Programming techniques for real time software implementation of optimal edge detectors:

- a comparison between state of the Art DSPs and RISC architectures -

Frantz LOHIER[1,2], Lionel LACASSAGNE[1,2], Patrick GARDA[1]

[1]Laboratoire des Instruments et Systèmes
Université Pierre et Marie Curie
4 place Jussieu - B.C. 252
75252 Paris Cedex 05, France
lionel | lohier | garda@lis.jussieu.fr

[2]Electronique Informatique Applications
Bâtiment 4, Burospace
route de Gisy
91571 Bièvres Cedex, France
eia@wanadoo.fr

ABSTRACT

This paper presents the real time implementations of the Canny-Deriche optimal edge detectors on RISC and DSP processors. For each type of architecture, the most leading optimization techniques are described. A comparison is then made between DSP and RISC processing speeds.

INTRODUCTION

Canny-Deriche operators have asserted themselves to the edge detection field which stands as a fundamental component of image processing and computer vision.

The main drawback is the prohibitive computational power they require. It has led people to design dedicated hardware implementations (FPGA, ASIC) to achieve the real time execution of these detectors. On the other hand, the still increasing performance of DSP and RISC calls into question the need for a dedicated architecture.

This paper shows that crafty software implementation of Canny-Deriche edge detectors may achieve real time execution on state of the art DSP and RISC processors. To achieve this goal, we introduce data parallelism techniques as well as flow optimization methods.

1 OPTIMAL EDGE DETECTORS

In this section we introduce optimal edge detectors and the derived filters we will use in the subsequent sections.

1.1 Canny's optimal filters

Canny's approach [1] consists in finding the optimal FIR filter which satisfies the three following constraints for a Heaviside input signal: good detection, good localization, low maximum multiplicity due to the noise.

Deriche [3], using Canny's approach, has looked for an IIR filter which satisfies the same constraints. He got the same differential equation, but while changing the conditions at the limits, he obtained, for the Canny's performance index, an improvement of 25%.

Deriche's operators are used for two important methods of edge detection. The first is based on the gradient maxima, the second, on the laplacian's zero crossings. The state of the art methods combine both of them. In this paper we will focus on the gradient method. The implemented operators are those proposed by Deriche in [3].

1.2 Deriche's gradient

The horizontal derivative is the result of a smoothing in the vertical direction followed by a derivation in the horizontal direction. Respectively, the vertical derivative is based on those transposed directions. The smoothing operator is the sum of a causal and an anti-causal filter:

$$y_1(n) = k\left[x(n) + e^{-a}(a-1)x(n-1)\right] + 2e^{-a}y_1(n-1) - e^{-2a}y_1(n-2)$$

$$y_2(n) = k\left[e^{-a}(a-1)x(n+1) - e^{-a}(a-1)x(n+2)\right] + 2e^{-a}y_2(n+1) - e^{-2a}y_2(n+2)$$

$$y(n) = y_1(n) + y_2(n),\ k = \frac{(1-e^{-a})^2}{1+2ae^{-a}-e^{-2a}}$$

so as the derivation filter:

$$y_1(n) = -kx(n-1) + 2e^{-a}y_1(n-1) - e^{-2a}y_1(n-2)$$

$$y_2(n) = kx(n+1) + 2e^{-a}y_2(n+1) - e^{-2a}y_2(n+2)$$

$$y(n) = y_1(n) + y_2(n),\ k = (1-e^{-a})^2$$

1.3 FGL's optimizations

The first optimization consists in modifying the filter's equation in order to reduce its computational complexity. The major breakthrough was achieved by Frederico Garcia-Lorca in [4]. He introduced a new decomposition of Deriche's IIR filter with a lower computational burden. Deriche's smoother is replaced by the cascade of a causal and an anti-causal filter in each direction. FGL provides the same Canny's performance index than Deriche thanks to the 2 passes of the following first order equations:

Causal smoothing filter : $y(n) = (1-g)x(n) + gy(n-1) = Bx(n) + Ay(n-1)$

Anti-causal smoothing filter : $y(n) = (1-g)x(n+1) + gy(n+1) = Bx(n+1) + Ay(n+1)$

with $g = e^{-a}, a \geq 0, B = (1-g), A = g$.

Deriche's derivative filter is computed by the convolution of the following 2×2 kernels, followed by the L1 magnitude:

$$\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} -1 & -1 \\ +1 & +1 \end{bmatrix}$$

We can even get a lower complexity by replacing the two passes of the FGL's smoother by a single pass of a 2nd order filter which is the square of the first order filter:

$$y(n) = (1-g)^2 x(n) + 2gy(n-1) + g^2 y(n-2) = b_0 x(n) + a_1 y(n-1) + a_2 y(n-2)$$
$$y(n) = (1-g)^2 x(n) + 2gy(n+1) + g^2 y(n+2) = b_0 x(n) + a_1 y(n+1) + a_2 y(n+2)$$
$$\text{and } b_0 = (1-g)^2, a_1 = 2g, a_2 = -g^2 .$$

We emphasize the fact that the sum of the coefficients are normalized. The following table shows that the Garcia-Lorca's operator complexity is half that of Deriche's:

Deriche			**FGL 1st order**			**FGL 2nd order**		
			& Gradient					
MUL	ADD	ABS	MUL	ADD	ABS	MUL	ADD	ABS
28	29	2	16	15	2	12	15	2

Table 1: filters' complexity

FGL's first and second order filters being qualitatively equivalent, the next section introduces the first order implementation on Texas Instrument's C80. The second order is preferred for the implementation on TI's new C62 DSP and for the RISC architectures with the aim of maximizing the performance (in section 3 and 4).

