The SARC Media Accelerator

Specialization of the Cell SPE for Media Acceleration

Technical Report
CE-TR-2009-01

January 2009
Cor Meenderinck and Ben Juurlink
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
cor@ce.et.tudelft.nl

TU Delft

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.

Contents

1 Introduction

We have entered the era of Chip MultiProcessors (CMPs). They are already being deployed in many market segments. It is generally expected that the number of cores will grow at a steady rate. Initially this growing TLP will deliver the increase in performance as predicted by Moore's Law.

However, the impact of the power wall is also increasing over time [1]. That means that the power budget will be the main limitation of performance and not technology or the transistor budget. Clock frequencies will have to be throttled down and eventually it will not be possible to use all available cores concurrently. In such a situation performance increase can only be obtained by improving the power efficiency of the system. This will lead us to specialization of cores (often referred to as domain specific accelerators) such that those can run a specific domain of applications in short time using little area, and thus being power efficient.

An important application domain of today as well as of the future is media. Therefore, in this report we investigate the specialization of an existing core for media applications. Specifically we choose H.264 video decoding as the target application. H.264 is the best video coding standard in terms of compression ratio and picture quality currently available and is a good representative of future media workloads. As the baseline core for specialization we take the Cell SPE. The Cell broadband engine has eight SPEs that operate as multimedia accelerators for the PowerPC (PPE) core. Therefore it is an excellent starting point for specialization.

This document is organized as follows. Section 2 provides a short overview of the SPE architecture. In Section 3 the experimental setup is described. The architectural enhancements of and modifications to the SPE are described in Section 4. The resulting media accelerator is evaluated in Section 5. Related work is described in Section 6 and Section 7 concludes this report.

2 The SPE Architecture

The baseline architecture for this research is the Cell SPE (Synergistic Processing Element) [2]. The Cell processor is one of the most advanced chip multiprocessor available today. It consists of one PPE and eight SPEs (Synergystic Processing Elements). 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.

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 quadword aligned. The ISA of the SPU is completely SIMD and the 128-bit vectors can be treated as one quadword (128-bit), two double words (64-bit), four words

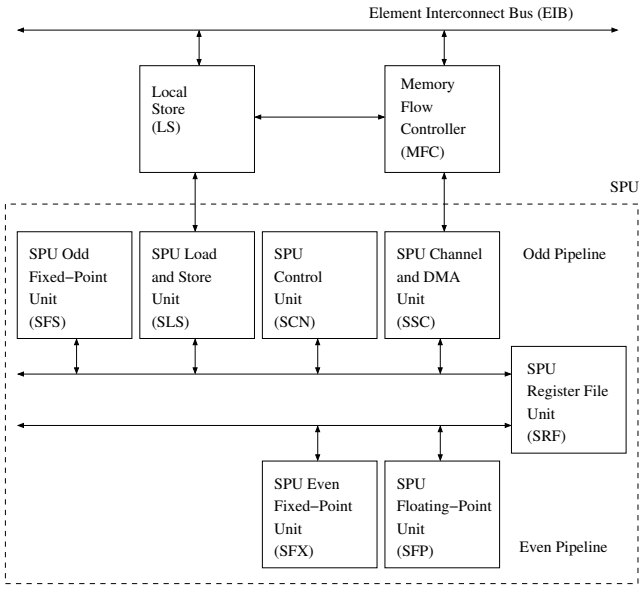


Fig. 1: Overview of the SPE architecture (based on [3]).

(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.

The Cell SPE was chosen as a starting point for specialization because of its properties. It was mentioned before that the SPE functions as an accelerator in the Cell processor. Therefore, from the system level, the SPE is a good choice. There are also some other architectural reasons to choose the SPE.

The fundamental idea behind the **DMA** style of accessing global memory is the *shopping list model*. In this model, before processing a task all the required data is brought in the local store through DMA. For video decoding this model suits well as almost all required data is known a priori. The only exception is in the motion compensation where the motion vector can point to an arbitrary location in a previously decoded frame. In future work we will investigate if the latency of loading the reference frame data can be hidden by other computation.

The **register file** of the SPE is rather large as it has 128 registers 128-bit each. As a result, the data a kernel is processing fits entirely in the register file in most cases. However, getting the compiler to generate code that does so is not so easy. Basically all pointers have to be replaced by arrays that can be declared `register`. Also the compiler should be able to resolve the array indices at compile time. Often kernels consist of multiple functions. To maintain data in the register file and have all the functions operate on these registers, function calls have to be avoided, i.e., the functions have to be replaced by macros.

For large kernels such as the Luma and Chroma interpolation[1] this is far from trivial. Moreover, the effect of this on the code size will not be insignificant and might constitute a problem.

The **local store** (LS) of the SPE is 256KB large and has to contain both data and instructions. The entire H.264 decoding application does not fit in this LS, but it doesn't have to. The media accelerator is meant to function in cooperation with a general purpose core. The latter would run the main application and spawns threads to on or more accelerators. A good parallelization strategy for such a system is described in [4]. It applies data partitioning and divides macroblocks among the available cores for decoding. Thus, only the code that decodes one macroblock has to fit in the LS. Taking this code from FFmpeg and compiling it for the SPE without optimizations it is 256KB large. However, compilation with the -O3 optimizations results in a code of 155KB while using -Os results in 118KB. Note that also 24KB for data is required and also some additional memory for intermediate values, a stack, etc. Thus the LS size is large enough but leaves little room to implement optimizations as mentioned in the previous paragraph.

The SPU is completely SIMD with a **vector width** of 128 bits and has no scalar path. Scalar operations can be performed using the SIMD path and thus a scalar path is not necessary. The width of the SIMD vectors seems to be rather optimal. Although the pixel components are encoded in 8 bits, all computation is done using 16-bit values in order not to loose precision. That means that in one SIMD vector eight elements ca be stored. The kernels[1] operate mostly on 4×4 and 8×8 sub-blocks and would not benefit very much of a larger SIMD vector. In the deblocking filter the functions for the luma components could use a larger vector size. In the motion compensation only the functions for the Luma 16×16 , 16×8, and 8×16 mode could benefit from this. However, using a larger vector size would increase the overhead for all 8×8 based functions. An analysis of real movies could be interesting to see what modes generally are used.

3 Experimental Setup

In this section we describe the experimental setup consisting of benchmarks, compiler, and simulator. The target baseline architecture is the Cell SPE [2]. The Cell processor is the most advanced chip multiprocessor available today. The SPE is a SIMD core targeted at multimedia, but not very specialized for the target domain. Thus it makes a perfect starting point for investigating specialization for media applications.

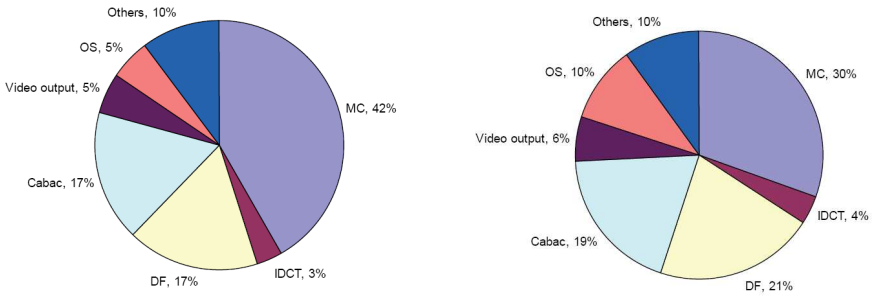
3.1 Benchmark

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 [5]. However it is also very computationally demanding. One of the best publicly available H.264 decoders is FFmpeg [6]. Versions of the kernels of this application ported to the Cell SPE [7] and SIMDimized were available to us. We used the following H.264 kernels for the analysis in this work: Inverse

[1] The H.264 kernels used throughout this work are presented in the next section.

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 did not use this kernel in this research as it is highly bit serial. We are investigating an accelerator specifically for this kernel.

Figure 2 shows the execution time breakdown of the entire FFmpeg application. This analysis was performed on a PowerPC with Mac-Os-X and running at 1.6 GHz. Both the scalar and the Altivec implementations are depicted. Cabac is one of entropy decoding algorithms part of the H.264 standard. Suppose a general purpose core performs the operating system (OS) calls, the video output, and the 'others' parts. The entropy decoding can best be done using a special Cabac accelerator while the other kernels run on the media accelerator. Using this approach the media accelerator has the largest workload: 55% assuming Altivec compared to 26% for the gpp and 19% for the Cabac accelerator. Therefore, any speedup achieved in the media core directly translates to a speedup of the total application. A detailed analysis of the used kernels will be presented in the results section.



(a) Scalar (b) Altivec

Fig. 2: Execution time breakdown of the FFmpeg H.264 decoder for the scalar and Altivec implementation.

3.2 Compiler

We used spu-gcc version 4.1.1. Most of the architectural enhancements we implemented were made available to the programmer by intrinsics. In some cases implementing the intrinsic would require complex modifications to the compiler. Those new instructions are only available by using inline assembly. We modified the gcc toolchain to support the new intrinsics and instructions. The procedure to modify the gcc compiler is described in [8].

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.

3.3 Simulator

Analysis of the kernels on the Cell SPE could be done using the real processor. However, analyzing architectural enhancements requires an experimental environment that can be modified. Therefore we used CellSim [9]; a cell simulator build in the Unisim environment.

3.3.1 Profiling in CellSim

First, we implemented a profiling tool for CellSim as it did not have one yet. The profiling tool allows to gather many statistics (like cycles, instructions, IPC, etc.) either from the entire application or from a specific piece of selected code. The profiler is controllable from the native code (the code running on the simulated processor). Furthermore, the profiler allows to generate a trace output for detailed analysis of the execution of the kernels.

Profiling the code influences the execution, especially if the size of the profiled code is small. Each call to the profiler inserts one line of volatile inline assembly. This ensures that only code that is between the start and stop command of the profiler is measured. However, because instructions are not allowed to cross this *volatile* boundary there is less opportunity for the compiler to optimally schedule the instructions.

Section 5 presents the profiling analysis of the kernels that show what time the different execution phases take. The execution times reported in those analysis are off by approximately 25%. However, the purpose is to show the relative size of the phases rather than the actual cycles spend on them. Thus, the influence of the profiling itself is not problematic.

When reporting the total execution time of the kernels, the influence of the profiling can be neglected. For this analysis only a start and stop command have to be inserted at the beginning and end of the kernel respectively. No volatile boundaries are included inside the kernel and thus all compiler optimizations are unaltered.

3.3.2 Configuration of CellSim

CellSim has many configuration parameters, most of which have a default value that match the real Cell processor. However, the latencies of all instructions classes were all set by default to 6 cycles. We adjusted those to match the ones reported in [10] as described in Table 1. The table mentions both the name of the instruction class as used in the IBM documentation as well as the pipeline name as used in CellSim and in spu-gcc. Each instruction class maps to one of the units depicted in Figure 1 (see [10] for more details). The latency of an instruction class is the time it takes the corresponding execution unit to produce the result. More precise, it is the amount of clock cycles between the operands entering the execution unit and becoming available (by forwarding) as input for another computation. Thus, the instruction latency does not include instruction fetch, decode, dependency check, issue, routing, register file access, etc. For a detailed picture of the SPU pipeline, the reader is referred to [2].

The table does not include all latencies. The latency of channel instructions are modelled in a different way in CellSim and were not adjusted. The latency of branches, i.e., the branch miss penalty, was not modelled in CellSim. We implemented this as well as we implemented the branch hint instructions.

Tab. 1: SPU instruction latencies.

Instruction class	Latency (cycles)	Description	CellSim pipe
SP	6	Single-precision floating-point	FP6
FI	7	Floating point integer	FP7
DP	13	Double-precision floating point	FPD
FX	2	Simple fixed point	FX2
WS	4	Word rotate and shift	FX3
BO	4	Byte operations	FXB
SH	4	Shuffle	SHUF
LS	6	Loads and stores	LSP

Furthermore, we adjusted the fetch parameters. The fetch width was set to one line of 32 instructions. The fetch buffer size was set to 78, which corresponds to 2.5 lines of 32 instructions. Finally, we adjusted the instruction fetch rate of CellSim. The real processor can fetch two instructions per cycle, but only if one executes in the odd pipeline and the other in the even pipeline. CellSim does not distinguish between the odd and even pipeline and therefore initially fetched to many instructions per cycle on average. Through experiments we observed that an instruction fetch rate of 1.4 in CellSim corresponds best with the behavior of the real processor.

To improve simulation speed some adjustments were made to the PPE configuration parameters. The issue width was set to 1000 while the L1 cache latency was made zero cycles. Both measures increase the IPC of the PPE, which is required to spawn the SPE threads, and thus speedup simulation time. These modifications have no effect on the simulation results as the kernels are run in the SPE.

3.3.3 Validation of CellSim

To validate the configuration of CellSim, a few tests were performed. The execution times measured in CellSim were checked by comparing them to the IBM full system simulator SystemSim. We did not use the real processor as SystemSim provides more statistics and is accurate as well. Those additional statistics allowed us to check the correctness of CellSim on more aspects than just execution time.

Table 2 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%. The number of executed instructions is slightly different. As the profiling was started and stopped at exactly the same point in the C code, we expect this difference in instruction count to be caused by the different implementation of the profiling tool

Altogether, from these results we conclude that CellSim is sufficiently accurate to evaluate the architectural enhancements described in this report. Comparison of other statistics generated by the two simulators also showed that CellSim is well suitable for its purpose. Off course, when evaluating architectural enhancements we always compare between results that are all obtained

Tab. 2: Comparison of execution times and instruction count in CellSim and SystemSim.

	Cycles			Instructions		
Kernel	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%

with CellSim.

4 Enhancements to the SPE Architecture

In this section we present the architectural enhancements and modifications to the SPE that create the media accelerator. The enhancements and modifications are based on the identified deficiencies in the execution of the H.264 kernels. The latter are discussed in detail in Section 5 while discussing the results for the kernels.

To identify the deficiencies in the execution of the H.264 kernels, we performed a thorough analysis down to the assembly level. Among others, this analysis comprised the following. We compared the C code with the generated assembly code to determine which simple operations require much overhead instructions. We performed extensive profiling to determine the most time consuming parts. Analyzing trace files allowed us to determine why certain parts were time consuming. We analyzed both C and assembly code to find opportunities for instruction collapsing.

The newly added instruction are presented in detail in this report. One of the details we show is the chose opcode for each instruction. We do not know if there is a certain structure in the opcodes and thus we chose them randomly from those available. We show them though, to prove that the added instructions fit (or not) in the ISA.

The notation used in this document to describe the instruction set architecture follows that used in the IBM documentation with a few extensions. First, temporary values w are 64-bit wide. Second, temporary values p are 16-bit wide. Third, temporary values d are 8-bit wide. An extensive description of the notation is provided in [11] for example.

In the code depicted in this report we use the following notation for vector types. A `vector signed short` type is denoted as `vsint16_t` while a `vector unsigned char` type is denoted as `vuint8_t`. In general the integer types are denoted as `v[s/u]int[8/16/32]_t`.

This section is organized as follows. First, we present four categories of new instructions that we added to the ISA. Finally, we propose a modification to the architecture that allows unaligned memory accesses to the local store.

4.1 Enhancing Scalar Operations

Since the SPU has a SIMD-only architecture, scalar operations are performed by placing the scalar in the so-called preferred slot and using the SIMD FUs. 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.

4.1.1 Lds Instructions

The lds (LoaD Scalar) instructions load a scalar from a quadword unaligned address inside the local store and stores it in the preferred slot of a register. The original SPU architecture allows only quadword aligned accesses with size of 128 bits. To load a scalar from a non quadword aligned location a lot of additional instructions are required.

Listing 1 shows an example of an array access. The left part of the figure shows the C code including the declaration of the used array and index variable. The middle of the figure shows the assembly code generated by the compiler for the original SPU architecture. The first instruction (`shli`) performs a shift left on the index variable `i`, stored in `$5` by 2 bits. That is, the index is multiplied by 4 and the result is the distance of the desired element to the start of the array in bytes. The second instruction (`ila`) loads the start address of the array into a register. The third instruction (`lqx`) loads the quadword (128 bits) that contains the desired element. It adds the index in bytes to start address of the array, rounds it of downward to a multiple of 16, and accesses the quadword at that address. At this point the desired element of the array is in register `$4` but is not in the preferred slot. The fourth instruction (`a`) adds the index in bytes to the start address of the array. The four least significant bits of this result is used by the last instruction (`rotqby`) to rotate the loaded value such that the desired element is in the preferred slot.

Listing 1: Example of how the lds instructions speed up code. Left is an array access in normal C code, in the middle is the original assembly code, while the right depicts the assembly code using the `ldsw` instruction.

```
1 int array[8];          shli   $2,$5,2      ila    $6,array
2 int i;                 ila    $6,array     ldsw   $2,$6,$5
3 int a = array[i];      lqx    $4,$2,$6
4                        a      $2,$2,$6
5                        rotqby $2,$4,$2
```

Using the `ldsw` (LoaD Scalar Word) instruction the same array access can be done in two instructions, as shown in the right side of Listing 1. The first instruction (`ila`) stores the base address of the array in a register. Next, the `ldsw` instruction takes as input the index (`$5`), the start address of the array (`$6`), and the destination register (`$2`). It loads the corresponding quadword from the local store, then picks the correct word and puts it in the preferred

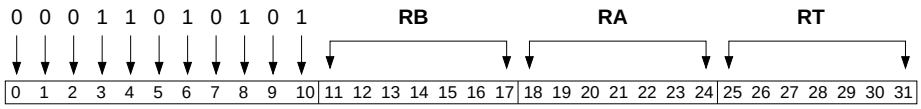
slot, and finally stores the whole in the destination register.

