV-F).

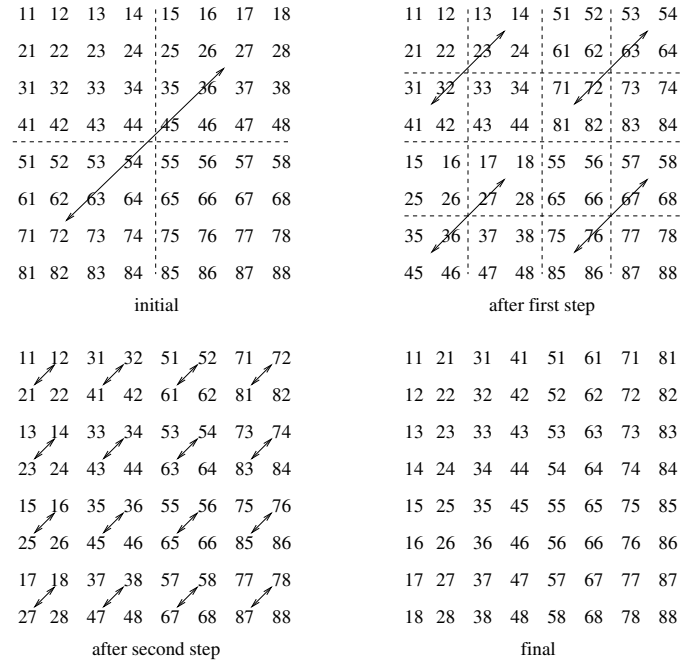


Fig. 4. Eklundh's matrix transpose algorithm [9].

### D. Support for Arithmetic Operations

SPE instructions have an execution latency of several cycles. In most cases, the latency can be hidden by instruction scheduling. In several cases, however, an arithmetic operation is immediately followed by a dependent instruction, in which case the latency cannot be hidden. In those cases it is advantageous to collapse these operations in a single instruction. Several instruction classes have been added to enhance arithmetic operations, mainly due to instruction collapsing. The added instructions are the `sfxsh` (Simple FiXed point and SHift), `mpy_byte` (multiply bytes), `mpy_tr` (multiply and truncate), and `idct` instructions.

The `sfxsh` instructions combine a simple fixed point arithmetic operation (either addition or subtraction) with a bit shift operations. Figure 5 shows the four variations that are provided. Each of them is available for both vectors of halfwords and words. The `sfxsh` instructions have four operands, which could fit the RRR instruction format (see [2] for more details). However, only few opcodes of this type are available. Therefore, we defined the RRI3 instruction format that has three normal register operands and one immediate operand that takes only three bits of the opcode. Thus, more instructions with four operands can fit in the ISA. At the same

time, three bits for the shift value is enough for all the H.264 kernels as it is always between 0 and 7.

```
shadd{hw,w}  rt,ra,rb,imm ; rt: ra + rb>>imm
shsub{hw,w}  rt,ra,rb,imm ; rt: ra - rb>>imm
addsh{hw,w}  rt,ra,rb,imm ; rt: (ra + rb)>>imm
subsh{hw,w}  rt,ra,rb,imm ; rt: (ra - rb)>>imm
```

Fig. 5. Overview of the `sfxsh` instructions. Each instruction is available for both vectors of halfwords and words.

The `mpytr` instructions perform a multiplication and truncate the results to the number of bits of the input operands, discarding the higher bits. All integer multiplication instructions in the original SPE ISA produce outputs of a larger data type than the input, for obvious reasons. As the SPE is SIMD, this implies that the output vector contains half as many data elements. Although pixel data are bytes, computations are performed on halfwords in order not to lose precision. In many cases of multiplication, however, it is known that the result will never exceed the halfword range. In these cases the `mpytr` instructions can be used in order not to lose data level parallelism. The `mpytrhw[i]` instructions perform a multiply, while the `maddtrhw` instruction performs a multiply-add.

Another approach is to perform the multiplications before unpacking pixel data to halfwords. By doing the multiplication on bytes, the result is a vector of halfwords and the unpacking is obtained for free. This approach is not always possible, but in certain cases it is beneficial. The `mpy_byte` instructions were added for this purpose. The `mpytr` and `mpy_byte` instructions were used in the interpolation filter of the Luma kernel.

The `idct8` is an *intra-vector* instruction that performs a row-wise 1D IDCT. A 2D IDCT consists of a 1D IDCT on each row followed by a 1D IDCT on each column. The column-wise 1D IDCTs can be SIMDmized efficiently using conventional SIMD instructions by performing all of them in parallel. In order to do the same for the row-wise 1D IDCTs, however, the matrix needs to be transposed twice, which motivated the `swapoe` instructions. To eliminate the transpositions completely, we propose the `idct8` instruction that performs a row-wise 1D IDCT on a vector of eight 16-bit signed values. The `idct4` instruction is similar but performs an entire  $4 \times 4$  2D IDCT. It takes two vectors as input that both contain two rows of the  $4 \times 4$  matrix.