2 DSP IMPLEMENTATION ON THE C80

In this section, we present architectural optimization techniques which can lead to real-time execution on the C80 processor. We review its main characteristics to better understand the algorithmic mapping that we choose to target on the C80's 4 advanced DSPs. For those processors, we use the assembly language and the fixed point representation. Finally, we review DMA implications for FGL's algorithm.

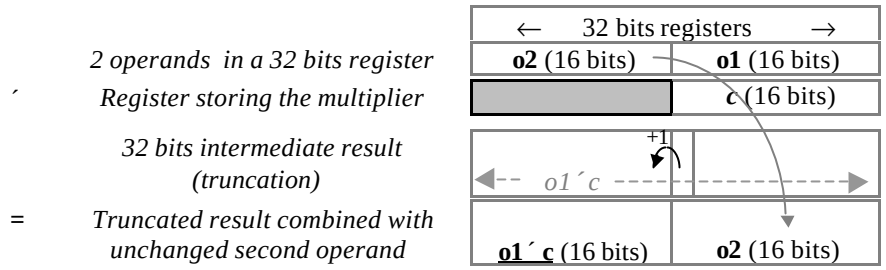
2.1 C80's VLIW advanced DSPs

TI's C80 is capable of **2,5 Gops** at 60 Mhz thanks to a RISC floating point processor (the master processor) and to 4 advanced 32 bits fixed point VLIWs (or Parallel Processor, PP), all gathered in a single chip. Processors access internal memory (cache or general purpose static memory) through a crossbar thus inducing minimum contention. An advanced DMA controller performs all processors data transfers between off-chip and on-chip memory where processing takes place. Its maximum bandwidth is of 480 Mbytes/s at 60 Mhz. Since the C80's performance is essentially based on PP's VLIW instructions and their ALU capabilities, we now detail their features.

VLIW instructions

Each PP can perform 4 sets of operations in parallel in a single cycle VLIW instruction: the multiply hardware, the general ALU hardware and 2 memory load/store address units.

- ***PP's multiply hardware***: among many possible combinations, two cycles are necessary for two rounded and truncated multiplies between two 16 bits integers and two fixed point constants. We depict the first cycle mecanism:



  When this result is used as the operand of another run of this multiplication mechanism with another multiplier coefficient c', we gain o2×c and o1×c' in a 32 bits register.

- ***PP's ALU***: It can combine algebraic and arithmetic operations in a single cycle. Its general equation is of the form $A \,\&\, f(B,C) \pm g(B,C)$ where A, B and C correspond to the raw ALU input ports (& stands for bitwise logical AND). B can result from a register left rotate ($B \equiv register \setminus\setminus rotate\_amount$) and C can be used to generate a mask of a specified number of bits ($C \equiv \%n = 2^n - 1$). f and g summarize independent boolean functions.

- ***PP's address units***: two independent powerful address units can access data without contention in a direct or indexed way. Moreover arithmetic (+/-) is allowed on any pointer register before or after the memory access.

Hardware loop controllers

To avoid handling a loop counter and the associated compare and branch instructions, each PP features a hardware mechanism to cope with up to 3 nested loops.

2.2 SPMD mapping on the PPs

To achieve real-time performance with FGL, we choose to parallelize the processing using the PPs and a SPMD data partitioning scheme.

- Mapping the first order FGL IIR filter:

We achieve **2 cycles per pixel** using the multiplication mechanism we former depicted. A and B coefficient range in]0,1.0]. For the fixed point representation, we scale those constants to use all the available 16 bits dynamic to code the decimals. With C defining either A or B, its fixed point representation, *c*, is $c = (2^{16} - 1) \times C$.

PP's registers file is "multi-ported". Registers can be used both as operand and destination of operations. Registers are modified at the end of all the parallel operations. The following pseudo code gives the initialization phase and the two cycles kernel loop (',' stands for sequential, '||' for parallel):

Init.	: x ← X[0] || τ ← 0, θ ← x×B || x ← X[1]
1st cycle	: θ ← x×B || y ← θ+τ || x ← X[i ← i+1]
2nd cycle	: τ ← y×A || Y[j ← j+1] ← y

with τ and θ being intermediate results, X[i] the input data, Y[i] the output ones.

We detail the ALU features where the rotation and masking are used in accordance with the first cycle of the fixed point multiply:

Init.	: x ← X[0] || τ ← 0, θ ← x×B || x ← X[1], θ ← θ\\16
1st cycle	: θ ← x×B || y ← (θ & %16)+(τ\\16 & %16) || x=X[i ← i+1]
2nd cycle	: τ ← y×A || θ ← θ\\16 || Y[j ← j+1] ← y

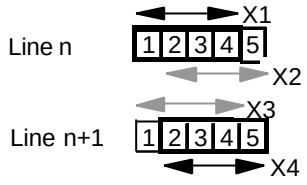
- Gradient magnitude computation:

We achieve **12 cycles for 4 pixels** per PP thanks to their ALU SIMD processing ability we now present. The general ALU's equation can operate on a 32 bits register split as four separate bytes. A 4 bits multiple status flag (*mf[1…4]*) is maintained for each split operation. In addition, the C port can be used to expand a mask from register *mf*:

$$C \equiv \underset{i:0\ldots4}{@mf} = \begin{cases} 0, i = 0 \\ @\, mf_{i-1} \mid (0xFF \setminus\setminus (8 \times (i-1)), mf[i] = 1 \end{cases}^{1}$$

[1] '|' stands for bitwise logical OR.