The lds instructions use scaled index addressing mode. That means, the effective address is the base address plus the scaled index. The ldsw instruction loads a word (four bytes) and thus its index is multiplied by four. For the other lds instructions the index is scaled in a similar way. The scalars that are loaded should be aligned according to their size. That is, words should be word aligned. The base address provided to the instruction does not have to be properly aligned, as the effective address is truncated according to the data size (see also instruction details below).

Loads and stores are handled by the SLS (SPU Load and Store) execution unit. The SLS has direct access to the local store and to the register file unit. Besides loads and stores, the SLS also handles DMA requests to the LS and load branch-target-buffer instructions. In order to implement the `ldsw` instruction a few modifications to the SLS unit are required. First, computing the target address is slightly different then a normal load. Instead of adding the two operands, the index should be multiplied by four before adding it to the start address of the array. This multiplication can be done by a simple shift left. Second, for each outstanding load (probably maintained in a buffer) two more select bits are required to store the SIMD slot in which the element will be located. These two bits are bits 2 and 3 of the target address. For normal loads, these two bits should be set to zero. Finally, when the quadword is received from the local store, the correct word should be shifted to the preferred slot. This can be done by inserting a 4-to-1 mux in the data path which is controlled by the select bits.

The lds instructions fit the RR format. The instruction details are depicted below. Although for the benchmarks used in this work only the `ldsw` instruction is required, for completeness we defined the lds instructions for halfwords and bytes as well. These element sizes are used frequently in media applications and thus it is reasonable to provide them. Doublewords are not considered in this work as neither long integers nor single precision floating point are used in the media applications targeted. Furthermore, the SPU ISA does not contain any long integer operations.

ldsw **rt,ra,rb**

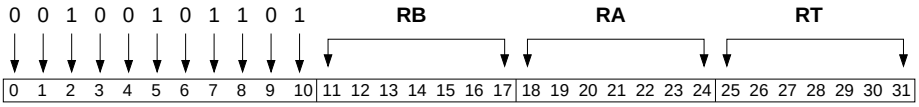


LSA	$\leftarrow (RA^{0:3} + RB^{0:3} << 2)$ & LSLR & 0xFFFFFFF0
b	$\leftarrow (RA^{0:3} + RB^{0:3} << 2)$ & 0x0000000C
Q	$\leftarrow$ LocStor(LSA,16)
$RT^{0:3}$	$\leftarrow Q^{b::4}$

The lds instructions were made available to the programmer through intrinsics of the form `d = spu_ldsx(array,index)` (x=b,hw,w) where `array` is a pointer to a word and `index` is a word. The compiler generates both the `ldsx` and the `ila` instruction required to load the desired element of the array.

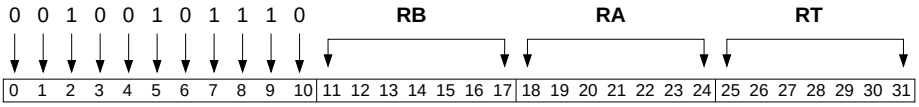
The lds instructions were added to the simulator and the latency was set to

ldshw rt,ra,rb



$$
\begin{array}{ll}
\text{LSA} & \leftarrow (RA^{0:3} + RB^{0:3} << 1) \text{ \& LSLR \& 0xFFFFFFF0} \\
b & \leftarrow (RA^{0:3} + RB^{0:3} << 1) \text{ \& 0x0000000E} \\
Q & \leftarrow \text{LocStor(LSA,16)} \\
RT^{0:1} & \leftarrow Q^{b::2}
\end{array}
$$

ldsb rt,ra,rb



$$
\begin{array}{ll}
\text{LSA} & \leftarrow (RA^{0:3} + RB^{0:3}) \text{ \& LSLR \& 0xFFFFFFF0} \\
b & \leftarrow (RA^{0:3} + RB^{0:3}) \text{ \& 0x0000000F} \\
Q & \leftarrow \text{LocStor(LSA,16)} \\
RT^{0} & \leftarrow Q^{b}
\end{array}
$$

seven cycles. A normal load or store operation takes six cycles, and we added one cycle to account for the extra mux in the datapath. Note that the latency of an instruction is the time it takes to execute it in the corresponding pipeline. Instruction fetch, decode, etc are not included. For more information on the SPU pipeline the reader is referred to [2].

4.1.2 As2ve Instructions

As2ve (Add Scalar to Vector element) instructions add a scalar to a specific element of a vector. As the SPU architecture is completely SIMD, scalar operations often require a considerable amount of time. Listing 2 shows an example from the IDCT8 kernel where the value 32 has to be added to the first element of a vector of halfwords.

The middle of the figure depicts how the normal SPU ISA handles this code. The pointer `*block` is stored in `$4`. First, it is calculated (`ai`) by how much bytes the vector must be rotated to put the desired element in the preferred slot. This shift count is stored in `$2`. Second, the vector is loaded (`lqd`) into register `$8`. Third, a mask is created (`chd`) and stored in `$7` which is needed by the shuffle latter on. Fourth, the vector is rotated (`rotqby`) based on the shift count. Fifth, the value 32 is added (`ahi`) to the desired element that now is in the preferred slot. Finally, the result of the addition is shuffled (`shufb`) back into its original position of the vector. Note that this compiler generated code is not optimal. The shift count is known a priori as pointer `*block` is quadword aligned. Apparently the compiler is not aware of that.

Using the as2ve instructions the same addition of the scalar to the vector can be done in one instruction (or two, including the load as in the example). In

Listing 2: Example of how the as2ve instructions speed up code. Left are two lines of code from the IDCT8 kernel, in the middle is the normal assembly code, while the right depicts the assembly code using as2ve instructions.

```
1 vsint16_t *block;      ai      $2,$4,14          lqd       $8,0($4)
2 block[0] += 32;        lqd     $8,0($4)          as2vehwi  $8,32,0
3                        chd     $7,0($4)
4                        rotqby  $2,$8,$2
5                        ahi     $2,$2,32
6                        shufb   $6,$2,$8,$7
```

the example of Listing 2 we used the `as2vehwi` instruction that adds a halfword immediate to a vector of halfwords. Figure 3 shows all the as2ve instructions. Only the instructions for bytes and halfwords are used in the benchmarks, but we provide the word versions as well as media application sometimes use this data type. For each data type there are two instructions. The first assumes the scalar is hold in the preferred slot of a register. The second assumes the scalar is put as immediate value in the instruction.

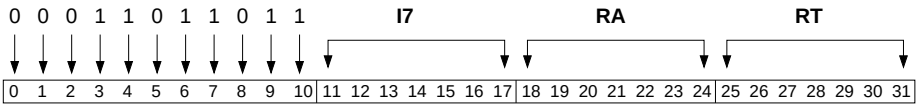
One might consider the lds instruction, augmented with an sts (store scalar) to perform this operation. However, this might incur a lot of overhead. Suppose the as2ve operation is in the middle of a series of instructions that operate on variables in the register file. In that case, first the vector should be stored in memory, second using an lds instruction the vector element is loaded into the register file, third the addition can be done, fourth a sts instruction has to be used to store the vector element back into the vector, and finally the entire vector has to be loaded again for further operation. This method requires four overhead instructions that are dependent and have a large latency.

```
as2veb    rt,ra,immb    \\rt[immb] += ra      (for vector of bytes)
as2vebi   rt,imma,immb  \\rt[immb] += imma    (for vector of bytes)
as2vehw   rt,ra,immb    \\rt[immb] += ra      (for vector of halfwords)
as2vehwi  rt,imma,immb  \\rt[immb] += imma    (for vector of halfwords)
as2vew    rt,ra,immb    \\rt[immb] += ra      (for vector of words)
as2vewi   rt,imma,immb  \\rt[immb] += imma    (for vector of words)
```

Fig. 3: Overview of the as2ve instructions.

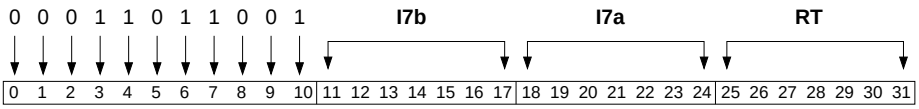
The as2ve instructions fit the RI7 format except that `as2vebi`, `as2vehwi`, and `as2vewi` have two immediate values. The instruction details are depicted below.

as2veb rt,ra,imm



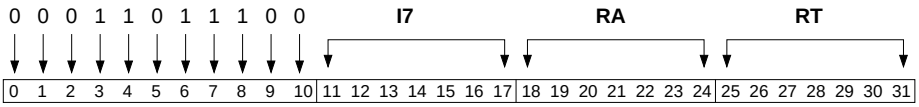
$b \leftarrow RA^{3}$
$c \leftarrow$ RepLeftBit(I7,8) & 0x0F
$d \leftarrow RT^{c}$
$RT^{c} \leftarrow d + b$

as2vebi rt,imma,immb



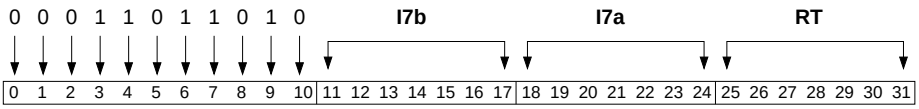
$b \leftarrow$ RepLeftBit(I7a,8)
$c \leftarrow$ RepLeftBit(I7b,8) & 0x0F
$d \leftarrow RT^{c}$
$RT^{c} \leftarrow d + b$

as2vehw rt,ra,imm



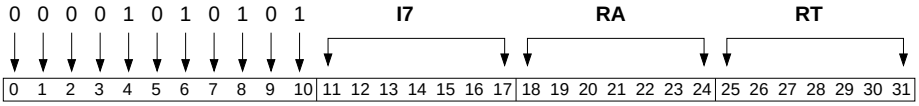
$s \leftarrow RA^{2:3}$
$b \leftarrow$ RepLeftBit(I7,8) & 0x07
$r \leftarrow RT^{2b::2}$
$RT^{2b::2} \leftarrow r + s$

as2vehwi rt,imma,immb



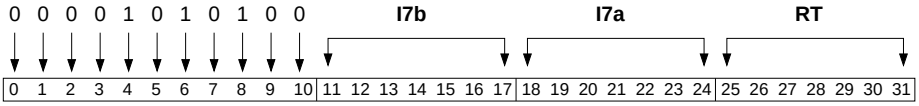
$s \leftarrow$ RepLeftBit(I7a,16)
$b \leftarrow$ RepLeftBit(I7b,8) & 0x07
$r \leftarrow RT^{2b::2}$
$RT^{2b::2} \leftarrow r + s$

as2vew rt,ra,imm



t	$\leftarrow RA^{0:3}$
b	$\leftarrow$ RepLeftBit(I7,8) & 0x03
u	$\leftarrow RT^{4b::4}$
$RT^{4b::4}$	$\leftarrow t + u$

as2vewi rt,imma,immb



t	$\leftarrow$ RepLeftBit(I7a,32)
b	$\leftarrow$ RepLeftBit(I7b,8) & 0x03
u	$\leftarrow RT^{4b::4}$
$RT^{4b::4}$	$\leftarrow t + u$

The as2ve instructions were made available to the programmer through intrinsics of the form `a = spu_as2vebi(a,c,x)` where `a` is the vector operand. By using `a` as both input and output of the intrinsic, the compiler generally understands that the register is read and written by the instruction. However, in a certain case this did not happen and we had to use inline assembly as follows:

```
asm ("as2vebi %0,c,x"      \
        : "=r" (a)         \
        : "0" (a)          \
    );
```

The instructions were added to the simulator and the latency was set to three cycles. A possible implementation uses the FX2 (fixed point) pipeline and replicates the scalar to all SIMD slots of input a. The vector goes to input b of the ALU. Some additional control logic selects the proper SIMD slot to do addition while the others transfer input b. The latency of the FX2 pipeline is two cycles. Adding another cycle for the increase complexity results in three cycles.

4.2 Enhancing Saturation and Packing

The elementary data type to store pixel information is a byte. However, computation is done using halfwords (16 bits) in order not to loose precision. This introduces some overhead known as pack/unpack and saturation. In this section we propose three instruction classes that reduce the cost of this overhead.

4.2.1 Clip Instructions

Clip instructions saturate the elements of a vector between two values. A clip instruction can be used to emulate saturating arithmetic but also saturation between arbitrary boundaries. For example, in de deblocking filter kernel the index of a table lookup has to be clipped between 0 and 51. In the normal SPU architecture this operation is done using four instructions (see Listing 3). First, two masks are created by comparing the input vector with the lower and upper boundaries. Second, using the masks the result is selected from the input and the boundary vectors.

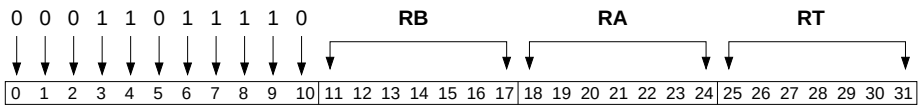
Listing 3: C code of a clip function using the normal SPU ISA. The clip instructions perform this operation in one step.

```
1 vsint16_t clip(vsint16_t a, vsint16_t min, vsint16_t max) {
2     vuint16_t min_mask, max_mask;
3     min_mask = spu_cmpgt(min, a);
4     max_mask = spu_cmpgt(a, max);
5     vsint16_t temp = spu_sel(a, min, min_mask);
6     return spu_sel(temp, max, max_mask);
7 }
```

The clip operation is mainly used in integer and short computations. The instruction could be implemented for bytes as well, but we do not expect any application to use it. The clip instructions fit the RR format if the source and

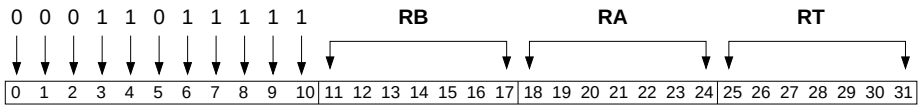
destination register (of the vector to clip) are the same. The instruction details are depicted below.

cliphw rt,ra,rb



for $j = 0$ to 7
  if $RT^{2j::2} > RB^{2j::2}$ then $RT^{2j::2} \leftarrow RB^{2j::2}$
  if $RT^{2j::2} < RA^{2j::2}$ then $RT^{2j::2} \leftarrow RA^{2j::2}$
end

clipw rt,ra,rb



for $j = 0$ to 3
  if $RT^{2j::4} > RB^{2j::4}$ then $RT^{2j::4} \leftarrow RB^{2j::4}$
  if $RT^{2j::4} < RA^{2j::4}$ then $RT^{2j::4} \leftarrow RA^{2j::4}$
end

The clip instructions were made available to the programmer through intrinsics of the form `a = spu_cliphw(a, min, max)` where the operands all have the same data type. We created intrinsics for both vectors and scalars. Note that the same instruction can be used for both.

Using these intrinsics to perform a saturation to 8 or 16-bit values, requires the programmer to define two variable containing the minimum and maximum value. To ease programming intrinsics of the form `a = spu_satub(a)` (saturate to unsigned byte value) can be created. This requires some small modifications to the machine description in the back-end of the compiler, such that it creates the two vectors containing the minimum and maximum values.

In certain cases the clip intrinsics suffered the same problem the as2ve intrinsics, namely that the compiler did not understand that the source and destination registers are the same. The solution is to use inline assembly as follows:

```
asm ("clipw %0,%1,%2"                          \
      : "=r" (a), "=r" (amin), "=r" (amax) \
      : "0" (a), "1" (amin), "2" (amax)    \
    );
```

The clip instructions were added to the simulator and the latency was set to four cycles. A normal compare operation takes two cycles. The two compares can be done in parallel but that requires that three quadwords are read from

the register file into the functional unit. Another approach would be to serialize the operation. First two operands are loaded, while these are compared and the result is selected the third operand is loaded, and finally the latter is compared to the result of the first compare. We added one cycle to account for loading three quadwords and also one cycle to account for the extra mux in the datapath that selects the correct value.

4.2.2 Sat_pack Instructions

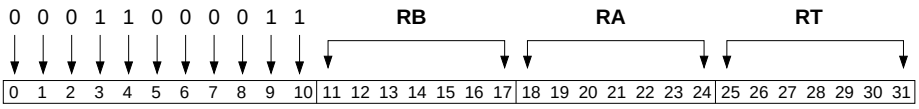
The sat_pack instructions takes two vectors of a larger data type, and saturates and packs them into one vector of a smaller data type. This operation is useful, for example, whenever data has been processed in signed 16-bit values but has to be stored to memory as unsigned 8-bit values. Listing 4 shows such a piece of code from the Luma kernel that can entirely be replaced by one sat_pack instruction. The kernel operates on 16×16 matrices (entire macroblocks). The vectors `va` and `vb` together contain one row of the output matrix.