### E. Enhanced Memory Access

In the SPE architecture accesses to the Local Store are quadword aligned. The four least significant bits of a memory address used in a load or store instruction are truncated. Thus, byte aligned memory addresses can be used but the local store always returns an aligned quadword. If a quadword has to be loaded or stored from an unaligned address this has to be done in several steps. First, two quadwords are loaded, each containing a part of the targeted quadword. Next, the two quadwords are shifted left and right respectively to put the elements in the correct slot. Finally, the two words are

combined using an OR instruction. The procedure for an unaligned store is even more complicated and involves two loads and two stores.

To overcome this limitation of the SPE architecture we implemented unaligned loads and stores. To assure backward compatibility of code the existing load and store operations were not altered. Instead we added new instructions that perform the unaligned loads and stores (`lq{d,x}u` and `stq{d,x}u`).

To enable unaligned accesses some modifications to the local store are required. A quadword might be aligned across two memory lines. Thus additional hardware is required to load or store the unaligned quadword from memory. To account for this we added one cycle to the latency of the local store.

### F. Instruction Latencies

The instruction latencies are conservative and have been based on the latencies of similar instructions and by adding an extra cycle if the new instruction is slightly more complex than a similar existing instruction. For example, a possible implementation of the `as2ve` instruction uses the FX2 (fixed point) pipeline and replicates the scalar to all SIMD slots (see Figure 6). The multiplexer at the SIMD slot indicated by the I7 value is set to select the replicated RA value. For the other SIMD slots, the mux selects the zero input. The vector RT goes to the other input of the ALU. The ALUs perform a normal addition operation. The latency of the FX2 pipeline is two cycles. Adding another cycle for the slightly increased complexity results in three cycles.

Similarly, the latency of the `idct8` instruction is assumed to be eight cycles, since there are four simple fixed point operations in the critical path and fixed point operations take two cycles in the SPE architecture.

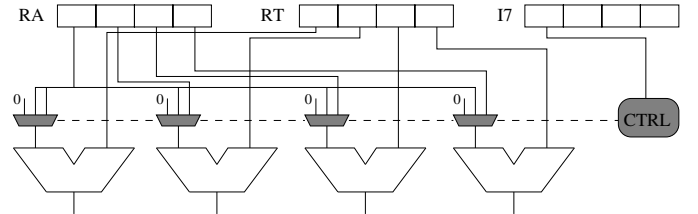


Fig. 6. Example of an implementation of the `as2ve` instructions. The gray units are added or modified and slightly increase the latency.

## V. EVALUATION

Figure 7 depicts the speedup and reduction of the instruction count achieved on the considered H.264 kernels. It shows that the speedup achieved is between 1.84x and 2.37x, and the reduction of the instruction count is between 1.75x and 2.67x. In most cases the speedup is lower than the reduction of the instruction count. This is expected since multiple shorter-latency instructions have been replaced by a single instruction but with a longer latency. In other words, the latency of the arithmetic operations stays the same. For example, the latency

of the `idct8` instruction is assumed to be 8 cycles, because there are 4 simple fixed point operations on the critical path, each taking two cycles.

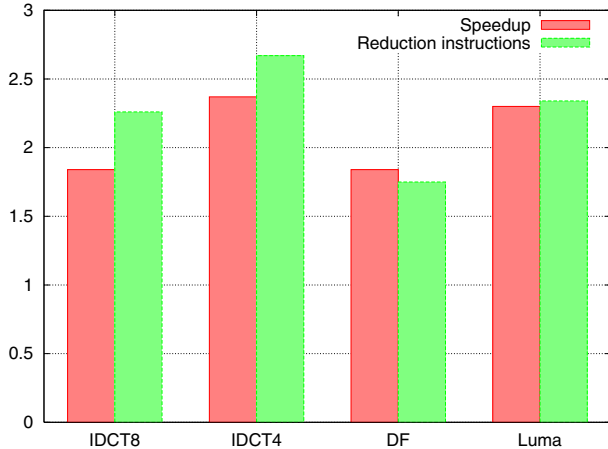


Fig. 7. Speedup and reduction of the instruction count achieved on the H.264 kernels.