This introduces how we can perform four 8 bits absolute differences in just two cycles. We use the following pixel organization $\begin{bmatrix} x1 & x2 \\ x3 & x4 \end{bmatrix}$ and its SIMD counterpart $\begin{bmatrix} X1 & X2 \\ X3 & X4 \end{bmatrix}$, packing four pixel values as explained with the following figure:



and perform, using θ=X2+X4 and τ= X1+X3:

1st cycle : SIMD subtraction : SplitDiffs $\leftarrow \theta - \tau$ (we set *mf* base on the carry)
2nd cycle : gain absolute values: **SplitAbsDiff ¬ (SplitDiffs & *@mf*) | (-SplitDiffs & ~*@mf*)**[2]

Here the expander serves as a selector between values of interest.

Performance estimates

The following table summarizes the estimated performance without the I/O impact:

	Per PP		Tot. Cycles with 4 PPs
	Cycles/pixel	Num. pass	
FGL 1st order	2/1	8	4×512^2
Gradient	12/4	1	$12/16\times512^2$
		Total cycles	1245184
Raw duration at 60 Mhz (without transfer)			**20 ms**

Table 2: C80 raw figures for a 512^2 image

DMA transfer during calculation

To reduce the I/O impact, while processing, the DMA brings the completed data back to external memory and downloads new data to be processed. We use the double buffering technique to make sure no contention occurs between the DMA and the processor when both access the same internal memory bank.

	Time interval →			
	1	2	3	4
Memory Bank 1	**1. DMA: Ext. ® Int.**	Processing *(input = output)*	**1. DMA: Int. ® Ext.** **2. DMA: Ext. ® Int.**	Processing *(input = output)*
Memory Bank 2		**1. DMA: Ext. ® Int.**	Processing *(input = output)*	**1. DMA: Int. ® Ext.** **2. DMA: Ext. ® Int.**

Table 3: The double buffering technique

[2] '~' stands for logical bitwise negate.

As the DMA stands as a shared resource among all the processors, DMA requests' contention can occur and stands as a bottleneck especially when processing is faster than the transfer duration. This last parameter much depends on the type of external memory that is interfaced with the c80 chip.

2.3 Conclusion

This program was implemented and benched on a 40 Mhz device with an external DRAM memory requiring 3 cycles per column DRAM for reads and 2 cycles for writes. 50 ms are needed to process a 512^2 image meaning that in our implementation, the DMA roughly represents 2/5 of the total duration. We conclude that a 60 Mhz device interfaced with synchronous DRAM may achieve real-time processing for image bigger than 512^2.

3 VLIW PROCESSING ON THE C62 ADVANCED DSP

VLIW mono processor appears as an alternative to the MIMD parallel processors such as the C80. After a short review of TI's new C62's architecture, the subsequent sections introduce two optimization techniques on which we base the assembly implementation of FGL's second order edge detector in fixed point representation.

TI's C62 is a VLIW DSP capable of 1.6 GIPS at 200 Mhz thanks to:

- a 12 levels VLIW instruction superpipeline with 5ns cycle
- 256 bits VLIW instruction coding 8 operations which can partially be executed sequentially or concurrently[3]
- two 16x16 multiplications, 1 shift, 2 adds, 1 branch, 2 loads/stores (with pointer modification) can be done in parallel thanks to 8 independent functional units
- Several DMAs leading to about 250 Mbytes/s peak.

The functional units are redundant and organized as 2 sets of 4 distinct blocks each capable of executing certain operations. Each side (set) has its own associated register file. Some operations allow one register operand to be connected to the other side's file. **One register cross path per side and per VLIW instruction is allowed** The following table summarizes the functional units and the delays for some operations of interest:

Opcode	Description	Delay	Description	Units per side			
				.L	.S	.M	.D
ADD/SUB	Add/Sub	1	Add/Sub	**X**	**X**		**X**
SHR	Signed shift right	1	Signed shift right		**X**		
B	Branch	6	Branch		**X**		
ADD2/SUB2	SIMD Add/Sub	1	SIMD Add/Sub		**X**		
ABS	Absolute Val.	1	Absolute Val.	**X**			
MPY	Multiply	2	Multiply			**X**	
LOAD/STORE		5					**X**

Table 4: Operations: delay and functional units

[3] Texas calls this the VelociTI™ technology.

The C6x features two split 16 bits adds/subtracts which individually allow two parallel additions of two 16 bits numbers (ADD2/SUB2), all in a single cycle.

3.1 Optimization techniques

Software pipelining is the main technique used to obtain performance from the C6x. Due to the different delays of operations in the pipeline (Table 4), we maximize the use of the different units by executing the largest number of parallel operations for the loop kernel even if they do not concern the same processing iteration. We then provide a prolog and epilog piece of code to guarantee that operations get properly time stamped in accordance with the kernel.

Loop unrolling is another efficient method that can be used to fill the VLIW units. Originally, this technique is intended to reduce the impact of loop handling. The contents of the loop is replicated *r* time and the loop upper bound is divided per *r*. Replicating the loop kernel induces more potential operations that can be executed in parallel.

3.2 C62's implementation

We use the C6x SIMD processing ability and the software pipelining technique that we detail to implement the gradient. To maximize performance, we use the same technique that we combine with loop unrolling to map the second order smoother.