Listing 4: Example code from the Luma kernel that can be replaced by one sat_pack instruction. The code saturates the vectors `va` and `vb` between 0 and 255, and packs all 16 elements into one vector of unsigned bytes.

```
1  vsint16_t va, vb, vt, sat;
2  vuint8_t vt_u8;
3  vuint8_t mask = {0x01,0x03,0x05,0x07,0x09,0x0B,0x0D,0x0F,\
4                   0x11,0x13,0x15,0x17,0x19,0x1B,0x1D,0x1F}
5  vsint16_t vzero = spu_splats(0);
6  vsint16_t vmax = spu_splats(255);
7  sat = spu_cmpgt(va,vzero);
8  va = spu_and(va,sat);
9  sat = spu_cmpgt(va,vmax);
10 va = spu_sel(va,vmax,sat);
11 sat = spu_cmpgt(vb,vzero);
12 vb = spu_and(vb,sat);
13 sat = spu_cmpgt(vb,vmax);
14 vb = spu_sel(vb,vmax,sat);
15 vt_u8 = spu_shuffle(va,vb,mask);
```

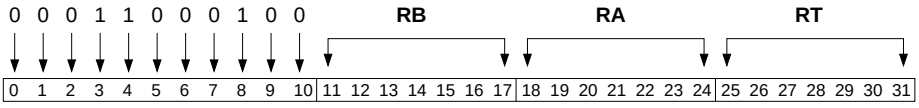
The sat_pack instructions fit the RR instruction format. For each type of operands sat_pack instructions could be implemented, but for H.264 decoding only packing from 16-bit signed to 8-bit unsigned is required. For audio processing packing from 32-bit signed to 16-bit signed can be useful. We implemented two different ways of packing. The `sp2x` (x=ub,hw) instruction packs the elements of the first and second operand to the first and second doubleword respectively. The `spil2x` instruction interleaves the elements of the two operands, i.e., the elements of the first operand go to the odd elements and those of the second operand to the even elements. The latter instruction is useful when the unpack was performed by a multiply instruction. The instruction details are depicted below.

sp2ub rt,ra,rb



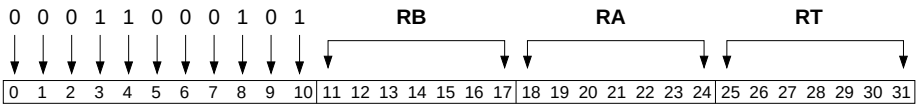
for $j = 0$ to 7
  r $\leftarrow RA^{2j::2}$
  if $r > 255$ then $r \leftarrow 255$
  if $r < 0$ then $r \leftarrow 0$
  $RT^{j} \leftarrow r_{0:7}$
  r $\leftarrow RB^{2j::2}$
  if $r > 255$ then $r \leftarrow 255$
  if $r < 0$ then $r \leftarrow 0$
  $RT^{j+8} \leftarrow r_{0:7}$
end

spil2ub rt,ra,rb



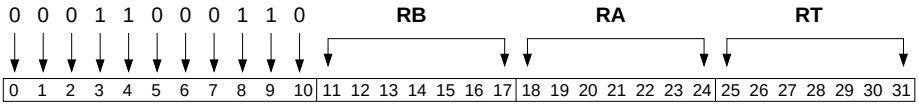
for $j = 0$ to 7
  r $\leftarrow RA^{2j::2}$
  if $r > 255$ then $r \leftarrow 255$
  if $r < 0$ then $r \leftarrow 0$
  $RT^{2*j+1} \leftarrow r_{0:7}$
  r $\leftarrow RB^{2j::2}$
  if $r > 255$ then $r \leftarrow 255$
  if $r < 0$ then $r \leftarrow 0$
  $RT^{2*j} \leftarrow r_{0:7}$
end

sp2hw rt,ra,rb



for $j = 0$ to 3
  t $\leftarrow RA^{4j::4}$
  if $t > 32767$ then $t \leftarrow 32767$
  if $t < -32768$ then $t \leftarrow -32768$
  $RT^{j::2} \leftarrow t_{0:15}$
  t $\leftarrow RB^{4j::4}$
  if $t > 32767$ then $t \leftarrow 32767$
  if $t < -32768$ then $t \leftarrow -32768$
  $RT^{j+8::2} \leftarrow t_{0:15}$
end

spil2hw rt,ra,rb



for $j = 0$ to 3
  t $\leftarrow RA^{4j::4}$
  if $t > 32767$ then $t \leftarrow 32767$
  if $t < -32768$ then $t \leftarrow -32768$
  $RT^{4*j+2::2} \leftarrow t_{0:15}$
  t $\leftarrow RB^{4j::4}$
  if $t > 32767$ then $t \leftarrow 32767$
  if $t < -32768$ then $t \leftarrow -32768$
  $RT^{4*j::2} \leftarrow t_{0:15}$
end

The sat_pack instructions were made available to the programmer through intrinsics and were added to the simulator. The latency was set to two cycles as the complexity of the instruction is similar to that of fixed point operations.

4.2.3 Asp Instructions

The sat_pack instructions pack two vectors to one, which means that the two vectors are placed in consecutive memory locations. However, the data is not in all cases stored in memory in this way. For those situations the asp (Add, Saturate, and Pack) instructions come in hand which combine an addition, a saturation, and a pack.

Listing 5: Example code from the IDCT8 kernel that can be replaced by one asp instruction. The code adds the vectors `va` and `vb`, saturates them between 0 and 255, and packs the elements to 8-bit values.

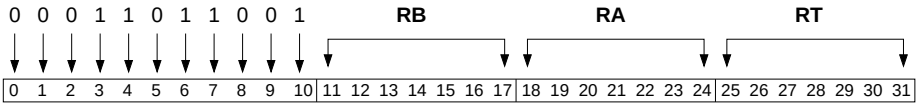
```
1  vsint16_t va, vb, vt, sat;
2  vuint8_t vt_u8;
3  const vuint8_t packu16 =
4                    {0x01,0x03,0x05,0x07,0x09,0x0B,0x0D,0x0F,\
5                     0x11,0x13,0x15,0x17,0x19,0x1B,0x1D,0x1F}
6  vsint16_t vzero = spu_splats(0);
7  vsint16_t vmax = spu_splats(255);
8  vt = spu_add(va,vb);
9  sat = spu_cmpgt(vt,vzero);
10 vt = spu_and(vt,sat);
11 sat = spu_cmpgt(vt,vmax);
12 vt = spu_sel(vt,vmax,sat);
13 vt_u8 = spu_shuffle(vt,vzero,packu16);
```

Listing 5 shows a piece of code from the IDCT kernel that entirely can be replaced by one asp instruction. The code adds the result of the IDCT to the predicted picture. First it adds two vectors of signed halfwords. Second, the elements of the vector are saturated between 0 and 255. Finally, the elements are packed to bytes and stored in the first half of the vector. Using the normal SPU ISA many instructions and auxiliary variables are needed. The asp instructions are tailored for this kind of operation and can perform the entire operation at once. Note, that if only a saturate and a pack is required, one of the operands can be assigned a zero vector.

The asp instructions fit the RR format. For each type of operands an asp instruction could be implemented, but we found it only necessary to implement one: `asp2ub` (add, saturate, and pack to unsigned bytes). It takes as input two vectors of signed halfwords and stores the output in a vector of unsigned bytes. The eight elements are stored in the first eight slots while the last eight slots are assigned value zero. The instruction details are depicted below.

The asp instruction was made available to the programmer through an intrinsic. The instruction was added to the simulator and the latency was set to four cycles. A normal addition costs two cycles and we added two more cycles for the saturation and the data rearrangement.

asp2ub rt,ra,rb



for $j = 0$ to 7
r $\leftarrow RA^{2j::2} + RB^{2j::2}$
if $r > 255$ then $r \leftarrow 255$
if $r < 0$ then $r \leftarrow 0$
$RT^j \leftarrow r_{0:7}$
end

4.3 Enhancing Matrix Transposes

Matrix Transposition (MT) is at the heart of many media algorithms and can take up a large part of the execution time. Typically, multimedia data is organized in blocks (matrices) of 16×16, 8×8, and sometimes smaller. This allows utilization of SIMD instructions to exploit DLP. However, often computations have to be performed row wise and column wise. Thus matrix transposes are required to arrange the data for SIMD processing.

In this section we show how fast matrix transposition at low cost can be achieved using swapoe (swap odd-even) instructions. But first we review existing methodologies for matrix transposition using SIMD.

4.3.1 Conventional SIMD Matrix Transposition

The conventional way of doing MT with SIMD instructions is performing a series of permutations. We review an 8×8 transpose in AltiVec as an example. Altivec has 128-bit SIMD registers and thus we assume a matrix of halfwords, stored in eight registers (`a0` through `a7`). The code in Listing 6 performs the MT and stores the result in registers (`b0` through `b7`).

The instruction `vec_mergeh(ra, rb)` functions as follows. Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the elements in the most significant 8 bytes of `ra`. The odd-numbered elements of the result are taken, in order, from the elements in the most significant 8 bytes of `rb`. The instruction `vec_mergel(ra, rb)` functions similar but takes the elements from the least significant 8 bytes.

The AltiVec MT requires 24 instructions for an 8×8 matrix. In general this approach requires $\log n$ steps of n instructions. Thus in total $n \log n$ instructions are required. For $n = 16$ 64 AltiVec or SPU instructions are required.

4.3.2 Matrix Transposition using swapoe instructions

In this section we present the swapoe instructions and explain how they speedup matrix transposition. The swapoe instructions are similar to the mix instructions introduced by Lee [12] and used in HP's MAX-2 multimedia extension to the PA-RISC architecture. Although the mix instructions found their way

Listing 6: An 8×8 matrix transpose using Altivec instructions.

```
1  TRANSPOSE_8_ALTIVEC(a0,a1,a2,a3,a4,a5,a6,a7,\
2                      b0,b1,b2,b3,b4,b5,b6,b7){
3      b0 = vec_mergeh( a0, a4 );
4      b1 = vec_mergel( a0, a4 );
5      b2 = vec_mergeh( a1, a5 );
6      b3 = vec_mergel( a1, a5 );
7      b4 = vec_mergeh( a2, a6 );
8      b5 = vec_mergel( a2, a6 );
9      b6 = vec_mergeh( a3, a7 );
10     b7 = vec_mergel( a3, a7 );
11     a0 = vec_mergeh( b0, b4 );
12     a1 = vec_mergel( b0, b4 );
13     a2 = vec_mergeh( b1, b5 );
14     a3 = vec_mergel( b1, b5 );
15     a4 = vec_mergeh( b2, b6 );
16     a5 = vec_mergel( b2, b6 );
17     a6 = vec_mergeh( b3, b7 );
18     a7 = vec_mergel( b3, b7 );
19     b0 = vec_mergeh( a0, a4 );
20     b1 = vec_mergel( a0, a4 );
21     b2 = vec_mergeh( a1, a5 );
22     b3 = vec_mergel( a1, a5 );
23     b4 = vec_mergeh( a2, a6 );
24     b5 = vec_mergel( a2, a6 );
25     b6 = vec_mergeh( a3, a7 );
26     b7 = vec_mergel( a3, a7 );
27 }
```

into the IA-64 architecture because of the collaboration between HP and Intel, no modern multimedia extension incorporates it. We rediscover the swapoe instructions and show how it is valuable for nowadays media applications.

The swapoe instructions are inspired by Eklundh's recursive matrix transposition algorithm [13], which is illustrated in 4 for an 8×8 matrix. The new instructions have two source registers and two destination registers. In this report we assume that the source and the destination registers are the same, but if allowed by the instruction format, they can also be different. Using the same registers for source and destination has the advantage that the transpose can be performed in-place. The complete set of swapoe instructions is depicted in Figure 5.

The instructions swap the odd elements of register `ra` with the even elements of `rb`. Figure 6 shows the `swapoehw` instruction as example. On top the two registers before the instruction is executed are depicted. The bottom of the figure shows the contents of the register after the `swapoehw`. The other instructions operate similar but on different element sizes.

The swapoe instructions directly implement the steps of Eklundh's matrix transpose algorithm. The 8×8 MT function using swapoe is shown in Listing 7

This implementation requires 12 instructions. Note that this method performs the MT in-place, i.e., the destination registers are the source registers and

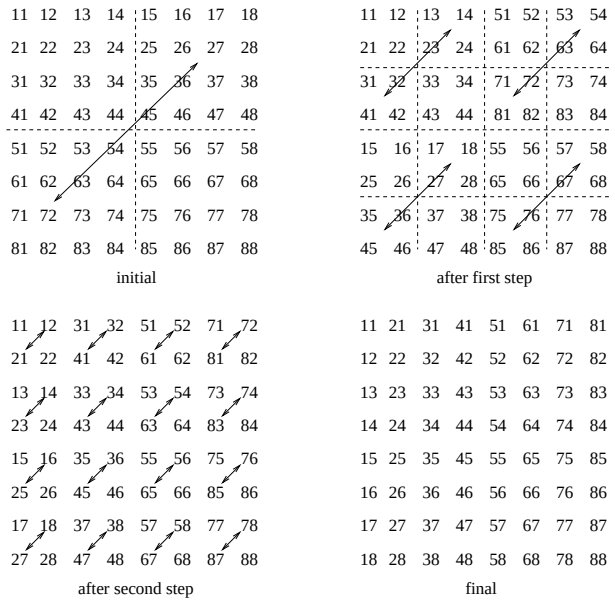


Fig. 4: Eklundh's matrix transpose algorithm.

```
swapoedw  ra,rb          \\Swap doublewords
swapoew   ra,rb          \\Swap words
swapoehw  ra,rb          \\Swap halfwords
swapoeb   ra,rb          \\Swap bytes
```

Fig. 5: Overview of the swapoe instructions.

no additional registers are required for temporal storage. Depending on the size of the register file and the kernel the MT is used in, this might save a lot of loads and stores and further increase performance.

In general for an $n \times n$ matrix each step can be accomplished using $n/2$ instructions. Since there are $\log n$ stages, the total number of instructions is $(n \log n)/2$, assuming the matrix is already loaded in the register file. Thus, when $n = 16$ 32 instructions are required. Indeed our approach also requires half the number of instructions compared to the AltiVec implementation.

4.3.3 Implementation Details

The swapoe instructions have two register operands and thus fit the RR instruction format of the SPU [11], where the `RB` slot is not used. Below the details of all swapoe instructions are provided.

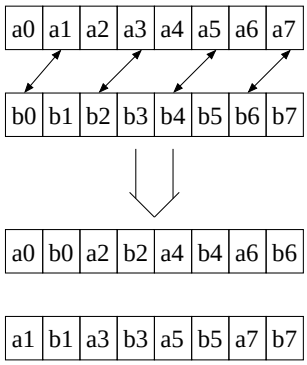
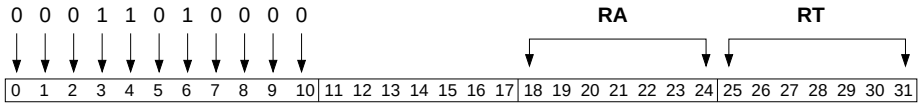


Fig. 6: Graphical representation of the `swapoehw` instruction. The odd elements of the `a` register are swapped with the even elements of the textttb register.

Listing 7: An 8×8 matrix transpose using the swapoe instructions.

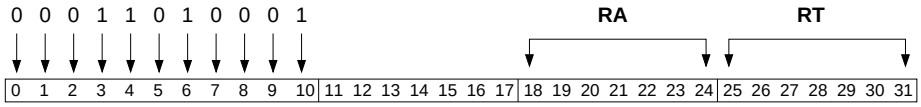
```
1  TRANSPOSE_8_SWAPOE(a0, a1, a2, a3, a4, a5, a6, a7) {
2      swapoedw(a0, a4);
3      swapoedw(a1, a5);
4      swapoedw(a2, a6);
5      swapoedw(a3, a7);
6      swapoew(a0, a2);
7      swapoew(a1, a3);
8      swapoew(a4, a6);
9      swapoew(a5, a7);
10     swapoehw(a0, a1);
11     swapoehw(a2, a3);
12     swapoehw(a4, a5);
13     swapoehw(a6, a7);
14 }
```

swapoedw rt,ra



$$
\begin{aligned}
w &\leftarrow RT^{8:15} \\
RT^{8:15} &\leftarrow RA^{0:7} \\
RA^{0:7} &\leftarrow w
\end{aligned}
$$

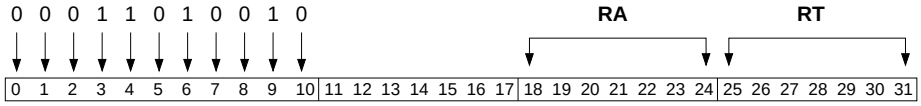
swapoew rt,ra



$t \leftarrow RT^{4:7}$
$RT^{4:7} \leftarrow RA^{0:3}$
$RA^{0:3} \leftarrow t$

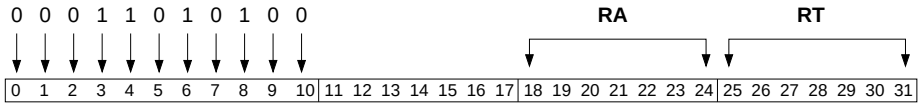
$t \leftarrow RT^{12:15}$
$RT^{12:15} \leftarrow RA^{8:11}$
$RA^{8:11} \leftarrow t$

swapoehw rt,ra



for $j = 0$ to 15 by 4
    $r \leftarrow RT^{j+2::2}$
    $RT^{j+2::2} \leftarrow RA^{j::2}$
    $RA^{j::2} \leftarrow r$
end

swapoeb rt,ra



for $j = 0$ to 15 by 2
    $b \leftarrow RT^{2+1}$
    $RT^{j+1} \leftarrow RA^{2}$
    $RA^{j} \leftarrow b$
end

All swapoe instructions were made accessible from C code through intrinsics. However, using the intrinsic can cause problems due to the following. The SPU compiler assumes that for an instruction `mnemonic rt, ra, [rb/rc/im]` only register `rt` is modified. As a consequence, the compiler assumes the variable located in register `ra` is also available in memory, and might decide to re-use the register for another variable. To avoid this problem inline assembly was used, which allows the programmer to specify the mode of each register:

```
asm volatile ("swapoedw %0,%1"    \
        : "=r" (a0), "=r" (a4)    \
        : "0" (a0), "1" (a4)      \
        );
```

Note that we used the keyword `volatile` here. It was not required to maintain correct order of execution. We used it to enforce the compiler to maintain the hand optimized order of instructions. Simulations confirmed that indeed the hand optimized order of these swapoe instruction resulted in faster execution compared to compiler scheduled swapoe instructions.