The application-specific instructions mainly reduce the overhead and eliminate stalls due to data dependencies. For the DF kernel, however, the speedup is slightly higher than the instruction count reduction. This is because many branch miss stalls are avoided. The `swapoe` instructions allow to keep the entire macroblock in the register file. In order to achieve this, functions had to be replaced by macros avoiding many branches. Table III details the speedup and the reduction of the instruction count achieved by the separate instructions that have been added to the SPE.

Surprisingly, the `swapoe` instructions provide no benefit for the IDCT8 kernel, even though they reduce the number of instructions required for matrix transposition by 2x. The time taken by this kernel is mainly determined by data dependencies. Between the data dependent instructions there was enough time to execute the matrix multiplication instructions. Reducing the latter does not change the dependencies and therefore neither the execution time of the kernel. For the DF kernel, on the other hand, they provide a significant speedup (by 1.35x), because the matrix can be transposed in the register file and no loads/stores are needed to transpose the matrix in the local store. Not surprisingly, for the IDCT kernels, the `idct{4, 8}` instructions provide the largest benefit, by 1.39x for the IDCT8 kernel and by 1.23x for the IDCT4 kernel. The Luma kernel benefits most from the unaligned load/store instructions. It loads data from a position determined by the motion vectors, which in most cases is an unaligned position.

A full power efficiency analysis has yet to be done. However, the presented results do provide a clear indication that the proposed architecture has gained on power efficiency. The same operations are performed in about half the amount of cycles and about half the amount of instructions. Those translate to less switching in the local store, the register file, the control, etc. Many of the new instructions also reduce the

switching in the ALUs.

## VI. RELATED WORK

A linear time matrix transposition method was proposed in [10], which uses a vector register file with diagonal access. For  $16 \times 16$  matrices our approach is as fast but much simpler. The `swapoe` instructions are similar to the `mix` instructions used in HP's MAX-2 [11].

The (I)DCT algorithm used in the H.264 standard is a low complexity version, first proposed in [12]. A hardware implementation was proposed in [13], which was designed to be used in a streaming environment and therefore is not very suitable for a programmable media accelerator. It also does not exploit data level parallelism as our approach does.

Alignment issues on different multimedia SIMD architectures have been investigated in [14]. The paper describes several software approaches to minimize the overhead. The benefits of supporting unaligned accesses in hardware is studied in detail in [15] for the AltiVec/VMX architecture. The TM3270 Media-Processor, which was specialized for H.264, also supports unaligned memory accesses.

A combination of `pack` and saturation is also found in the `pack.sss` and `pack.uss` instructions of the IA64 architecture. Also MMX has instructions that combine `pack` and `saturate`. In the TriMedia there are a few instructions that include a clip operation.

An ISA specialization for video coding was presented in [16]. The authors suggested some simple instructions for the MPEG-4 standard. Many of the proposed instructions, e.g., arbitrary permute, MAC (multiply-accumulate), round, and average, are currently being used in architectures. In [17] an ISA specialization is presented for H.263 encoding. Only a few instructions are described. Some are encoding specific, like SAD (sum of absolute differences). Their IDCT instruction cannot be used for H.264 due to the different algorithm.

## VII. CONCLUSIONS

In this paper we described the specialization of the Cell SPE for media applications, specifically H.264. We enhanced the architecture with twelve instructions classes, most of which have not been reported before, and that can be divided in five groups. First, we added instructions to enable efficient scalar-vector operations. Second, we added several instructions that perform saturation and packing as needed by the H.264 kernels. Third, we added the `swapoe` instructions to perform fast matrix transposition, needed by most kernels. Fourth, we added specialized arithmetic instructions that compute frequently used simple arithmetic expressions. Finally, we added support for unaligned access to the local store, especially needed for motion compensation. The speedup achieved on the H.264 kernels is between 1.84x and 2.37x, while the dynamic instruction count reduction is between 1.75x and 2.67x. The largest performance improvement is achieved by the `swapoe` instructions because it allowed more efficient usage of the register file, by the IDCT instructions because

TABLE III

OVERVIEW OF THE EFFECT OF ALL ARCHITECTURAL ENHANCEMENTS ON THE EXECUTION TIME AND INSTRUCTION COUNT OF THE KERNELS.

	Speedup				Reduction instructions			
	IDCT8	IDCT4	DF	Luma	IDCT8	IDCT4	DF	Luma
As2ve	1.04	1.05			1.03	1.06		
Asp	1.15	1.11			1.19	1.23		
Clip			1.06	1.05			1.05	1.07
Idct4		1.23				1.42		
Idct8	1.39				1.46			
Lds			1.12				1.13	
Mpytr				1.10				1.14
Mpy_byte				1.13				1.20
Sat_pack				1.19				1.12
Sfxsh	1.07			1.04	1.12			1.03
Swapoe	0.99		1.35		1.12		1.36	
Unaligned L/S		1.21		1.29		1.08		1.28
Total	1.84	2.37	1.84	2.30	2.26	2.67	1.75	2.34

they completely avoid matrix transposes, and by the unaligned load/store instructions because of the large overhead reduction.