- Gradient magnitude computation:

First of all we demonstrate that an intermediate result for the horizontal gradient calculation is shared between 2 adjacent positions of the mask. With the following pixel configuration $\begin{bmatrix} p1 & p2 \\ p3 & p4 \end{bmatrix} \begin{matrix} p5 \\ p6 \end{matrix}$, we have $\nabla_n \approx |p2 + p4 - (p1 + p3)| + |p3 + p4 - (p1 + p2)|$, with $k = p2 + p4$, the next gradient is $\nabla_{n+1} \approx |p5 + p6 - k| + |p4 + p6 - (p2 + p5)|$.

Assuming that *p1,p2* and *p3,p4* are loaded into two separate registers, we can take advantage of the SIMD ADD2 and SUB2 instruction to fasten the calculation. To combine the results, we use the shift right instruction (SHR) to access the upper part of the result and the **MPYHL** instruction (2 delay slots) that multiplies the upper 16 bits of a register with the lower part of another to extract the lower part (this stands for a masking operation).

We then follow a three step procedure which consists in drawing an algorithmic dependency graph, mapping this graph on the architecture and folding it up to software pipeline the code.

We start drawing the dependency graph of our calculus. No specific rules yet apply, we just try to limit register's file cross references as we know it stands as a constraint when assigning the nodes to the VLIW instructions. For each side and each height of the graph, we try to diversify the units the nodes use.

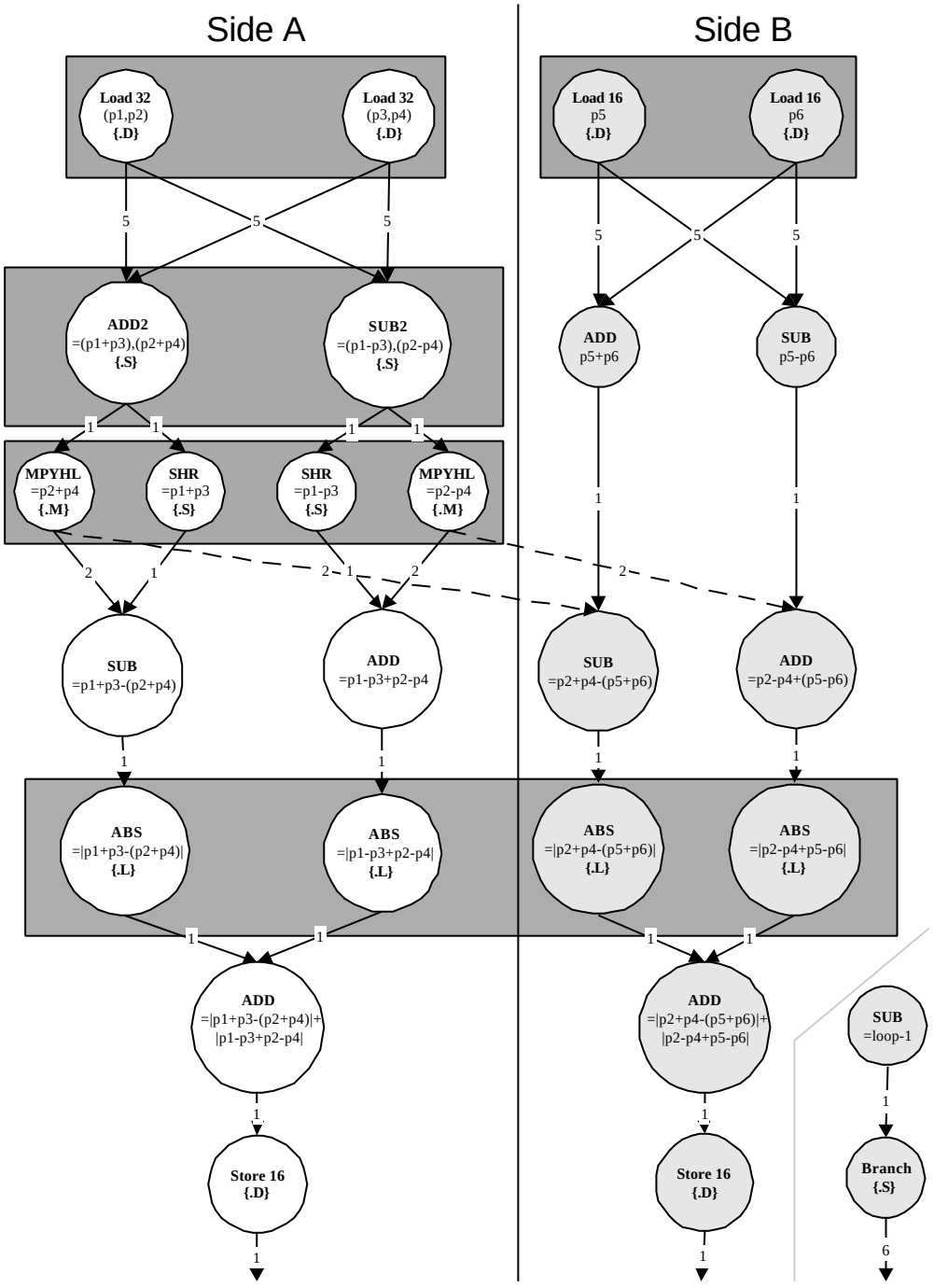


Figure 1: Algorithmic dependency graph with functional unit conflicts

For the gradient, we could obtain Figure 1. We have 26 nodes including the required operations to handle the loop, meaning that we will need at least 26/8=**4** VLIW instructions to code the kernel. We don't use any compare instruction for the loop since the branch can be conditioned by the loop counter at the branch instruction level.