The swapoe instructions were added to the simulator and to the compiler. We assume a latency of three clock cycles for the predefined permutations of the swapoe instructions. However, the instructions write to two registers instead of one. Writing to two registers can be serialized if only one write-port is available. The SPU register file probably has two write-ports as it has an odd and even pipeline. Thus, the swapoe instructions could as well use both these write ports. However, this might cause a stall in the other pipeline. We account for the second write, no matter how it is implemented, by adding one extra cycle. So in total the latency of the swapoe instructions was set to four clock cycles.

4.4 Enhancing Arithmetic Operations

In this section we propose several instruction classes that enhance the execution of regular arithmetic expressions in video decoding.

4.4.1 Sfxsh Instructions

Sfxsh (Simple FiXed point & SHift) instructions combine simple fixed point operations with a shift operation. Listing 8 shows an example from the IDCT8 kernel where one operand is shifted and added to the second. Using a normal ISA these two lines of code would require four instructions.

Listing 8: Example of how the sfxsh instructions speedup code of the IDCT kernel. Left are two lines of C code, in the middle is the normal assembly code, while the right depicts the assembly code using sfxsh instructions.

```
1  b1 = a7>>2 + a1;     rotmahi $11,$9,-2   shaddhw   $11,$11,$9,2
2  b3 = a3 + a5>>2;     ah      $11,$11,$4  shaddhw   $8,$8,$15,2
3                       rotmahi $8,$15,-2
4                       ah      $8,$8,$7
```

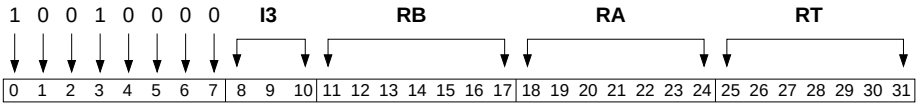
This kind of code is found frequently in the kernels we investigated. More general, they can be characterized as follows. There are two register operands, a simple arithmetic operation (like addition and subtraction), and one shift right by a small fixed amount of bits. Our analysis showed that the instructions depicted in Figure 7 are beneficial. Our benchmarks only used the instructions that operate on vectors of halfwords, but we provide the same operations for vectors of words as well. To prevent overflow, computation is generally not performed using bytes but halfwords. Thus the sfxsh instructions for bytes does not seem to be useful.

```
shaddhw  rt,ra,rb,imm  \\rt = ra + rb>>imm      (for vector of halfwords)
shsubhw  rt,ra,rb,imm  \\rt = ra - rb>>imm      (for vector of halfwords)
addshhw  rt,ra,rb,imm  \\rt = (ra + rb)>>imm    (for vector of halfwords)
subshhw  rt,ra,rb,imm  \\rt = (ra - rb)>>imm    (for vector of halfwords)
shaddw   rt,ra,rb,imm  \\rt = ra + rb>>imm      (for vector of words)
shsubw   rt,ra,rb,imm  \\rt = ra - rb>>imm      (for vector of words)
addshw   rt,ra,rb,imm  \\rt = (ra + rb)>>imm    (for vector of words)
subshw   rt,ra,rb,imm  \\rt = (ra - rb)>>imm    (for vector of words)
```

Fig. 7: Overview of the sfxsh instructions.

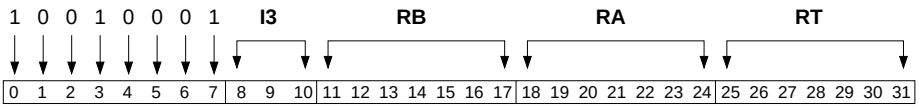
The sfxsh instructions could fit the RRR format of the SPU ISA if the immediate value would be stored in a register. However, this approach would cause an unnecessary load to a register. Furthermore, there are not enough opcodes of the RRR format available to implement all instructions. It is the case, though, that the shift operation is only performed by a few bits, i.e., the shift count is between 0 and 7. Thus, the immediate value can be represented by 3 bits only. Using this observation we defined a new instruction format RRI3 that uses 8 bits for the opcode, 3 bits for the immediate value, and 7 bits for each of the three register operands. The instruction details, depicted below, show the RRI3 format graphically.

shaddhw rt,ra,rb,imm



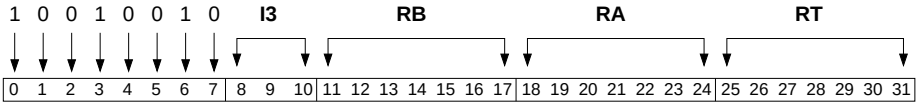
s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 2
    p ← $RT^{j::2}$
    for $b = 0$ to 15
        if $(b - s) > 0$ then $r_b \leftarrow p_{b-s}$
        else $r_b \leftarrow 0$
    end
    $RB^{j::2} \leftarrow RA^{j::2} + r$
end

shsubhw rt,ra,rb,imm



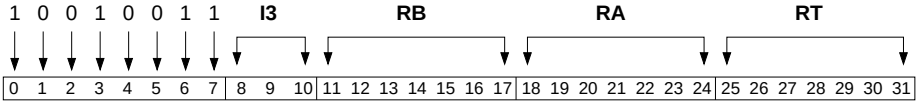
s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 2
    p ← $RT^{j::2}$
    for $b = 0$ to 15
        if $(b - s) > 0$ then $r_b \leftarrow p_{b-s}$
        else $r_b \leftarrow 0$
    end
    $RB^{j::2} \leftarrow RA^{j::2} - r$
end

addshhw rt,ra,rb,imm



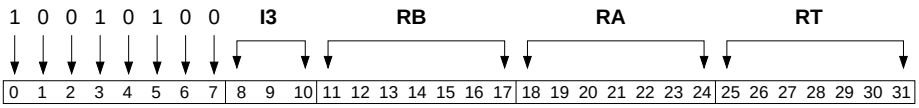
s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 2
    p ← $RA^{j::2} + RT^{j::2}$
    for $b = 0$ to 15
        if $(b - s) > 0$ then $r_b \leftarrow p_{b-s}$
        else $r_b \leftarrow 0$
    end
    $RB^{j::2} \leftarrow r$
end

subshhw rt,ra,rb,imm



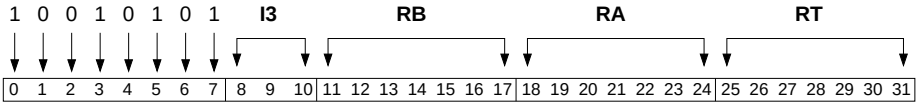
s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 2
    p ← $RA^{j::2} - RT^{j::2}$
    for $b = 0$ to 15
        if $(b - s) > 0$ then $r_b \leftarrow p_{b-s}$
        else $r_b \leftarrow 0$
    end
    $RB^{j::2} \leftarrow r$
end

shaddw rt,ra,rb,imm



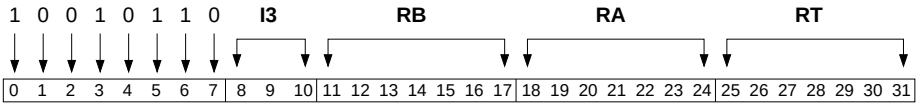
s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 4
    t ← $RT^{j::4}$
    for $b = 0$ to 31
        if $(b - s) > 0$ then $u_b \leftarrow t_{b-s}$
        else $u_b \leftarrow 0$
    end
    $RB^{j::4} \leftarrow RA^{j::4} + u$
end

shsubw rt,ra,rb,imm



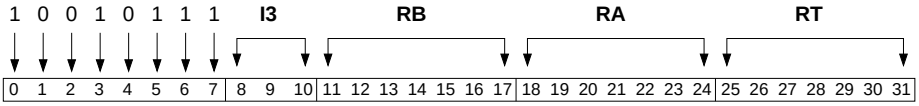
s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 4
    t ← $RT^{j::4}$
    for $b = 0$ to 31
        if $(b - s) > 0$ then $u_b \leftarrow t_{b-s}$
        else $u_b \leftarrow 0$
    end
    $RB^{j::4} \leftarrow RA^{j::4} - u$
end

addshw rt,ra,rb,imm



s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 4
    t ← $RA^{j::4} + RT^{j::4}$
    for $b = 0$ to 31
        if $(b - s) > 0$ then $u_b \leftarrow t_{b-s}$
        else $u_b \leftarrow 0$
    end
    $RB^{j::4} \leftarrow u$
end

subshw rt,ra,rb,imm



s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 4
    t ← $RA^{j::4} - RT^{j::4}$
    for $b = 0$ to 31
        if $(b - s) > 0$ then $u_b \leftarrow t_{b-s}$
        else $u_b \leftarrow 0$
    end
    $RB^{j::4} \leftarrow u$
end

The sfxsh instructions were made available to the programmer through intrinsics. We did not modify the compiler to handle the new instruction format. Instead we used a little trick at the programmer's side, such that the compiler could handle the sfxsh instructions as if they were of the RRR format. Inside the compiler we created the instruction `rrrx rt,ra,rb,imm` using the RRR format except that we replaced `rc` with an immediate (see Figure 8). The opcode is 4 bits and has value `0x480` (`1001`). This instruction is available through the intrinsic `d = spu_rrrx(a,b,imm)`. To create the instruction word corresponding to the sfxsh instructions we created the defines of Listing 9, that add an offset to the immediate value. The offset will change bits 4 to 7 of the instruction word such that the right RRI3 opcode is created.

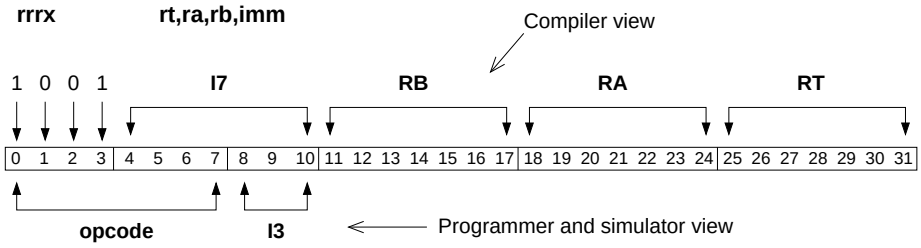


Fig. 8: The `rrrx` instruction of format RRR was used inside the compiler to handle the sfxsh instructions with format RRI3. At the programmers side an offset was added to the immediate value to set bits 4 to 7.

Listing 9: Defines translate the sfxsh intrinsics to the rrrx intrinsic and add the offset to the immediate value.

```
1 #define spu_shaddhw(a,b,c)     spu_rrrx(a,b,c);
2 #define spu_shsubhw(a,b,c)     spu_rrrx(a,b,c+8);
3 #define spu_addshhw(a,b,c)     spu_rrrx(a,b,c+16);
4 #define spu_subshhw(a,b,c)     spu_rrrx(a,b,c+24);
5 #define spu_shaddw(a,b,c)      spu_rrrx(a,b,c+32);
6 #define spu_shsubw(a,b,c)      spu_rrrx(a,b,c+40);
7 #define spu_addshw(a,b,c)      spu_rrrx(a,b,c+48);
8 #define spu_subshw(a,b,c)      spu_rrrx(a,b,c+56);
```

The simulator was modified to recognize the new RRI3 instruction format and the new instructions. The latency of the sfxsh instructions was set to five cycles. Normal shift and rotate instructions take four cycles. We added one cycle to this for the extra addition or subtraction.

4.4.2 Mpytr Instructions

The Mpytr (multiply and truncate) instructions perform a multiply or multiply-add on vectors of halfwords and stores the results as halfwords as well by truncating the most significant bits. The normal multiply instructions in the SPU ISA take halfword operands as input and produce word values. The reason is

of course not to loose precision, but the side effect is that only four SIMD slots are used while the input vector consist of eight elements.

In video decoding the primary input and output data type is unsigned byte. In order not to loose precision, computations are mostly performed in signed halfword values. In the Luma and Chroma interpolation filters a few multiplications are performed. Using the normal ISA the intermediate results are words (32 bits) while we know from the algorithm that the maximum possible value can be represented by 15 bits. Using the mpytr instructions, the conversion to 32-bit values is avoided.

Listing 10: Example code from the Luma kernel that can be replaced by one mpytr instruction. In the code, first the odd elements of `input` are multiplied by constant 5, followed by the even elements. A shuffle packs the two vectors of words into one vector of halfwords.

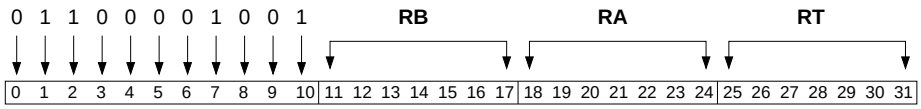
```
1  vsint16_t v5 = spu_splats(5);
2  vuint8_t mask = {0x02,0x03,0x12,0x13,0x06,0x07,0x16,0x17,\
3                   0x0A,0x0B,0x1A,0x1B,0x0E,0x0F,0x1E,0x1F};
4  vsint32_t A = spu_mulo(input, 5);
5  vsint32_t B = spu_mule(input, v5);
6  vsint16_t result = spu_shuffle(B, A, mask);
```

Listing 10 shows a piece of code from the Luma kernel that multiplies the vector `input`, containing eight signed halfwords, with a constant 5. First the odd elements are multiplied (`spu_mulo`) followed by the even elements (`spu_mule`). Using shuffle the two vectors of words are packed into one vector of halfwords. Furthermore, there is some overhead involved to create a vector of constants and to load the mask. This overhead though is performed once per function call and thus shared among a few multiplications.

The code of Listing 10 can be replaced by one `mpytrhwi` instruction, which multiplies a vector of signed halfwords by an unsigned immediate value. The other instructions in this class are `mpytrhw` (multiply two vectors of signed halfwords) and `maddtrhw` (multiply-add two vectors of signed halfwords). We do not expect mpytr instructions for unsigned halfwords to be useful as most media computations are done using signed values. Mpytr instructions for bytes do not seem useful as unpacking to 16-bit is generally necessary and thus normal byte multiplication instructions can be used (see next section). Mpytr instructions for words might be useful in certain cases. As we are not aware of any, we did not implement those instructions. Further investigation of applications, for example audio processing, would be useful in this respect.

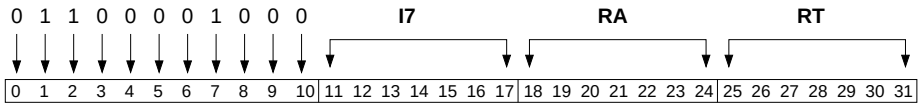
The `mpytrhw` instructions has the RR format and the `mpytrhwi` has the RI7 format. Thus the latter has seven bits for the immediate value which is enough for video decoding. The RI10 format could be used as well in case more bits are necessary for the immediate value. The `maddtrhw` instruction has the RRR format. The instruction details are depicted below.

mpytrhw rt,ra,rb



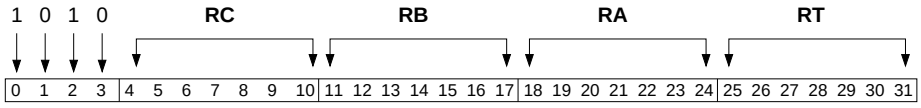
for $j = 0$ to 7
$\quad t \leftarrow RA^{2j::2} * RB^{2j::2}$
$\quad RT^{2j::2} \leftarrow t_{0:15}$
end

mpytrhwi rt,ra,imm



for $j = 0$ to 7
$\quad t \leftarrow RA^{2j::2} * I7$
$\quad RT^{2j::2} \leftarrow t_{0:15}$
end

maddtrhw rt,ra,rb,rc



for $j = 0$ to 7
$\quad t \leftarrow RA^{2j::2} * RB^{2j::2} + RC^{2j::2}$
$\quad RT^{2j::2} \leftarrow t_{0:15}$
end

The mpytr instructions were made available to the programmer through intrinsics and were added to the simulator. All regular multiply instructions are executed in the FP7 pipeline which has a latency of seven cycles. The mpytr instructions are no more complex than the other multiply instructions and thus the latency of those was set to seven cycles.