The proposed IDCT instructions are unconventional in the sense that they perform intra-vector operations. In general SIMD instructions perform operations on elements that are in the same slot, but in different vectors. We intend to investigate if such intra-vector SIMD instructions are beneficial for other kernels and other application domains as well. Other future work includes the performance analysis of the entire H.264 codec as well as a full power efficiency analysis.

#### ACKNOWLEDGEMENTS

This work was supported by the European Commission in the context of the SARC integrated project #27648 (FP6).

#### REFERENCES

- [1] C. Meenderinck and B. Juurlink, "(When) Will CMPs hit the Power Wall?" in *Proc. Workshop on Highly Parallel Processing on a Chip*, 2008.
- [2] *Cell Broadband Engine Programming Handbook*, IBM, 2006. [Online]. Available: [http://www.bsc.es/plantillaH.php?cat\\_id=326](http://www.bsc.es/plantillaH.php?cat_id=326)
- [3] B. Flachs *et al.*, "The Microarchitecture of the Synergistic Processor for a CELL Processor," in *Proc. Int. Solid-State Circuits Conference*, 2005.
- [4] T. Oelbaum, V. Baroncini, T. Tan, and C. Fenimore, "Subjective Quality Assessment of the Emerging AVC/H.264 Video Coding Standard," in *Int. Broadcast Conference*, 2004.
- [5] "The FFmpeg Libavcoded." [Online]. Available: <http://ffmpeg.mplayerhq.hu/>
- [6] A. Azevedo, C. Meenderinck, B. Juurlink, M. Alvarez, and A. Ramirez, "Analysis of Video Filtering on the Cell Processor," in *Proc. Int. Symposium on Circuits and Systems*, 2008.
- [7] "CellSim: Modular Simulator for Heterogeneous Multiprocessor Architectures." [Online]. Available: <http://pcsostrs.ac.upc.edu/cellsim/doku.php/>
- [8] C. Meenderinck and B. Juurlink, "The SARC Media Accelerator - Specialization of the Cell SPE for Media Acceleration," Delft University of Technology, Tech. Rep., 2009, <http://ce.et.tudelft.nl/publications.php>, CE-TR-2009-01.
- [9] J. Eklundh, "A fast computer method for matrix transposing," *IEEE Transactions on Computers*, vol. 100, no. 21, pp. 801–803, 1972.
- [10] B. Hanounik and X. Hu, "Linear-time Matrix Transpose Algorithms Using Vector Register File With Diagonal Registers," in *Proc. Int. Parallel & Distributed Processing Symposium*, 2001.
- [11] R. Lee and J. Huck, "64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture," in *Proc. Compcon*, 1996.
- [12] H. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-complexity Transform and Quantization with 16-bit Arithmetic for H. 26L," in *Proc. IEEE Int. Conference on Image Processing (ICIP02)*, 2002.
- [13] T. da Silva, C. Diniz, J. Vortmann, L. Agostini, A. Susin, S. Bampi, and B. Pelotas-RS, "A Pipelined 8x8 2-D Forward DCT Hardware Architecture for H. 264/AVC High Profile Encoder," in *Advances in Image and Video Technology: Second Pacific Rim Symposium, PSIVT*, 2007.
- [14] A. Shahbahrani, B. Juurlink, and S. Vassiliadis, "Performance Impact of Misaligned Accesses in SIMD Extensions," in *Proc. Workshop on Circuits, Systems and Signal Processing*, 2006.
- [15] M. Alvarez, E. Salami, A. Ramirez, and M. Valero, "Performance Impact of Unaligned Memory Operations in SIMD Extensions for Video Codec Applications," in *Proc. Int. Symposium on Performance Analysis of Systems & Software*, 2007.
- [16] M. Berekovic, H. Stolberg, M. Kulaczewski, P. Pirsch, H. Möller, H. Runge, J. Kneip, and B. Stabernack, "Instruction Set Extensions for MPEG-4 Video," *The Journal of VLSI Signal Processing*, vol. 23, no. 1, pp. 27–49, 1999.
- [17] Z. Shen, H. He, Y. Zhang, and Y. Sun, "VS-ISA: A Video Specific Instruction Set Architecture for ASIP Design," in *Proc. Int. Conference on Intelligent Information Hiding and Multimedia Signal Processing*, 2006.