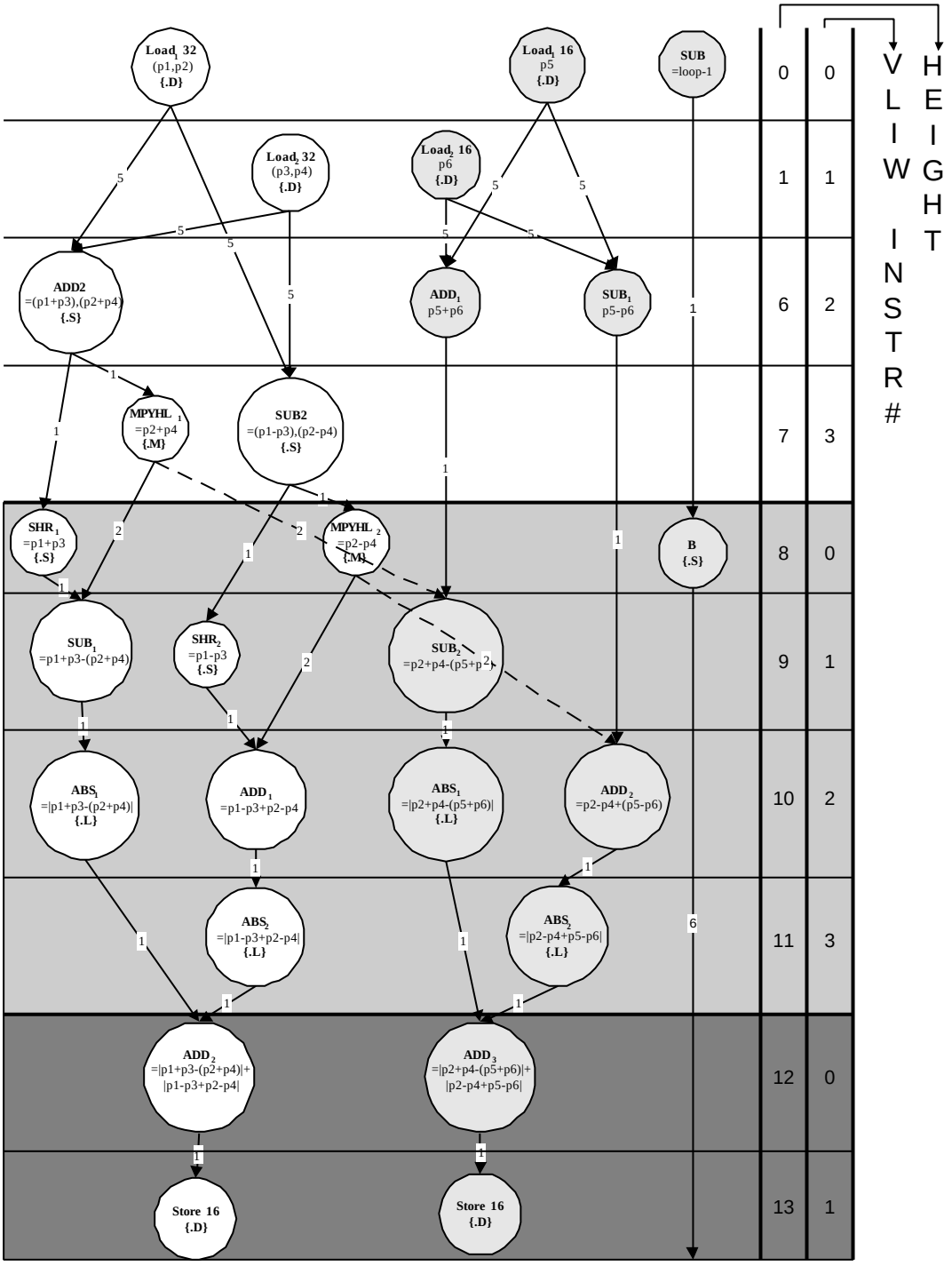


Figure 2: **The mapped dependency graph**

The brackets used to describe some nodes detail the units we already know the operations must be mapped to. The striped area underline contentions if the nodes are executed in parallel (**one occurrence of a functional unit per side**).

The second phase constitutes one step towards operations clustering into VLIW instructions. We stretch the dependency graph so as to avoid concurrency regarding the functional units and to take into account the nodes' delay (eventually by filling those delays with other nodes). In our example, the height of the graph is now of 14 nodes on Figure 2.

The next step of the procedure consists in folding up the mapped dependency graph so as to software pipeline the code. Let's use the theoretical 4 VLIW instructions (4 cycles) as a goal. We need to find the starting point for the fold up operation.

We start the calculation at cycle 6, after loading the first values. If we place the LOADs in the unit .D of the first 2 cycles of our 4 VLIW instructions loop, we get the result at cycle 2 of the next iteration of our instructions' set ((1+5) mod 4 = 2). Each subsequent operations in the graph will be located at instruction *n* where *n* is the node's height modulus the loop kernel size (4).

The mapping of operations 0 to 7 is straight forward. Regarding side B, the dependency graph allows us to choose when to execute some nodes (ADD_1 can occur either at cycle 6 (VLIW instruction 2), 7 (3) or 8 (0)). In addition multiple VLIW's units can sometimes be targeted to receive the same operation (as for ADD or SUB).