4.4.3 Mpy_byte Instructions

The mpy_byte instructions perform a multiply on vectors of unsigned byte elements and produces vectors of unsigned halfwords. These instructions are not novel at all, but the SPU ISA did not contain any byte multiplication instructions although they are useful for many media applications.

In video decoding typically the byte input data is first unpacked to halfwords, after which the arithmetic computations are performed. In the previous section we showed how the mpytr instructions can prevent the overhead of going to words when performing a multiplication. Another way to resolve this issue is to do multiplications on the byte values directly and get the unpack to halfwords for free.

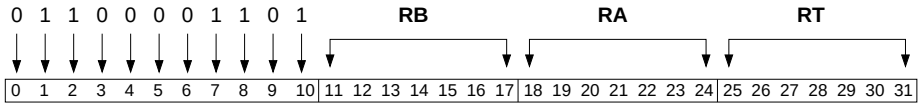
As byte multiplication instructions are not part of the normal SPU ISA we implemented them. The `mpyubi` and `mpyhhubi` instructions multiply a vector of unsigned bytes with an immediate value and store the results as unsigned halfwords. The first instruction operates on the odd elements of the input vector while the latter operates on the even elements. These instructions fit the RI7 format. Although not used in the benchmarks, we also added the `mpyub` and `mpyhhub` instructions to the ISA. These instructions are the same as the previous ones, but take two registers operands as input.

The `maddub` and `msubub` perform a multiply-add and multiply-sub operation on vectors of unsigned bytes and store the result as a vector of unsigned halfwords. These instruction operate on the odd elements of the input vectors. These two instructions fit the RRR format. Only a few opcodes of the RRR format are available and in the normal SPU ISA only two were left unused. However, we already used these opcodes for the `rrrx` and `maddtrhw` instructions. To be able to use the `maddub` and `msubub` anyway, we removed two floating point instructions from the ISA. For a media accelerator these floating point instructions are not needed and thus this choice is justified. If a more general purpose core is targeted, the `msubub` is most likely not worth removing another instruction for. Simulation results showed that this instruction provides very limited benefit compared to others.

We also implemented the `mpyhhaub` which multiply-adds the even elements of vectors of bytes. Similar to the existing `mpyhha` instruction it has the RR format and thus writes the result to the register of the first operand. We expected this instruction to be a good addition to the `maddub` instruction. However, simulation results showed no speedup using this instruction compared to using a shuffle and a normal multiply-add. Therefore we did not include it in the accelerator architecture. For the same reason we omitted the `mpyhhsub` (multiply high high and sub unsigned bytes) instruction.

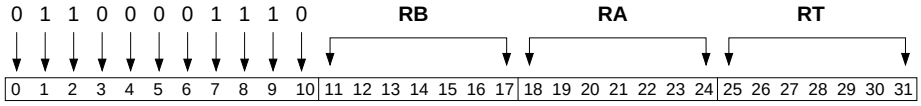
We did not implement any multiplication instructions for signed bytes. We are not aware of any application using this data type. The details of the new byte multiplication instructions that we implemented are depicted below.

mpyub rt,ra,imm



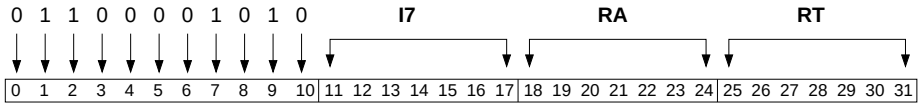
for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j+1} \mid * \mid RB^{2j+1}$
end

mpyhhub rt,ra,imm



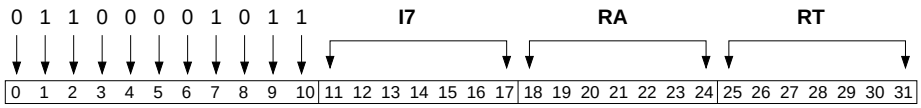
for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j} \mid * \mid RB^{2j}$
end

mpyubi rt,ra,imm



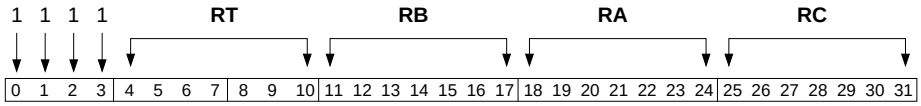
for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j+1} \mid * \mid I7$
end

mpyhhubi rt,ra,imm



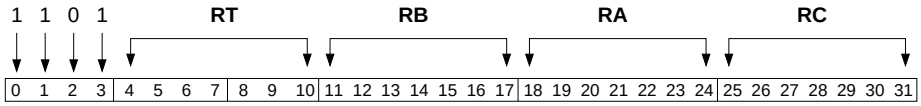
for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j} \mid * \mid I7$
end

maddub rt,ra,rb,rc



for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j+1} \mid * \mid RB^{2j+1} + RC^{2j+1}$
end

msubub rt,ra,rb,rc



for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j+1} \mid * \mid RB^{2j+1} - RC^{2j+1}$
end

The mpy_byte instructions were made available to the programmer through intrinsics and were added to the simulator. Just as the mpytr instruction, the mpy_byte instructions are no more complex than the other multiply instructions and thus the latency of those was set to seven cycles.

4.4.4 IDCT8 Instruction

The Inverse Discrete Cosine Transform (IDCT) is an operation often found in picture and movie processing. The IDCT is typically performed on 8×8 or 4×4 pixel blocks. Performing the IDCT comprises a one dimensional IDCT row wise followed by a one dimensional IDCT column wise. FFmpeg uses the LLM IDCT algorithm [14]. The computational structure of the one dimensional IDCT8 is depicted in Figure 9. If elements 0 through 7 are a row of the matrix, then this diagram represents a row wise one dimensional IDCT8. The diagram also shows that the IDCT consists of simple additions, subtractions, and some shifts.

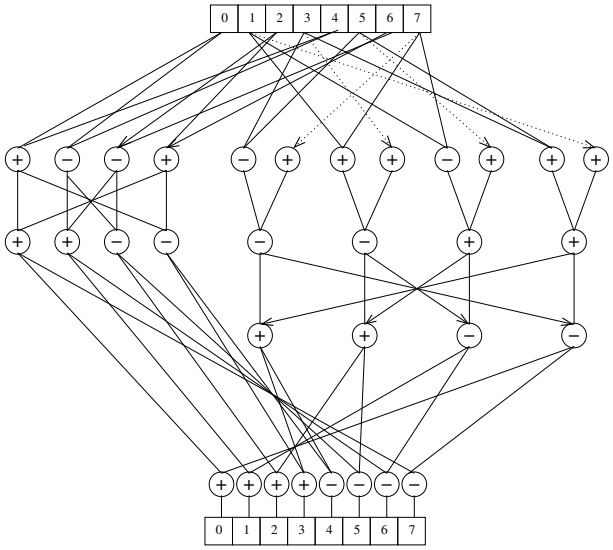


Fig. 9: Computational structure of the one dimensional IDCT8. Arrows indicate a shift right by one bit. A dotted line means the two operands are the same.

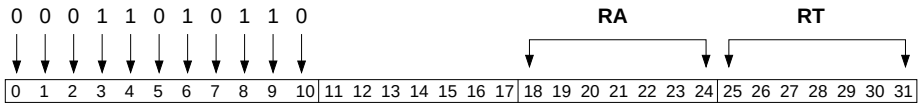
The typical way to SIMDimize this one dimensional IDCT is to perform the operations on vectors of elements. Thus for the row wise 1D IDCT a vector would be a column of the matrix, while for the column wise 1D IDCT a vector would be a row of the matrix. Assuming the sub-block is located in memory row major (row elements in consecutive addresses), the whole IDCT is performed in four steps: matrix transpose (MT); row wise 1D IDCT; MT; column wise 1D IDCT.

We propose the `IDCT8` instruction that performs the row wise 1D IDCT8. This intra-vector instruction takes one row of the matrix as input, perform the arithmetic operations as depicted in Figure 9, and stores the result in a register.

Thus, for an 8×8 matrix the total row wise 1D IDCT8 takes eight instructions. Besides the reduction in the number of executed instructions, the two MTs are also avoided. This is because using the `IDCT8` instruction the whole IDCT can be performed in two steps: row wise 1D IDCT using the `IDCT8` instruction; conventional column wise 1D IDCT.

The `IDCT8` instruction fits the RR format where the RB slot is not used. Register RA is the input register while RT is the output register. Usually for the IDCT8 these two would be the same but the instruction allows them to be different. The `IDCT8` instruction assumes its operand to be a vector of signed halfwords. The instruction details are depicted below.

idct8 rt,ra



$$
\begin{array}{lllll}
r0 & \leftarrow RA^{0:1} & +RA^{8:9} & & \\
r2 & \leftarrow RA^{0:1} & -RA^{8:9} & & \\
r4 & \leftarrow RA^{2:3}/2 & -RA^{12:13} & & \\
r6 & \leftarrow RA^{2:3} & +RA^{12:13}/2 & & \\
 & & & & \\
s0 & \leftarrow r0 + r6 & & & \\
s2 & \leftarrow r2 + r4 & & & \\
s4 & \leftarrow r2 - r4 & & & \\
s6 & \leftarrow r0 - r6 & & & \\
 & & & & \\
r1 & \leftarrow RA^{10:11} & -RA^{6:7} & -(RA^{14:15} & +RA^{14:15}/2) \\
r3 & \leftarrow RA^{2:3} & +RA^{14:15} & -(RA^{6:7} & +RA^{6:7}/2) \\
r5 & \leftarrow RA^{14:15} & -RA^{2:3} & +(RA^{10:11} & +RA^{10:11}/2) \\
r7 & \leftarrow RA^{6:7} & +RA^{10:11} & +(RA^{2:3} & +RA^{2:3}/2) \\
 & & & & \\
s1 & \leftarrow r7/4 + r1 & & & \\
s3 & \leftarrow r3 + r5/4 & & & \\
s5 & \leftarrow r3/4 - r5 & & & \\
s7 & \leftarrow r7 - r1/4 & & & \\
 & & & & \\
RT^{0:1} & \leftarrow s0 + s7 & & & \\
RT^{2:3} & \leftarrow s2 + s5 & & & \\
RT^{4:5} & \leftarrow s4 + s3 & & & \\
RT^{6:7} & \leftarrow s6 + s1 & & & \\
RT^{8:9} & \leftarrow s6 - s1 & & & \\
RT^{10:11} & \leftarrow s4 - s3 & & & \\
RT^{12:13} & \leftarrow s2 - s5 & & & \\
RT^{14:15} & \leftarrow s0 - s7 & & &
\end{array}
$$

The `IDCT8` instruction was made available to the programmer through an intrinsic. The instruction was added to the simulator and the latency was set to eight cycles for the following reason. Fixed point operations take two cycles in the SPU architecture. The `IDCT8` instruction performs four steps of simple fixed point operations and thus the latency of the new instruction is at most four times two making a total of eight cycles.

4.4.5 IDCT4 Instruction

The IDCT4 is similar to the IDCT8 but operates on a 4×4 pixel block. The computational structure of the one dimensional IDCT4 is simpler than that of the IDCT8 as shown in Figure 10.

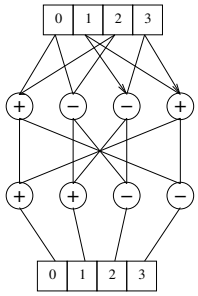
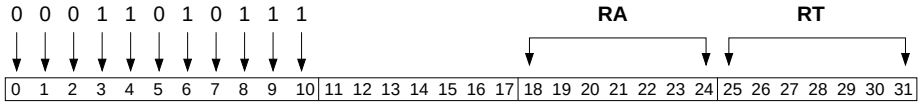


Fig. 10: Computational structure of the one dimensional IDCT4. An arrow indicates a shift right by one bit.

For the IDCT4 kernel a similar approach could be used as for the IDCT8. However, as a 4×4 matrix of halfwords completely fits in two registers of 128-bits, the entire two dimensional IDCT4 could be performed using one instruction with two operands. Such instruction would read the matrix from its two operand registers, perform the arithmetic operations, and store the result back in the two registers. Storing the result could be done in one cycle if the the register file has two write ports, or otherwise two cycles are required.

The `IDCT4` instructions fits the RR format where the RB slot is not used. Register RA is the input and output register that contains the first two rows of the matrix while RT holds the last two rows. The IDCT4 instruction assumes its operands to be vectors of signed halfwords. The instruction details are depicted below.

idct4 rt,ra



for $i = 0$ to 8 by 8

$$
\begin{array}{lll}
r0 & \leftarrow RA^{i::2} & +RA^{(i+4)::2} \\
r1 & \leftarrow RA^{i::2} & -RA^{(i+4)::2} \\
r2 & \leftarrow (RA^{(i+2)::2} >> 1) & -RA^{(i+6)::2} \\
r3 & \leftarrow RA^{(i+2)::2} & +(RA^{(i+6)::2} >> 1) \\
 & & \\
Q^{i::2} & \leftarrow r0 & +r3 \\
Q^{(i+2)::2} & \leftarrow r1 & +r2 \\
Q^{(i+4)::2} & \leftarrow r1 & -r2 \\
Q^{(i+6)::2} & \leftarrow r0 & -r3 \\
 & & \\
r0 & \leftarrow RT^{i::2} & +RT^{(i+4)::2} \\
r1 & \leftarrow RT^{i::2} & -RT^{(i+4)::2} \\
r2 & \leftarrow (RT^{(i+2)::2} >> 1) & -RT^{(i+6)::2} \\
r3 & \leftarrow RT^{(i+2)::2} & +(RT^{(i+6)::2} >> 1) \\
 & & \\
R^{i::2} & \leftarrow r0 & +r3 \\
R^{(i+2)::2} & \leftarrow r1 & +r2 \\
R^{(i+4)::2} & \leftarrow r1 & -r2 \\
R^{(i+6)::2} & \leftarrow r0 & -r3
\end{array}
$$

end

for $i = 0$ to 3

$$
\begin{array}{ll}
r0 & \leftarrow Q^{2i::2} + R^{2i::2} \\
r1 & \leftarrow Q^{2i::2} - R^{2i::2} \\
r2 & \leftarrow (Q^{2(i+4)::2} >> 1) - R^{2(i+4)::2} \\
r3 & \leftarrow Q^{2(i+4)::2} + (R^{2(i+4)::2} >> 1) \\
 & \\
RA^{2i::2} & \leftarrow ((r0 + r3) >> 6) \\
RA^{2(i+4)::2} & \leftarrow ((r1 + r2) >> 6) \\
RT^{2i::2} & \leftarrow ((r1 - r2) >> 6) \\
RA^{2(i+4)::2} & \leftarrow ((r0 - r3) >> 6)
\end{array}
$$

end

As the instruction details show, the result is shifted right by six bits before writing to the registers. This operation is not depicted in Figure 10 but is always performed after the two one dimensional IDCTs have been computed. Therefore we added this operation to the `IDCT4` instruction.

The `IDCT4` instruction was made available to the programmer through inline assembly. Using inline assembly it is possible to specify that both operands of the instruction are read and written. The following code shows how to do this:

```
asm volatile ("idct4 %0,%1"         \
        : "=r" (v0), "=r" (v1)      \
        : "=0" (v0), "=1" (v1)      \
        );
```

The instruction was added to the simulator and the latency was set to ten cycles for the following reason. Fixed point operations take two cycles in the SPU architecture. The `IDCT4` instruction performs four steps of simple fixed point operations making a total of eight. We also count one cycle for the final shift (although this could be hardwired) and one cycle for writing the second result to the register file.

4.5 Other Intra-Vector Instructions

In the previous two sections we proposed the IDCT intra-vector instructions. The results will show that these instructions provide a large speedup for the IDCT kernels. The question arises if such intra-vector instructions can be effective for other kernels as well.

At first glance, it seems to be a lot more difficult to find a practical and effective intra-vector instruction for the deblocking filter and the luma kernel. In order not to loose DLP and with that performance, intra-vector instructions should produce multiple outputs. The deblocking filter produces multiple outputs (up to six pixels can be modified) but has a complex, dynamically selected filter function with several input parameters. A thorough investigation is necessary to reveal the exact costs and benefits of a deblocking filter instruction. The interpolation filter used in the luma kernel produces only one output. Using such an instruction would decrease the amount of exploited DLP severely. However, from one row of fourteen input pixels eight output pixels can be computed because there is a large overlap in the required input pixels. It will have to be investigated if the hardware cost of these eight interpolation filters (that include multiplications) is worth the benefits. We are currently in the process of investigating these issues.

4.6 Enhancing Memory Accesses

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 LS 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. Listing 11 shows an example of an unaligned load taken from the Luma kernel. First, two aligned quadwords are loaded, each containing a part of the targeted quadword. Next, the two quadwords

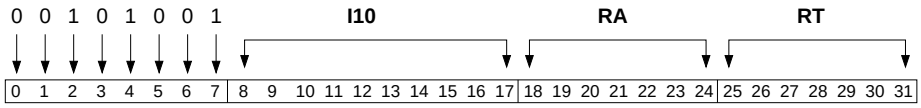
are shifted left and right respectively to put the elements in the correct slot. Finally, the two words are combined using an OR instruction. The procedure for an unaligned store is even more complicated and involves two loads and two stores.

Listing 11: Example code from the Luma kernel that loads one quadword from an unaligned memory address (`src`).

```
1 int perm_src = (unsigned int) src & 15;
2 vuint8_t srcA = *(vuint8_t *)(src);
3 vuint8_t srcB = *(vuint8_t *)(src+16);
4 vuint8_t src= spu_or(spu_slqwbyte(srcA, perm_src),
5                      spu_rlmaskqwbyte(srcB, perm_src-16));
```

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. The details of these instructions are depicted below. We did not implement the unaligned version of the a-form Load and Store (L/S) instructions as we did not find those necessary.

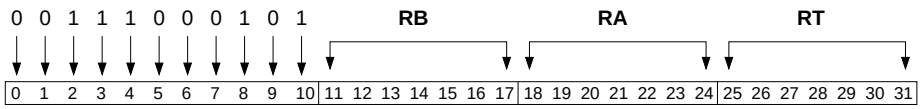
lqdu rt,symbol(ra)



$LSA \leftarrow (\text{RepLeftBit}(\text{I10} \parallel \text{0b0000,32}) + RA^{0:3}\ \&\ \text{LSLR}$

$RT \leftarrow \text{LocStor}(LSA, 16)$

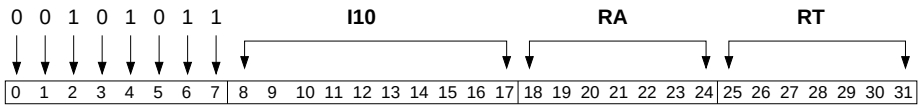
lqxu rt,ra,rb



$LSA \leftarrow (RA^{0:3} + RB^{0:3})\ \&\ \text{LSLR}$

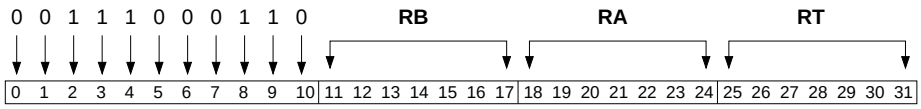
$RT \leftarrow \text{LocStor}(LSA, 16)$

stqdu rt,symbol(ra)



LSA	$\leftarrow (\text{RepLeftBit}(\text{I10} \parallel \text{0b0000,32}) + RA^{0:3}$ & LSLR
$\text{LocStor}(LSA, 16)$	$\leftarrow RT$

stqxu rt,ra,rb



LSA	$\leftarrow (RA^{0:3} + RB^{0:3})$ & LSLR
$\text{LocStor}(LSA, 16)$	$\leftarrow RT$

To allow byte aligned 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 misaligned quadword from memory. To account for this we added one cycle to the latency of the local store.

The unaligned L/S instructions were made available to the programmer through inline assembly only. Modifying the compiler to automatically use the unaligned L/S instructions is a complex task beyond the scope of this project. We tried to use intrinsics but we couldn't make them work properly within reasonable time.

Analysis of the assembly code revealed that the compiler was not able to schedule the unaligned L/S instructions as optimal is it does for normal L/S instructions. Therefore we optimized the scheduling of the unaligned L/S instructions manually. We did not hand optimize the scheduling of other instructions to keep the comparison fair.

5 Performance Evaluation

In the previous section we presented many ISA enhancements to accelerate media applications, especially H.264 decoding. This section evaluates what the effect of these enhancements is on the kernels of our benchmark suite. Each kernel is analyzed by means of profiling, the identified deficiencies are described, and it is shown how the proposed architectural enhancements resolve the deficiencies.

Figure 12 and Table 12 at the end of this section (pages 57 and 58, respectively) provide an overview of all architectural enhancements and modifications and the effect on the execution time and instruction count of all the kernels.

5.1 Results for the IDCT8 kernel

The first benchmark is the IDCT kernel which is the smallest of all. It takes as input a macroblock of 16×16 16-bit elements, chops it in blocks of 8×8 or 4×4 , and on each sub-block performs a one dimensional transformation first row wise then column wise. The size of the sub-blocks is dependent on the mode of the macroblock. We will refer to the these modes as IDCT8 and IDCT4, respectively and treat them as different kernels. This section describes the IDCT8 kernel while the next section describes the IDCT4 kernel.

The IDCT8 kernel was analyzed by splitting the kernel in parts and measure the execution times spend on them. The kernel consists of four 8×8 IDCTs, that are identical in terms of execution. In this analysis of the kernel we measured one 8×8 IDCT only. Table 3 presents the results. We remind the reader that this low level of profiling influences the compilation optimizations and thus the measured execution times. However, for the purpose of this analysis it suffices to know the relative execution times.

Most of the parts of the IDCT8 function explain themselves, however we describe them shortly here. Although we profiled only one 8×8 IDCT, we included the overhead caused by the loop that starts the four 8×8 IDCTs needed to process the entire macroblock. The overhead includes calculating the pointers to the source and destination address of the data and the stride. The initialization of variables mainly consists of creating or loading constants in the register file. Loading the data includes address computation but also adding an

Tab. 3: Profiling of the baseline IDCT8 kernel. Both the original and the code optimized version are depicted. No architectural enhancements are included.

	Original		Optimized	
	cycles	%	cycles	%
Loop overhead	33	6%	36	11%
Initialize variables	11	2%	11	3%
Load data & offset	18	4%	30	9%
Control	62	12%	77	24%
First transpose	28	5%	27	8%
First 1D IDCT	38	7%	38	12%
Second transpose	33	6%	32	10%
Second 1D IDCT	37	7%	37	11%
Shift right 6	12	2%	12	4%
Idct	148	28%	146	45%
Calculate dst shift	8	2%	14	4%
Addition and store	300	58%	89	27%
Addition	308	60%	103	31%
Total	518	100%	326	100%

offset to the first element of the 8×8 matrix for rounding purposes. The real computation of the IDCT involves two transposes, two one dimensional IDCTs, and finally a divide by 64 (or a shift right by 6 bits).

A part of the kernel is to add the result of the IDCT to the result of the prediction stage. In the original code this part takes up most of the execution time. Analysis of trace files revealed that it has a very low IPC because the compiler is not able to schedule that part of the code in an efficient way. The *addition and store* part consists basically of one macro for each matrix row. The macro loads one row of the matrix containing the result of the prediction stage, aligns the data, unpacks it, adds the result of the IDCT to it, packs the result, and stores it in memory. This macro expounds into twelve instructions that are completely serial and have many dependencies. The compiler schedules these blocks of twelve instructions sequential resulting in an IPC of 0.36 for this part. In other words, instructions are mainly stalled, waiting for the previous instruction to finish.

We optimized this part of the code by interleaving the instructions of the blocks manually. This is possible as processing the rows of the matrix is completely independent. Furthermore, we removed one unnecessary operation and combined a shift and a shuffle in one shuffle. These optimizations speedup up this part of the kernel by a factor three as depicted in the right side of Table 3.

We chose to use the code optimized version of the kernel as our baseline for evaluating architectural enhancements. Taking the non-optimized kernel it would be to easy to achieve speedup by architectural enhancements resulting in non realistic results.

Beside profiling the kernel, we also analyzed the assembly code of the kernel and the trace files generated by the simulator. In doing this we identified a number of deficiencies in the execution of the IDCT8 kernel. Below we present the

identified deficiencies and show how the proposed architectural enhancements of Section 4 resolve these issues.

- Matrix transposes: In order to utilize the SIMD capabilities, each matrix has to be transposed twice. This takes up a lot of time, even about as much as the actual IDCT computation does. Matrix transposes can be speeded up using the swapoe instructions. However, using the IDCT8 instruction the matrix transposes can completely be avoided.

- Scalar operations: Due to the lack of a scalar path, a scalar computation that involves a vector element takes a lot of instructions. Scalars are stored in the preferred slot of a vector. If one of the operands is an element of a vector, it has to be shifted to the preferred slot, the computation has to be done, and using a shuffle the result is stored in the original vector. The as2ve instructions allow to add a scalar to one element of a vector in one instruction.

- Lack of saturating arithmetic: The SPUs do not have saturating arithmetic and thus this has to be performed in software. This involves unpacking, adding, comparing, selecting, and packing. The asp instruction contains saturating arithmetic and can be used to avoid software saturation.

- Pack/unpack & alignment: Packing and unpacking has to be performed explicit and is done using shuffle instructions. Moreover, most packing and unpacking require alignment which has to be performed explicit as well. Consider an unpack from eight bytes elements to halfwords elements. If the programmer cannot be sure whether the eight bytes are in the lower or upper part of the vector he has to check the address, compute the shift and rotate the data. After that the unpack can be done. In the IDCT kernel this is the case. A part of this problem (mainly packing) is solved by the asp instruction. Alignment issues remain a deficiency.

Besides the enhancements mentioned above, also the sfxsh instructions are beneficial to the IDCT8 kernel. Table 4 shows the effect of the enhancements on the execution time and the instruction count of the IDCT8 kernel. First it shows what the effect is of each of the enhancement separately. Finally, also the combination of all enhancements was analyzed. This set of enhancements excluded the swapoe instructions as they become obsolete when using the IDCT8 instruction, which eliminates the matrix transposes. For this analysis we run the IDCT kernel on one complete macroblock. We remind the reader that this kernel-level profiling does not influence the compiler optimizations and thus these results are accurate.

The speedup achieved by the enhancements lies between 0.99 and 1.39. The IDCT8 instruction achieves the largest speedup, mostly due to avoiding matrix transposes. The swapoe instructions do not provide speedup in this kernel. The amount of ILP in the kernel is low and thus the critical path in the dependency graph determines the latency of the kernel. Although using swapoe instructions the number of executed instructions is reduced, the critical path of the matrix transpose does not change. Therefore, the critical path of the entire kernel does not change and neither does the latency.

Tab. 4: Speedup and reduction in instruction count of the IDCT8 kernel using the proposed architectural enhancements. The effect of each separate enhancement is listed as well as the aggregate effect.

Enhancement	Cycles			Instructions		
	Baseline	Enhanced	Speedup	Baseline	Enhanced	Reduction
Swapoe	896	908	0.99	1103	988	1.12
IDCT8	896	645	1.39	1103	756	1.46
Sfxsh	896	840	1.07	1103	984	1.12
Asp	896	781	1.15	1103	928	1.19
As2ve	896	865	1.04	1103	1072	1.03
All (but swapoe)	896	488	1.84	1103	488	2.26

The speedup achieved when combining all architectural enhancements is 1.84. Note that also a reduction in the number of instructions of 2.26 is achieved. The latter is important for power efficiency which has become a major design constraint in contemporary processor design. The achieved speedup is directly related to the reduction in instruction count, but does not necessarily has the same magnitude. For most kernels the speedup is lower than the instruction count reduction. This is expected since multiple shorter-latency instructions have been replaced by a single instruction with a longer latency. In other words, the latency of the arithmetic operations stays the same. For example, the latency of the IDCT8 instructions is assumed to be 8 cycles, because there are 4 simple fixed point operations on the critical path, each taking two cycles. The application-specific instructions mainly reduce the overhead and eliminate stalls due to data dependencies.

5.2 Results for the IDCT4 kernel

Besides the IDCT8 kernel we also investigated the IDCT4 kernel. The IDCT4 kernel splits a MB into 16 sub-blocks of 4×4 pixels and performs the transform on these sub-blocks. We analyzed this kernel by profiling the execution time spend on the different parts. The IDCT4 kernel has the same execution steps as the IDCT8 and the explanation of those can be found in the previous section. As in the IDCT8 we measured the execution of one 4×4 IDCT only. Table 5 presents the results.

We optimized the code of the IDCT4 kernel in the same way we optimized the IDCT8 kernel. In the *addition and store* part we interleaved intrinsics in order to achieve a higher ILP. Furthermore, we removed one unnecessary operation and combined a shift and a shuffle in one shuffle. Again, these optimizations speedup this part of the kernel by a factor three as depicted in the right side of Table 5. For the same reasons as mentioned in the previous section we chose the code optimized version as our baseline for evaluating architectural enhancements.

We identified the deficiencies in the execution of the IDCT4 kernel by analyzing the profiling results and a trace output. The identified deficiencies are somewhat the same as for the IDCT8 kernel, but there are some differences. Below we explain the deficiencies and show how the proposed architectural en-

Tab. 5: Profiling of the baseline IDCT4 kernel. Both the original and the code optimized version are depicted. No architectural enhancements are included.

	Original		Optimized	
	cycles	%	cycles	%
Loop overhead	33	10%	33	15%
Initialize variables	11	3%	11	5%
Load data & offset	29	9%	30	13%
Control	73	23%	74	33%
First transpose	22	7%	22	10%
First 1D IDCT	15	5%	16	7%
Second transpose	25	8%	24	11%
Second 1D IDCT	15	5%	16	7%
Shift right 6	11	3%	12	5%
Idct	88	27%	90	40%
Calculate dst shift	7	2%	9	4%
Addition and store	156	48%	53	24%
Addition	163	50%	62	27%
Total	324	100%	226	100%

hancements of Section 4 resolve these issues.

- Matrix transposes: Because of the small matrix size the transposes are less costly but more of them are required so they still take up about 20% of the execution time. Swapoe instructions are of no value for a 4×4 matrix as it fits entirely in two registers, and thus shuffle operations are sufficient. The IDCT4 instruction, on the other hand, completely removes all matrix transposes.

- Unused SIMD slots: All SIMD computations are performed row or column wise. As the matrix is 4×4 and the elements are 16-bit only half of the SIMD slots are used. The IDCT4 instruction does utilize all SIMD slots by exploiting the fact that an entire 4×4 matrix can be stored in two registers. This increases the efficiency of the computational part of the kernel.

- Scalar operations: Same as for IDCT8 kernel; see page 47 for a full description. The as2ve instructions were used to mitigate the effect of this deficiency.

- Lack of saturating arithmetic: Same as for IDCT8 kernel; see page 47 for a full description. The asp instruction was used to mitigate the effect of this deficiency.

- Pack, unpack & alignment: Same as for IDCT8 kernel; see page 47 for a full description. The asp instruction was used to resolve the pack/unpack deficiency. Alignment issues remain a deficiency.

Besides the enhancements mentioned above, we also used the unaligned L/S instructions. By doing this we avoided some branches that caused many stalls.

Table 6 shows what the effect of the enhancements is on the execution time and the instruction count of the IDCT4 kernel. First it shows what the effect is of each enhancement separately. Finally, also the combination of all enhancements was analyzed. For this analysis we run the IDCT4 kernel on one complete macroblock.

Tab. 6: Speedup and reduction in instruction count of the IDCT4 kernel using the proposed architectural enhancements. The effect of each separate enhancement is listed as well as the aggregate effect.

Enhancement	Cycles			Instructions		
	Baseline	Enhanced	Speedup	Baseline	Enhanced	Reduction
IDCT4	2524	2059	1.23	1858	1304	1.42
Asp	2524	2277	1.11	1858	1512	1.23
As2ve	2524	2407	1.05	1858	1756	1.06
Unaligned L/S	2524	2086	1.21	1858	1720	1.08
All	2524	1063	2.37	1858	696	2.67

The speedup achieved by the enhancements lies between 1.05 and 1.23. A synergistic effect happens when combining all enhancements such that the total speedup is larger than the multiplication of the separate speedups. The source of this synergistic effect seems to be the compiler optimization process. The speedup achieved when combining all architectural enhancements is 2.37 and the number of instructions is reduced by a factor 2.67.

5.3 Results for the Deblocking Filter kernel

The Deblocking Filter (DF) kernel is applied to remove an artifact introduced by the IDCT. The latter operates on 4×4 or 8×8 sub-blocks and as a result square areas might become visible in the decoded picture, which is called the 'blocking' artifact. The DF kernel smoothes these block edges and thus improves the appearance of the decoded picture.

The process of deblocking a macroblock is depicted in Figure 11. Each square represents a 4×4 sub-block. Note that also the left and top neighboring sub-blocks are depicted as they are involved in the filtering. The filter is applied on all edges, starting with the vertical ones from left to right followed by the horizontal ones from top to bottom. The filter is applied to each line of pixels perpendicular to the edge and with a range of four pixels to each side of the edge. The strength of the filter is determined dynamically based on certain properties of the macroblock, but more importantly on the pixel gradient across the edge. This adaptiveness decreases the available DLP, but still an efficient SIMD implementation for the Cell SPU was possible [7].

We analyzed the code of the DF kernel by profiling. Because the code of the kernel is not as small as the IDCT kernels we profiled the code on several levels, of which we present here the most important.

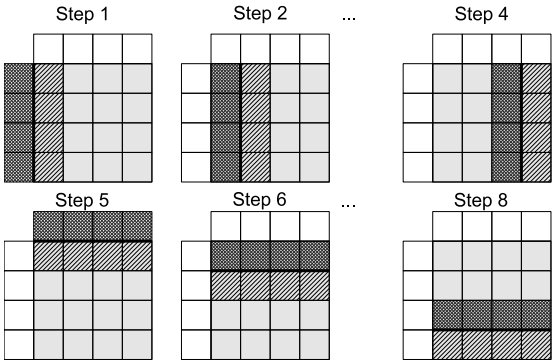


Fig. 11: The process of deblocking one macroblock. Each square is a 4×4 sub-block. Also the left and top neighboring sub-blocks are involved in the process.