The following table reviews the mapping of nodes 8 to 11 over the 4 VLIW instructions. The last column underlines the possible targets for the nodes 0 to 11 over side B:

VLIW Instr.	Unit	Side A		Side B		
		Mapped nodes	Possible slots for nodes [8-11]	Mapped nodes	Possible slots for nodes [8-11]	Possible slots for nodes[0-11]
0	.L					¬ ADD_1/SUB_1
	.S	**SHR_1**	⇐. SHR_1			¬ ADD_1/SUB_1
	.M	**$MPYHL_2$**	⇐. $MPYHL_2$			
	.D	$Load_1$ 32		$Load_1$ 16		
1	.L	**SUB_1**	⇐. SUB_1		¬ SUB_2	¬ SUB_1
	.S	SHR_2			¬ SUB_2	
	.M					
	.D	$Load_2$ 32		$Load_2$ 16		
2	.L	ABS_1	⇐ABS_1	**ABS_1**	⇐. ABS_1	¬ ADD_1/SUB_1
	.S	ADD2		B		
	.M					
	.D	**ADD_1**	⇐. ADD_1	**ADD_2**	⇐ADD_2	¬ ADD_1/SUB_1
3	.L	**ABS_2**	⇐. ABS_2	**ABS_2**	⇐. ABS_2	
	.S	SUB2				¬ ADD_1/SUB_1
	.M	$MPYHL_1$				
	.D					¬ ADD_1/SUB_1

Regarding side A, all 8-11 operations overlap with those of cycle 0-7 without conflict (either the operation's unit is fixed or only 1 compatible unit is left). For side B, the folding up operation reduces the choice of assignments for operations 0-11 (as for unit .L

of cycle 2 of the loop kernel that's locked by ABS_1). We place the branch instruction at cycle 2 for it to take effect at the end of the next 4 cycle loop (delay of 6 cycles).

Left are operations of cycle 12-13 where we encounter a conflict for the STORE operations. Regarding side A, only the .D unit of cycle 3 is left and no restriction apply for using it.

VLIW Instr.	Unit	Side A		Side B		
		Mapped nodes	Possible slots for nodes [12-13]	Mapped nodes	Possible slots for nodes [8-13]	Possible slots for nodes [0-11]
0	.L	**ADD_2**	⇐ ADD_2	**ADD_3**	¬ *ADD_3*	¬ *ADD_1/SUB_1*
	.S	SHR_1		**ADD_1**	¬ *ADD_3*	¬ *ADD_1/SUB_1*
	.M	$MPYHL_2$				
	.D	$Load_1$ 32		$Load_1$ 16		
1	.L	SUB_1		**SUB_2**	¬ *SUB_2/SUB_3*	¬ *SUB_1*
	.S	SHR_2		**SUB_3**	¬ *SUB_2/SUB_3*	
	.M					
	.D	$Load_2$ 32	⇐ Store 16	$Load_2$ 16	⇐ Store 16	
2	.L	ABS_1		ABS_1		
	.S	ADD2		B		
	.M					
	.D	ADD_1		ADD_2		
3	.L	ABS_2		ABS_2		
	.S	SUB2		**SUB_1**		¬ *ADD_1/SUB_1*
	.M	$MPYHL_1$				
	.D	**Store 16**		**Store 16**		

Finally we complete the assignment of all the possible left combinations. We also added the loop counter modification operation in a free slot. This procedure allows us to achieve **4 cycles for 2 output** 16 bits values with 16 bits inputs.

- Mapping the second order smoother:

Using the loop unrolling technique combined with software pipelining, we **achieve 8 cycles for 4 output** values.

Drawing the dependency graph of the second order smoother (Figure 3), we learn that at least 4 cycles per output are needed to map the equation. For each cycle, few of the functional units are used and we would like to launch several calculations of *y(n)* in parallel but the *y(n-1)* dependency stands as a strong constraint.

We expand the equation in order to delay the *y(n-1)* dependency:

$$y(n-1) = b_0 x(n-1) + a_1 y(n-2) + a_2 y(n-3)$$

thus,

$$y(n) = b_0 x(n) + b_0 a_1 x(n-1) + (a_1^{\,2} + a_2) y(n-2) + a_1 a_2 y(n-3)$$

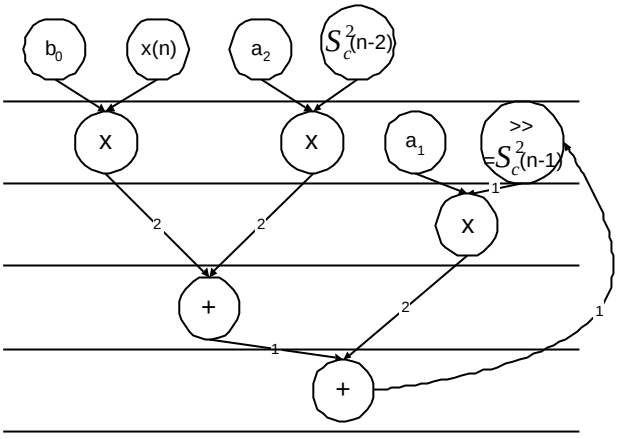


Figure 3: Dependency graph of FGL's second order smoother

by defining new coefficients:

$$b_0 = (1-g)^2, b_1 = 2g(1-g)^2, a'_2 = 3g^2, a_3 = -2g^3$$

we gain

$$y(n) = b_0 x(n) + b_1 x(n-1) + a'_2 y(n-2) + a_3 y(n-3)$$

Unrolling this new equation 4 times leads to the following:

$$\begin{cases} y(n) = b_0 x(n) + b_1 x(n-1) + a'_2 y(n-2) + a_3 y(n-3) \\ y(n+1) = b_0 x(n+1) + b_1 x(n) + a'_2 y(n-1) + a_3 y(n-2) \\ y(n+2) = b_0 x(n+2) + b_1 x(n+1) + a'_2 y(n) + a_3 y(n-1) \\ y(n+3) = b_0 x(n+3) + b_1 x(n+2) + a'_2 y(n+1) + a_3 y(n) \end{cases}, \text{ and } \begin{cases} y_0 = b_0 x_1 + b_1 x_0 + a'_2 y_{-2} + a_3 y_{-3} \\ y_1 = b_0 x_2 + b_1 x_1 + a'_2 y_{-1} + a_3 y_{-2} \\ y_2 = b_0 x_3 + b_1 x_2 + a'_2 y_0 + a_3 y_{-1} \\ y_3 = b_0 x_4 + b_1 x_3 + a'_2 y_1 + a_3 y_0 \end{cases}$$

with $x_i = x(n+i+1)$ and $y_{\pm i} = y(n \pm i)$.