Table 7 shows the profiling results of the top level, i.e., the function 'filter_mb_spu'. This function performs the filtering of one macroblock. The several parts that we identified in this function mostly speak for themselves. Packing and unpacking is necessary because the data is stored in memory as bytes but processing is done using halfwords. The matrix transposes are needed to rearrange the data such that it is amenable to SIMD processing. Analyzing the assembly code revealed that the explicitly loaded data was too large to maintain in the register file during the entire execution. The compiler itself generated a lot of loads and stores before and after each processing stage.

Tab. 7: Top level profiling of the DF kernel for one macroblock. Both the original and the code optimized version are depicted. No architectural enhancements are included.

	Original		Optimized	
	cycles	%	cycles	%
Initialize variables	22	0%	22	0%
Load & unpack data	761	10%	761	10%
First transpose	451	6%	451	6%
Filter horizontal	2741	34%	2539	34%
Second transpose	448	6%	448	6%
Filter vertical	2748	34%	2546	34%
Pack and store data	806	10%	806	11%
Total	7977	100%	7573	100%

The total DF kernel consists of several functions that call each other in a tree like fashion. To find the most time consuming parts we analyzed the number of function calls as well as the total amount of time spend in each function (including its subfunctions). The results are depicted in Table 8 which also states the level of each function. Functions of level x only call functions of level

$x + 1$. For this analysis we filtered an area of 4×4 macroblocks such that we include macroblocks on the left and top side of a frame as well as those completely surrounded by other macroblocks.

Tab. 8: Profiling of the functions of the DF kernel when filtering 4×4 macroblocks. The number of function calls and the time spend in the function (including subfunctions) is mentioned. Functions of level x only call functions of level $x + 1$.

	Calls	Time	Level
filter_mb_spu	16	100%	1
filter_mb_edgeh	240	53%	2
filter_mb_edgecv	112	18%	2
h264_v_loop_filter_luma_c	192	19%	3
h264_loop_filter_chroma	64	5%	3
h264_loop_filter_chroma_intra	48	1%	3
clip	704	5%	3
clip_altivec	640	4%	4
clip_uint8_altivec	512	3%	4

A great amount of time is spend in the 'filter_mb_edgeh' function which mainly calls 'h264_v_loop_filter_luma_c'. However, the latter is only 19% of the total execution time while the first is 53%. Further profiling revealed that in the first a lot of time is spend on table lookups which are time consuming on the SPU.

Table 8 also shows that 12% is spend in clip functions. The 'clip' and 'clip_altivec' functions are used to set the index of table lookups within the range of the table (mainly between 0 and 51). The 'clip_uint8_altivec' function is used to saturate a halfword between 0 and 255. This is necessary because the SPU does not contain saturating arithmetic.

While analyzing the kernel we also looked for non-optimal code. In this kernel we did not find much to optimize. We did change some code that generated constants. The code was written in normal C and the compiler generated non-optimal assembly code. We rewrote the code using SPU intrinsics. The right side of Table 7 shows the profiling results of the code optimized kernel. As with the previous kernels, we chose the code optimized version as the baseline for evaluating architectural enhancements.

We identified a number of deficiencies in the execution of the DF kernel. Below we present those and show how the proposed architectural enhancements of Section 4 can resolve these issues.

- Matrix transposes: In order to utilize the SIMD capabilities each data matrix has to be transposed twice. This rearrangement overhead takes up about 12% of the total execution time. Matrix transposes can be speeded up using the swapoe instructions.

- Data size: The total data size of the baseline kernel exceeds the size of the register file and therefore a lot of intermediate loads and stores are required. The main cause of the large data size are the matrix transposes as they require a double amount of registers: one set of registers for the

original and one for the transposed. The swapoe instructions perform a matrix transpose in place and thus require half the amount of registers. Simulation showed that using the swapoe instructions the entire data set fits into the registers file and thus many loads and stores are prevented.

- Scalar operations: As with the previous kernels, scalar operations turn out to be very costly. In the DF kernel this manifested in the table lookups. Listing 1 of Section 4.1.1 shows that a table lookup costs five instructions because of alignment issues. Using the lds instructions a table lookup takes two instructions.

- Lack of saturating function/arithmetic: The output of the DF kernel are unsigned bytes. Thus, the result of the filtering has to be saturated between 0 and 255. Many machines offer saturating arithmetic which does the saturation automatically. However, in the DF kernel halfwords are needed for the intermediate results anyway, and thus one saturate at the end suffices. Implementing many saturating instructions is costly in terms of opcodes. The clip instruction is in that sense a good solution as it provides a fast saturation with only one instruction per data type. Moreover, the clip instruction can also be used for other purposes as limiting the index of a table lookup to the size of the table.

- Pack & unpack: Packing and unpacking has to be performed explicit and is done using shuffle instructions. The asp or sat_pack instructions are of no help in the DF kernel as the saturate and the pack are not coupled together. Automatic (un)pack at load and store would be beneficial. As other people have investigated this before [15, 16] we do not include this technique in our architectural enhancements and the analysis of those.

- SIMDimization: Because of the adaptiveness of the filter limited DLP is available. Five different filter strengths are applied depending on the smoothness of the edge. That means that across the SIMD slots different filters might be required. The SIMDimization strategy applied is to compute all five and to select the right result for each SIMD slot. Off course this requires extra computation. None of the proposed architectural enhancements resolves this issue.

Table 9 presents the speedup and reduction in instruction count of the DF kernel using the proposed architectural enhancements. For this analysis we also applied the DF to the top left 4×4 macroblocks of a frame from a real video sequence. The swapoe instructions have a large impact on the execution time, mainly due to avoiding loads and stores. To achieve the latter some modifications to the code were required such that the compiler could ascertain all data accesses at compiler time. These modifications included loop unrolling, substituting pointers for array indices, and inlining of functions.

Both the lds and clip instructions provide a moderate speedup but significant considering the total kernel. When combining all three enhancements again a synergistic effect happens as the total speedup is larger than the multiplication of the separate speedups. The reduction in the number of instructions is proportional to the speedups. Combining all enhancements the number of instructions is reduced with a factor 1.75.

Tab. 9: Speedup and reduction in instruction count of the DF kernel using the proposed architectural enhancements. The effect of each separate enhancement is listed as well as the aggregate effect.

Enhancement	Cycles			Instructions		
	Baseline	Enhanced	Speedup	Baseline	Enhanced	Reduction
Swapoe	122135	90518	1.35	101480	74504	1.36
Lds	122135	108995	1.12	101480	89692	1.13
Clip	122135	114755	1.06	101480	96312	1.05
All	122135	66334	1.84	101480	58088	1.75

For the DF kernel the speedup is larger then the reduction in instructions for the following reason. When using the swapoe instructions it is possible to keep the entire macroblock in the register file. To achieve this some modifications to the code were required among which replacement of functions by macros. This reduced the number of branches significantly and with that the branch miss penalties. In the baseline kernel there were 15341 branch miss stall cycles, while for the enhanced kernel this was only 4529.

5.4 Results for the Luma/Chroma Interpolation Kernel

The Luma and Chroma interpolation kernels together constitute the motion compensation. The kernels are applied to blocks of 16×16 , 16×8, 8×16, 8×8 , 8×4, 4×8, and 4×4 pixels. Note that there is only one chroma sample per 2×2 pixels. For the rest the two kernels are identical and thus we focus on the Luma kernel only.

The accuracy of the motion compensation is a quarter of the distance between luma samples. If the motion vector points to an integer position, the predicted block consists of the samples the motion vector points to in the reference frame. If the motion vector points to a half-sample position, the predicted block is obtained by interpolating the surrounding pixels. This is done by applying a one-dimensional 6-tap FIR filter:

$$p[0]' = (p[-2] - 5p[-1] + 20p[0] + 20p[1] - 5p[2] + p[3]) + 16) >> 5 \quad (1)$$

horizontally and/or vertically. Prediction values at quarter-sample positions are generated by averaging samples at integer- and half-sample positions. For more information on the motion compensation of H.264 the reader is referred to [17].

The code of the Luma kernel contains many functions as there are many possible combinations of block sizes and quarter sample positions. For the analysis in this work we chose an input block size of 16×16 and a quarter sample prediction position that requires both horizontal and vertical interpolation. This filter function is one of the most compute intensive ones. The name of this filter function in the FFmpeg code is `put_h264_qpel16_mc11_spu`. The input for this kernel is one macroblock and thus the filter function is run once.

We analyzed the code of the Luma kernel by profiling it using CellSim. The filter function calls three subfunctions: `put_h264_qpel_v_lowpass_spu` (33%)

performs the vertical interpolation, `put_h264_qpel_h_lowpass_spu` (52%) performs the horizontal interpolation, and `put_pixels_16_l2_spu` (16%) averages the result of the first two functions and stores it in the destination address. We divided the functions in six parts as stated in Table 10 and measured how many cycles were spend in each of the parts combined for all three functions. Only 22.5% of the cycles is spend on actual computation of the interpolation. Most of the time is spend on loading/storing and the corresponding rearrangement of data to make it amenable to SIMD processing.

Tab. 10: Profiling results of the baseline Luma kernel. Both the original and the code optimized version are depicted. No architectural enhancements are included.

	Original		Optimized	
	cycles	%	cycles	%
Initialization of variables	44	1%	27	1%
Loop overhead	752	16%	896	21%
Loading, aligning, and unpacking	1630	36%	1546	37%
Computing interpolation filter	1024	23%	976	23%
Saturating and pack	512	11%	416	10%
Storing result	592	13%	368	9%
Total	4554	100%	4229	100%

While analyzing the kernel we also looked for non-optimal code. We performed three optimizations. The first optimization is in the horizontal and vertical interpolation functions. The original code used only the `spu_madd` intrinsic, which multiplies the odd elements only, and used shuffles to multiply even elements. We used the `spu_mule` and `spu_mulo` in places where they are beneficial and avoided thereby some of the shuffles. The `spu_mhhadd` instruction/intrinsic turned out not to be beneficial.

The second code optimization took place in the `put_pixels_16_l2_spu` function. It computes the average of two 16×16 matrices of which it assumes they are stored in memory non quadword aligned. Therefore this function contains a lot of overhead to align the data. However, in only few cases the input matrices are indeed stored non quadword aligned. For the other cases the alignment overhead can be avoided. For the filter function that we chose as subject of this analysis this is always the case. Thus we created another version of this function called `put_pixels_16_l2_spu_aligned` and use that one in our filter function.

The third optimization was performed on all three mentioned subfunctions. All those functions contain a loop that iterates over all the rows of the block being processed. The height of the block is a parameter of the function. Therefore, the loop count can not be determined at compile time and thus the compiler does not perform loop unrolling optimizations. The bodies of these loops are rather small and some contain very few ILP. Thus without loop unrolling a lot of dependency stalls occur. The height of the block can take only two values. Thus, we created two versions of each function, one for each block height. We applied this enhancement to the three subfunctions only when beneficial. For example, for the baseline kernel we applied this optimization to the smallest

subfunction only.

The right side of Table 10 shows the profiling results of the code optimized kernel. However, the third code optimization is not reflected in this analysis. The increase in ILP obtained by it was destroyed by the insertion of the volatile profiler commands. In total the code optimized version is about 33% faster as the original one. As with the previous kernels, we chose the code optimized version as the baseline for evaluating architectural enhancements.

We identified a number of deficiencies in the execution of the Luma kernel. Below we present those and show how the proposed architectural enhancements of Section 4 can resolve these issues.

- Unaligned loads: The Luma kernel loads the sub-block where the motion vector is pointing to. In far most cases this motion vector points to a non quadword aligned memory address. To load such an misaligned matrix into the register file a lot of overhead is required. This main deficiency of the Luma kernel was resolved by using the unaligned load instructions.

- Multiplication overhead: The input and output of the Luma kernel is a matrix of bytes while computations are performed in halfwords in order not to loose precision. When a multiplication is performed the instructions produce 32-bit results although the actual value never exceeds 15 bits. Thus a multiplication of one vector is performed as a multiplication of the odd elements, a multiplication of the even elements, and a shuffle to pack the odd and even elements back into one vector. This overhead can be avoided either using the mpytr instructions or the mpy_byte instructions.

  The mpytr instructions multiply vectors of halfwords and produce vectors of halfwords. The mpy_byte instructions multiply vectors of bytes and produce vectors of halfwords. The latter requires that the multiplication is done on the byte source data directly. In general that will not always be possible, but for the Luma kernel it is. The additional benefit of the mpy_byte instructions is that they unpack the input data automatically, making the explicit unpacks superfluous.

  In the horizontal interpolation function the mpy_byte instructions provided most speedup while in the vertical interpolation function the mpytr instructions did. Due to the way these functions are SIMDimized, the horizontal interpolation function contains much more unpacks which are done automatically by the mpy_byte instructions. The vertical interpolation function has few unpacks and does not benefit that much from the auto unpack of the mpy_byte instructions. However, using the mpy_byte instructions more total instructions are required for the same filter.

- Lack of saturation function/arithmetic: Just as the other kernels the result of the Luma kernel has to be saturated between 0 and 255 and packed to bytes. The clip instruction can be used for this purpose but the sat_pack instruction provides more benefit to the overall performance.

- Pack & unpack: Packing and unpacking has to be performed explicit and is done using shuffle instructions. Automatic (un)pack at load and store would be beneficial, but is not investigated in this work as said before. The sat_pack instruction does provide speedup as it combines a saturate and a pack.

Tab. 11: Speedup and reduction in instruction count of the Luma kernel using the proposed architectural enhancements. The effect of each separate enhancement is listed as well as the aggregate effect.

Enhancement	Cycles			Instructions		
	Baseline	Enhanced	Speedup	Baseline	Enhanced	Reduction
Mpytr	2461	2238	1.10	2473	2163	1.14
Mpy_byte	2461	2175	1.13	2473	2056	1.20
Clip	2461	2333	1.05	2473	2311	1.07
Sat_pack	2461	2065	1.19	2473	2212	1.12
Sfxsh	2461	2375	1.04	2473	2409	1.03
Unaligned L/S	2461	1915	1.29	2473	1936	1.28
All (but Clip)	2461	1068	2.30	2473	1059	2.34

Besides the architectural enhancements mentioned above, also the sfxsh instructions are beneficial to the Luma kernel, although very little. Table 11 presents the speedup and reduction in instruction count of the Luma kernel using the proposed architectural enhancements. For this analysis we run the Luma kernel on one entire macroblock.

The mpytr, mpy_byte, and sat_pack instructions provide a moderate speedup. The clip and sfxsh instructions are used only a few times in the Luma kernel and provide a small speedup. The unaligned L/S instructions provides most speedup as loading and storing is a large part of this kernel. Combining all enhancements, the clip instruction is not used as the saturation it performed is done by the sat_pack instructions. The total speedup achieved is 2.30 while the instruction count is reduced by a factor of 2.34.

Fig. 12: Speedup and reduction of the instruction count achieved on the H.264 kernels.

Tab. 12: Overview of the effect of all architectural enhancements and modifications 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

6 Related Work

There are many architectures that aim to speed up media applications. First of all there are General Purpose Processors (GPPs) with (multi)media SIMD extensions. The Intel IA-32 architecture was extended with MMX [18] and later with SSE [19]. AMD has its 3DNow! [20] technology, IBM's PowerPC was extended with Altivec/VMX [21], VIS was used in Sun's SPARC architecture, Silicon Graphics extended the MIPS architecture with MDMX, and MVI was added to the Alpha processor. A very small media extension (16 instructions only) was the MAX-2 [12] used in HP's PA-RISC core. GPPs provide high flexibility and programmability, but are power inefficient and cannot always meet real-time constraints [4].

The second class of architectures is the multimedia processor that aims to speedup a broad range of multimedia applications. Especially for embedded devices such a broad range of applications has to be supported. Most of the multimedia processors available in the market are really SoCs, like the Intel CE 2110 [22] and the Texas Instruments DaVinci Digital Media Processors [23]. Both are capable of H.264 video coding, but use a hardware codec for it. Such platforms lack the flexibility of enabling new codecs or codec profiles.

Also academia has proposed several multimedia architectures. Among those are stream/vector processors like Imagine [24], VIRAM [25], RSVP [26], and ALP [27]. Prerequisite to using those efficiently are certain application characteristics like little data reuse (pixels are read once from memory and are not revisited) and high data parallelism (the same set of operations independently computes all pixels in the output image) [24]. Many media applications exhibit those characteristics, but not modern video codecs like H.264. Motion compensation causes data to be accessed multiple times in an irregular fashion. DLP is limited by the size of the sub-block that can be as small as 4×4 . Because macroblocks can be partitioned in sub-blocks in various ways and because each sub-block can be coded with different parameters, there is no regular loop that architectures like the VIRAM needs to exploit DLP. Moreover, due to many data dependencies parallelizing is not trivial. The only kernel of H.264 that can benefit from a streaming architecture is entropy coding. For the rest of the application, explicit data transfers using DMAs is more efficient in our opinion. The ALP architecture supports all levels of parallelism (DLP, ILP, and TLP) and performs well on mpeg2. However, as it still relies on regular data streams we expect this architecture not to perform very well on a modern codec like H.264.

The CSI [16], MOM [28], and MediaBreeze [29] architectures optimize SIMD extensions by reducing the overhead of data accesses using techniques as hardware address generation, hardware looping, data sectioning, alignment, reorganization, packing/unpacking, etc. The MOM architecture is matrix oriented and supports efficient memory access for small matrices as well as some matrix operations like matrix transpose. All of these architectures provide two or more dimensions of parallelism, yet recognize the need to exploit fine-grained parallelism. Some of these techniques could be effective for modern video coding. Issues like alignment, reorganization, and packing/unpacking were also addressed in the SARC Media Accelerator.

The third class of architectures are Application Specific Instruction set Processors (ASIPs). These architectures are optimized for one or few applications

in the multimedia domain. They provide the best power efficiency while maintaining flexibility and programmability and can therefore support a wide variety of standards. The SARC Media Accelerator is an ASIP based on the Cell SPE architecture. The TriMedia is a VLIW ASIP for audio and video processing. The latest version (TM3270) has been specialized for H.264 encoding and decoding [30]. Our architecture is specialized for decoding only but takes specialization a step further. Tensilica's Diamond 388VDO Video Engine has two cores and a DMA unit [31]. One core is a SIMD architecture while the other is a stream architecture. The latter is used for entropy coding and assists the SIMD core in motion prediction. Further architectural details of this processor could not be found.

An early ASIP is found in [32] containing an instruction for motion estimation, some instructions for packing and interleaving, a specialized datapath, and IO for audio/video. Another early ASIP is found in [33], and has an interesting system architecture. A RISC core is used as control processor, a SIMD processor is used for the main decoding, and a sequence processor is used to perform entropy decoding. Moreover, the system provides local memory with a DMA unit. The cores itself though are very simple and do not have any real specialization. In [34] an ASIP was proposed to enhance pixel processing. It uses SIMD vectors with 8 bits for the luma component and two times 4 bits for the chroma components. This kind of SIMD can exploit only limited amount of DLP. Furthermore, the H.264 colorspace uses mainly a 4:2:0 subsampling scheme. That means that the number of chroma samples is reduced instead of the number of bits per chroma component. Thus this architecture cannot be used for modern video codecs.

Two other ASIPs are found in [35, 36]. Both processors target motion estimation only. In video encoding, motion estimation takes up most of the time, and therefore these ASIPs are very interesting and relevant work.

Finally, the last class of architectures are the ASICs. Some combinations of ASICs with GPPs for multimedia in general are mentioned above. There are also many examples of GPPs extended with ASIC for a very specific target [37, 38, 39, 40]. ASICs are the most power efficient solution and find their way in many embedded systems. In systems where a large number of standards have to be supported they might not be the most cost efficient solution. Even, when programmability is required, for example to support future standards or profiles, ASICs are not an option.

So far we discussed architectures at a high level. In the next sections we describe work related to the specific enhancements we applied in the SARC Media Accelerator.

6.1 Matrix Transposition

A linear time matrix transposition method was proposed in [41], which utilizes a special vector register file that allows diagonal access. The in-place matrix transpose (IPMT) algorithm allows a matrix, stored in register file, to be transposed by performing a number of rotations on the diagonal registers. The number of instructions required by this approach, assuming the buffered implementation, is $2n-2$. For $n = 16$ and $n = 8$ buffered IPMT requires 30 and 14 instructions, respectively which both are approximately twice as fast as the AltiVec MT. Our implementation using the swapoe instruction takes 32 and 12 instructions for

$n = 16$ and $n = 8$ respectively. Thus our approach is as fast as IPMT but is much simpler as we use a conventional register file.

The swapoe instructions are similar to the MIX instructions used in HP's MAX-2 [12]. Although these instructions found their way into the Itanium ISA [42] they have never been recognized as useful for media acceleration. In that sense we rediscovered these instructions and showed how they can speedup media applications.

As mentioned above, the MOM architecture provides a matrix transpose instruction. According to [28] this transpose takes eight cycles for an 8×8 matrix. As the architecture is only described at the ISA level, it is unclear how this is implemented and what the impact on the performance is.

6.2 IDCT

The (inverse) discrete cosine transform has been used in many applications to compress image data. Several algorithms exist to compute the (I)DCT and many hardware implementations have been proposed. Some of the more recent proposals can be found in [43] and [44], both which provide a good overview of other implementations in the past. Simai et al [45] proposed an implementation for a programmable media processor. They extended the TriMedia processor with a reconfigurable execution unit and demonstrated how to implement the IDCT on it.

The (I)DCT algorithm used in the H.264 standard is a simplified version that is much less complex than previous algorithms. For example, it uses only addition, subtraction, and shifts where other algorithm required many multiplications. This algorithm was proposed in [46, 47] for the 4×4 transform while the algorithm for the 8×8 transform was proposed in [48]. To the best of our knowledge, the only hardware implementation of this algorithm was proposed in [49]. It performs the 8×8 2D IDCT in a sequential manner, i.e., it consumes and produces one sample per cycle. Their hardware unit was designed to be used in a streaming environment and therefore is not very suitable for a programmable media accelerator. Further difference with our approach is that we exploit DLP reducing the latency of the IDCT operation.

Similar to our work, in [50] the authors proposed some techniques to speedup the Discrete Wavelet Transform (DWT) on SIMD architectures. DWT has a better compression ratio but is computationally more complex than the DCT. The authors proposed to add a MAC instruction to the ISA, use the extended subwords technique, and use a matrix register file.

6.3 Unaligned memory access

In vector processing, of which SIMD is a sub class, there is often a mismatch between the data order used in memory and the data order required for computation. Therefore, most vector and DSP processors provide complex memory access, that can be divided in three groups: unaligned access, strided access, and irregular access. In SIMD processors, due to the short vectors, mainly unaligned access is an issue. Multimedia SIMD extensions such as MMX, and SSE support unaligned accesses in hardware at the cost of some additional latency. In other SIMD architectures, such as Altivec/VMX, VIS, and 3DNow! alignment has to be performed in software, which is very costly in terms of latency.

Alignment issues on different multimedia SIMD architectures has been investigated in [51]. The paper describes several software approaches to minimize the overhead costs. The benefits of supporting unaligned access in hardware is studied in detail in [52] for the Altivec/VMX architecture. Although there research is general, they give an example of how unaligned access could be implemented. Similar work is presented in [53, 54], but those focus more on the actual memory access architecture and compiler optimizations. The latest TriMedia processor, which was specialized for H.264 supports unaligned memory access.

6.4 Others

A combination of pack and saturation is also found in the pack.sss and pack.uss instructions of the IA64 architecture [42]. Also MMX has instructions that combine pack and saturate. In the TriMedia there are a few instructions that include a clip operation.

One of the first ISA specializations for video coding was presented in [55] in 1999. The authors suggested some simple instructions for the upcoming mpeg4 standard. Many of the proposed instructions, e.g., arbitrary permute, MAC, round, and average, are being used in nowadays architectures. In [56] an ISA specialization is presented for H.263. Their instructions could be used for H.264 as well but are not as powerful as the ones we propose.

7 Conclusions and Future Work

In this report we described the specialization of the Cell SPE for media application, specifically H.264. The SPE architecture is an excellent starting points as it has a large register file, a local store, and DMA style of accessing global memory. This is very suitable for H.264, as most data accesses in are know in advance. 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 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 they completely avoid matrix transposes, and by the unaligned load/store instructions because of the large overhead reduction.

The IDCT instructions proposed in this report are unconventional in the sense that they perform intra-vector operations. In general SIMD instructions perform operations among elements that are in the same slot, but in different vectors. We will investigate if such intra-vector SIMD instructions are beneficial for other kernels and other application domains.

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 *Proceedings of the Workshop on Highly Parallel Processing on a Chip (HPPC)*, 2008.

[2] B. Flachs, S. Asano, S. Dhong, H. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, *et al.*, "The Microarchitecture of the Synergistic Processor for a CELL Processor," in *Proceedings of the IEEE International Solid-State Circuits Conference*, 2005.

[3] *Cell Broadband Engine Programming Handbook*, IBM, 2006. [Online]. Available: http://www.bsc.es/plantillaH.php?cat_id=326

[4] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez, and A. Ramirez, "Parallel Scalability of Video Decoders," *Journal of Signal Processing Systems*, 2008.

[5] T. Oelbaum, V. Baroncini, T. Tan, and C. Fenimore, "Subjective Quality Assessment of the Emerging AVC/H.264 Video Coding Standard," in *International Broadcast Conference (IBC)*, 2004.

[6] "The FFmpeg Libavcoded." [Online]. Available: http://ffmpeg.mplayerhq.hu/

[7] A. Azevedo, C. Meenderinck, B. Juurlink, M. Alvarez, and A. Ramirez, "Analysis of Video Filtering on the Cell Processor," in *Proceedings of International Symposium on Circuits and Systems (ISCAS)*, May 2008.

[8] C. Meenderinck, "Implementing SPU instructions in CellSim." [Online]. Available: http://pcsostres.ac.upc.edu/cellsim/doku.php/

[9] "CellSim: Modular Simulator for Heterogeneous Multiprocessor Architectures." [Online]. Available: http://pcsostres.ac.upc.edu/cellsim/doku.php/

[10] *Cell Broadband Engine Programming Handbook*, IBM, 2006. [Online]. Available: http://www.bsc.es/plantillaH.php?cat_id=326

[11] *Synergistic Processor Unit Instruction Set Architecture*, IBM, 2007. [Online]. Available: http://www.bsc.es/plantillaH.php?cat_id=326

[12] R. Lee and J. Huck, "64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture," in *Proc. Compcon*, 1996.

[13] J. Eklundh, "A fast computer method for matrix transposing," *IEEE Transactions on Computers*, vol. 100, no. 21, pp. 801–803, 1972.

[14] C. Loeffler, A. Ligtenberg, and G. S. Moschytz, "Practical Fast 1-D DCT Algorithms With 11 Multiplications," in *Proceedings of the International Conference on Acoustical and Speech and Signal Processing*, May 1989, pp. 988–991.

[15] N. Slingerland and A. Smith, "Measuring the Performance of Multimedia Instruction Sets," *IEEE Transactions on Computers*, pp. 1317–1332, 2002.

[16] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff, "The CSI Multimedia Architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–13, January 2005.

[17] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A.Luthra, "Overview of the H.264/AVC Video Coding Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, July 2003.

[18] A. Peleg, S. Wilkie, and U. Weiser, "Intel MMX for multimedia PCs," *Communications of the ACM*, vol. 40, no. 1, pp. 24–38, 1997.

[19] S. Raman, V. Pentkovski, and J. Keshava, "Implementing Streaming SIMD Extensions on the Pentium III Processor," *IEEE MICRO*, pp. 47–57, 2000.

[20] A. M. Devices, *3DNow! Technology Manual*, Advanced Micro Devices, Inc, 2000.

[21] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scale, "AltiVec extension to PowerPC accelerates media processing," *Micro, IEEE*, vol. 20, no. 2, pp. 85–95, 2000.

[22] "Intel CE 2110 Media Processor." [Online]. Available: http://www.intel.com/design/celect/2110/

[23] "DaVinci Digital Media Processors - Texas Instruments." [Online]. Available: http://www.ti.com/

[24] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: media processing with streams," *Micro, IEEE*, vol. 21, no. 2, pp. 35–46, 2001.

[25] C. Kozyrakis and D. Patterson, "Scalable, vector processors for embedded systems," *Micro, IEEE*, vol. 23, no. 6, pp. 36–45, 2003.

[26] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (RSVP/spl trade/)," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003, pp. 141–150.

[27] R. Sasanka, M. Li, S. Adve, Y. Chen, and E. Debes, "ALP: Efficient support for all levels of parallelism for complex media applications," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 1, 2007.

[28] J. Corbal, R. Espasa, and M. Valero, "MOM: a Matrix SIMD Instruction Set Architecture for Multimedia Applications," in *Supercomputing, ACM/IEEE 1999 Conference*, 1999.

[29] D. Talla and L. John, "Cost-effective hardware acceleration of multimedia applications," 2001, pp. 415–424.

[30] J. van de Waerdt, K. Kalra, P. Rodriguez, H. van Antwerpen, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, *et al.*, "The TM3270 Media-Processor," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, 2005, pp. 331–342.

[31] "Tensilica's Diamond 388VDO Video Engine." [Online]. Available: http://www.tensilica.com/products/video/video_arch.htm

[32] K. Suzuki, M. Daito, T. Inoue, K. Nadehara, M. Nomura, M. Mizuno, T. Iima, S. Sato, T. Fukuda, T. Arai, *et al.*, "A 2000-MOPS embedded RISC processor with a Rambus DRAM controller," *Solid-State Circuits, IEEE Journal of*, vol. 34, no. 7, pp. 1010–1021, 1999.

[33] S. Lee, J. Chung, and M. Lee, "High-speed and low-power real-time programmable videomulti-processor for MPEG-2 multimedia chip on 0.6 μm TLM CMOStechnology," in *Design Automation Conference, 1999. Proceedings of the ASP-DAC'99. Asia and South Pacific*, 1999, pp. 201–204.

[34] J. Kim and D. Wills, "Quantized color instruction set for media-on-demand applications," in *Multimedia and Expo, 2003. ICME'03. Proceedings. 2003 International Conference on*, 2003.

[35] H. Peters, R. Sethuraman, A. Beric, P. Meuwissen, S. Balakrishnan, C. Pinto, W. Kruijtzer, F. Ernst, G. Alkadi, J. van Meerbergen, *et al.*, "Application specific instruction-set processor template for motion estimation in video applications," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 15, no. 4, pp. 508–527, 2005.

[36] M.-A. Daigneault, J. P. Langlois, and J.-P. David, ""application specific instruction set processor specialized for block motion estimation"," in *IEEE International Conference on Computer Design*, 2008.

[37] T. Nishikawa, M. Takahashi, M. Hamada, T. Takayanagi, H. Arakida, N. Machida, H. Yamamoto, T. Fujiyoshi, Y. Maisumoto, O. Yamagishi, *et al.*, "A 60 MHz 240 mW MPEG-4 video-phone LSI with 16 Mb embedded DRAM," in *Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International*, 2000, pp. 230–231.

[38] S. Park, S. Kim, K. Byeon, J. Cha, and H. Cho, "An area efficient video/audio codec for portable multimediaapplication," in *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, vol. 1, 2000.

[39] H. Takata, T. Watanabe, T. Nakajima, T. Takagaki, H. Sato, A. Mohri, A. Yamada, T. Kanamoto, Y. Matsuda, S. Iwade, *et al.*, "The D30V/MPEG multimedia processor," *Micro, IEEE*, vol. 19, no. 4, pp. 38–47, 1999.

[40] T. Fujiyoshi, S. Shiratake, S. Nomura, T. Nishikawa, Y. Kitasho, H. Arakida, Y. Okuda, Y. Tsuboi, M. Hamada, H. Hara, *et al.*, "A 63-mW H. 264/MPEG-4 audio/visual codec LSI with module-wise dynamic

Voltage/frequency scaling," *Solid-State Circuits, IEEE Journal of*, vol. 41, no. 1, pp. 54–62, 2006.

[41] B. Hanounik and X. Hu, "Linear-time Matrix Transpose Algorithms Using Vector Register File With Diagonal Registers," in *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, 2001.

[42] "Intel Itanium Architecture Software Developer's Manual - Revision 2.2," 2006, http://download.intel.com/design/Itanium/manuals/24531905.pdf. [Online]. Available: http://download.intel.com/design/Itanium/manuals/24531905.pdf

[43] J. Guo, R. Ju, and J. Chen, "An efficient 2-D DCT/IDCT core design using cyclic convolution and adder-based realization," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 14, no. 4, pp. 416–428, 2004.

[44] D. Gong, Y. He, Z. Cao, E. Inc, and C. Fremont, "New cost-effective VLSI implementation of a 2-D discrete cosine transform and its inverse," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 14, no. 4, pp. 405–415, 2004.

[45] M. Sima, S. D. Cotofana, J. T. J. van Eijndhoven, S. Vassiliadis, and K. A. Vissers, "An 8x8 idct implementation on an fpga-augmented trimedia," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2001.

[46] H. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-Complexity Transform and Quantization in H. 264/AVC," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 598–603, 2003.

[47] ——, "Low-complexity Transform and Quantization with 16-bit Arithmetic for H. 26L," in *Proc. IEEE Int. Conference on Image Processing (ICIP02)*, 2002.

[48] S. Gordon, D. Marpe, and T. Wiegand, "Simplified use of 8x8 transforms–Updated proposal & results," *Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, doc. JVT–K028, Munich, Germany, March*, 2004.

[49] 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.

[50] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Implementing the 2d wavelet transform on simd-enhanced general-purpose processors," *IEEE Transactions on Multimedia, Vol. 10, No. 1*, pp. 43–51, January 2008.

[51] ——, "Performance Impact of Misaligned Accesses in SIMD Extensions," in *Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2006)*, November 2006, pp. 334–342.

[52] M. Alvarez, E. Salami, A. Ramirez, and M. Valero, "Performance Impact of Unaligned Memory Operations in SIMD Extensions for Video Codec Applications," in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, 2007, pp. 62–71.

[53] H. Chang and W. Sung, "Efficient vectorization of simd programs with non-aligned and irregular data access hardware," in *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, 2008.

[54] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," in *Proceedings of the 11th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2005, pp. 340–351.

[55] 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.

[56] 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.