Focusing on y_0, 8 cycles are necessary to get the results as described in the following table:

VLIW Instr.#	Launch	Result
0	$b_0 \ddot{A} x_1$	
1	$a_3 \ddot{A} y_{-3}$	
2	$b_1 \ddot{A} x_0$	$b_0 \times x_0$
3	$a'_2 \ddot{A} y_{-2}$ $b_0 \times x_0 \mathring{A} a_3 \times y_{-3}$	$a_3 \times y_{-3}$
4	$b_0 \times x_0 + a_3 \times y_{-3} \mathring{A} b_1 \times x_1$	$b_1 \times x_1$ $b_0 \times x_0 + a_3 \times y_{-3}$
5	$y_0' = b_0 \times x_0 + a_3 \times y_{-3} + b_1 \times x_1 \mathring{A} a'_2 \times y_{-2}$	$a'_2 \times y_{-2}$ $b_0 \times x_0 + a_3 \times y_{-3} + b_1 \times x_1$
6	$y_0' >> p$	y_0'
7		y_0

Our goal is to launch the 4 iterations in parallel using 8 cycles (software pipelining the code). If we launch 1 equation's calculus every 2 cycle we make sure that we have enough resources, especially regarding the multiplications (2 per VLIW instruction); see Table 5. At the same time we make sure that the *y(n-2)* and *y(n-3)* dependencies are satisfied (using Table 6). With *n:0..N*, the unrolled bounds are *k:0..N/4,n=n+4:*

VLIW Instr.#	VLIW resource → y_0	y_1	y_2	y_3
0	×			
1	´ **y_{-3}**			
2	×	×		
3	´ $\underline{y_{-2}}$ +	´ **y_{-2}**		
4	+	×	×	
5	+	´ $\underline{y_{-1}}$ +	´ **y_{-1}**	
6	>>	+	×	×
▼ 7	y_0	+	´ $\underline{y_0}$ +	´ **y_0**

Table 5: Mapping FGL 2

Unrolled loop	*y(n-2)*	**y(n-3)**
$y_{0,k}$	$y_{-2,k}=y_{2,k-1}$	$y_{-3,k}=y_{1,k-1}$
$y_{1,k}$	$y_{-1,k}=y_{3,k-1}$	$y_{-2,k}=y_{2,k-1}$
$y_{2,k}$	$y_{0,k}=y_{0,k}$	$y_{-1,k}=y_{3,k-1}$
$y_{3,k}$	$y_{1,k}=y_{1,k}$	$y_{0,k}=y_{0,k}$

Table 6: FGL 2 dependencies

Clearly, the gray part of the table stands as a sequence of operations that started the previous loop iteration (*k-1*) and that continue the current *k* iteration. All the expected results are available when we need them.

We won't review the complete mapping (at the assembly level) but instead we summarize the raw performance with the following table:

	Cycles/pixel	Num.pass	Tot cycles
FGL 2nd order	8/4	4	8×512^2
Gradient	4/2	1	2×512^2
Total cycles for FGL 2 and Gradient			2621440
Raw duration at 200 Mhz (without transfer)			**13 ms**

Table 7: C62 raw estimation for a 512^2 image

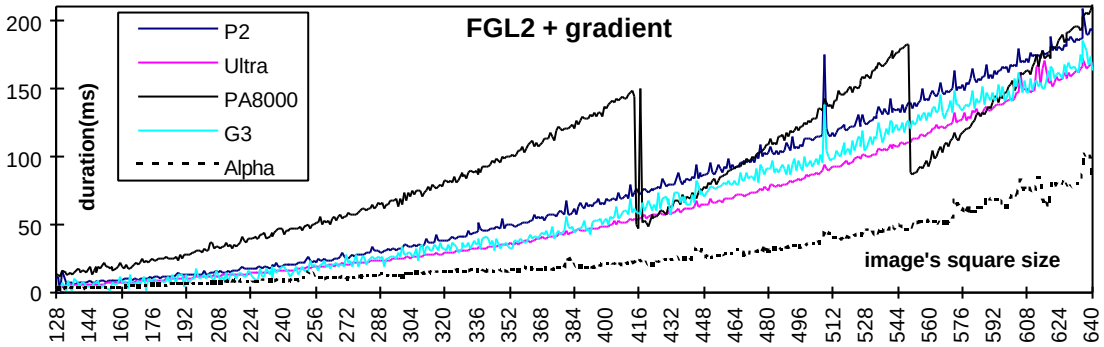
3.3 Conclusion

Using appropriate optimization techniques, we estimate very good raw performances. Yet we have no precise figures on the I/O impact which can still be reduced using the double buffering technique.

C62 single VLIW instruction pipeline stands as a true competitor to the latest RISC superscalar processors. Instructions' dynamic assignment and scheduling mechanisms we find on those architectures (especially with out-of-order kernel) are replaced by static ones that require more intelligent compilers.

4 COMPARISON WITH RISC IMPLEMENTATIONS

The same algorithm was coded in C using **floating point** for the smoother. Floating point calculation isn't meant to disadvantage RISC architectures since it appeared to run faster than for the fixed point representation. As all the benched architectures had separate pipelines for integer or floating operations and assuming that the integer pipeline is used to handle the loop counter and the pixels' indices, saying that the resources are better used when both pipelines are used stands as a rough explanation. We unrolled FGL's second order smoother 3 times and benchmarked various architectures on various image sizes. The following table summarize the results which we detail in [5]:



Architecture	Freq. (Mhz)	L1 cache size (Kbytes)	L2/L3 cache size (Kbytes)	RAM	Gradient for 512^2	FGL2+Grad. for 512^2	Real time's size FGL2+Grad.
HP PA8000	180	2 Mbytes	0	64	73 ms	135 ms	241^2
Sun Ultra2	300	16+16	512	64	18 ms	93 ms	372^2
Intel PII 300	300	16+16	512	32	25 ms	115 ms	325^2
Alpha 21164	625	8+8	96/4 M bytes	128	10 ms	40 ms	516^2
Apple G3	266	32+32	512	64	24 ms	97 ms	364^2

5 CONCLUSION

Through what can be considered a novel approach for edge detection, this article provides qualitative and quantitative elements to compare different processors' architectures. Furthermore, in the very competitive race for computational power, it is not an easy task to obtain liable figures.

The qualitative arguments we have put forward gather the hierarchical features we used for FGL's algorithm as well as the programming style and the optimization methods we introduced. If we come to compare both detailed DSPs, we believe the C80 is more flexible because of its four different sets of independent 64 bits VLIW instructions running concurrently. This point of view is emphasized with PPs' algebraic and logical ALU that is optimized for video processing functions. Yet this flexibility leads to a greater complexity. The C6x core and its 256 bits VLIW instruction is simpler and its higher operational frequency makes it a true competitor for the C80. However, this depends on the algorithm, as for the gradient calculation which runs almost as fast on

both architectures (the ratio of frequencies is almost balanced by the PPs' SIMD capabilities).

Whereas pre-built operations are used for fast processing on the C80, software pipelining and loop unrolling stands as generic methods the user or the compiler needs to cope with to achieve good performances on the C62. Observe that the C80's hardware loop controllers replace the unrolling technique.

Speaking of RISC's C floating point implementation, we generally encounter lower performances than on the two DSPs but we emphasize that it also requires much less effort and it allows for a better code reusability. Taking advantage of SIMD instructions embedded with some RISCs (like Intel's MMX technology) would reduce the performance's gap with the loss of the three enumerated advantages.

Focusing on the quantitative arguments, we have shown through estimations, that software implementation in assembly language on advanced fixed point DSPs achieves real time execution for image size larger than 512^2. In section 4, we have shown through benchmarks that C software implementation on floating point RISC processors achieve real time execution of optimal filters for image sizes up to 516^2 on DEC Alpha processors and up to about 350^2 on others.

More specifically, 512^2 images may be processed at 25 frames/s on a 625 Mhz DEC Alpha, 30 frames/s with a 60 Mhz C80 and at 75 with a 200 Mhz C62 which, however, does not consider the impact of memory transfers. These results outperform state of the art implementation of optimal edge detectors on DSPs [6], [7] and even FPGAs [8].

THANKS

Texas Instrument - *Dec alpha*: **DEC France**: Mr Decamps, Verdun, Denis, **CCR laboratory**: Mr. Racine – *Intel:* **Gateway France**: Mr Gontier, Arikessavane – **Intel France**: Mr Colin- *HP:* **HP France**: Mr Missélis, Coslovi, Robson – *Apple:* **Apple France**: Mr Stransky- *Sun:* **IEF laboratory**: M. Mérigot

REFERENCES

[1] J.F. Canny. *A computational Approach to Edge Detection*, IEEE Trans. on PAMI (IEEE), vol 8,6 p79-698 (1986).

[2] R. Deriche. Fast Algorithms for low level-vision, IEEE Trans. on PAMI, vol 12,1 (1990).

[3] F. Garcia-Lorca. *Filtres recursifs temps reel pour la detection de contours : optimisations algorithmiques et architecturales.* French Phd thesis (Orsay University 1996).

[4] J.L. Hennessy, A. Patterson. *Computer Architecture. A quantitative approach.* 2nd edition.

[5] L. Lacassagne, F. Lohier. *Real time execution of optimal edge detectors on RISC and DSP processors*, 23rd International conference on Acoustics, Speech and Signal Processing, 1998

[6] Y. Kim, Washington University http://icsl.ee.washington.edu/projects/iclib.

[7] J.P. Derutin, B. Besserer, T. Tixien, A. Klickel. *A parallel Vision Machine: Transvision*, CAMP91, Computer and Machine Perception, December 1991

[8] Sarifuddin. *Implantation sous forme de circuit spécialisé d'un algorithme de détection de contours multi-échelles*. French Phd Thesis (Montpellier II University 1996)

	VLIW instruction			
	ALU	Mult.	@ Unit 1	@ Unit 2
Cycle 1	$S^1_{c,k} = AS^1_{c,k-1} + Bx_{k-1} \Rightarrow r3 = r1 + r2$	$r2 = Bx_{k-1}$	$x_k \leftarrow x(n+1)$	X
Cycle 2	X	$r1 = AS^1_{c,k}$	$S^1_c(n \leftarrow n+1) \leftarrow S^1_{c,k} = r3$	X

Grad. ∇_n

Grad. ∇_n