A GENERIC METHODOLOGY FOR THE SOFTWARE MANAGING OF CACHES IN MULTI-PROCESSORS DSP ARCHITECTURES

Application to the real-time implementation of low level image processing on the TMS320C80

Frantz LOHIER[1,2], *Lionel LACASSAGNE*[1,2]
[1]Electronique Informatique Applications
Burospace, Bâtiment 4,
Route de Gisy
91571 Bièvres Cedex, FRANCE
eia@wanadoo.fr

Pr. Patrick GARDA[2]
[2]Laboratoire des Instruments et Systèmes
Université Pierre et Marie Curie
4 place Jussieu - B.C. 252
75252 Paris Cedex 05, FRANCE
lohier | lacassagne | garda@lis.jussieu.fr

ABSTRACT

This article introduces a novel software engineering methodology designed for the real-time execution of low-level image operators running on multi-processors DSP architectures. We detail the results we gained while implementing our approach on the TMS320C80, a shared memory multi-processors architecture [1]. Our contribution compares to other existing C80's image processing libraries [2][3] in terms of ***genericity***, ***flexibility***, and ***performance*** improvement. More specifically, generic mechanisms allow to address various operator's requirements as well as expanding them using a standard framework. Our approach is flexible enough to allow for the dynamic composing of concurrent and reconfigurable processing chains thanks to a modular library implementing basic operators. Processing chains work on various image sizes and with any number of processors. Above all, our methodology permits performance improvement by enhancing data locality.

1. INTRODUCTION

Whereas any modern RISC architecture includes a data cache, general purpose internal data memory combined with DMA co-processor(s) often stands as a more predictable and efficient alternative for the DSPs. However, DMA co-processors are getting more and more complex and now feature sophisticated multi-dimensional data transfers like for the C80 [1] or the C6X. Whereas programming them becomes more difficult, software tools and techniques are lacking. This paper introduces an original methodology for the software managing of caches based on a multi-dimensional DMA and targeting for multi-processors DSP architectures. Seeking the real-time execution of *generic* low-level image processing algorithms has led us to formalize basic entities we describe in the next section. Then in section 3, we introduce how entities' instances are combined to implement *flexible* processing chains. Finally, section 4 details some *performances* gained with our novel C80's library.

2. GENERIC METHODOLOGY

Our methodology defines a software engineering framework built upon well defined entities and rules to manipulate and extend them. We first review these entities.

2.1 Buffers

In our image processing context, a buffer stands as a *ROI* (Region Of Interest) with which 8 parameters are associated including the width W_{ref}, the height H_{ref}, the physical location (*@BUF*) and the width pitch W_{pitch}. W_{pitch} allows us to address any 2D image sub-area without requiring an extra transfer to isolate the region. Since direct external memory addressing is very costly on the C8X (10~15 cycles per access in the best case), the DMA is used to download a specified amount of data in internal data memory where processing can take place much faster (1 cycle per access). In a software manner, we need to manage a cache per buffer with which 2 more parameters are associated: the location of an internal memory area for caching purpose (*@CACHE[0]*) and the size *S* of this area. Temporally speaking, if we now think of overlapping raw computation with the transfer of the next bunch of data to be treated, we need to implement the double buffering technique which requires a second cache location (*@CACHE[1]*). This approach is meaningful thanks to the C8X's crossbar which prevents contention between processing and transfer when distinct internal memory banks are used. The last parameter, σ, stands as the number of processors used to partition buffer's data towards an SPMD processing scheme.

2.2 Processing node

A processing node is a function executing one or several iterations of an algorithmic operator. A sequence of nodes implements a processing chain which consume and produce data from and to buffers. Nodes execute on the C80's advanced DSPs or PPs for Parallel Processors and are written in assembly language that is mandatory to achieve reasonable performance.

2.3 The processing template

The processing template runs on each PP and aims at encapsulating the synchronization and handling of DMA data transfers with the launching of nodes.

2.4 Data templates

Data templates or templates are entities associated with operators. They gather parameters describing the input and output structuring element for a processing node. We now give more insight on these parameters.

- **Basic parameters and optimization issues**

A $n \times m$ convolution mask would stand as the input template of a convolution node ($W_t=n$, $H_t=m$) whereas 1×1 would correspond to the output geometry. The input geometry will allow us to ensure that enough data are transferred in internal memory for at least one processing iteration. This can be dependent on the operator's implementation itself. Focusing on 2 major optimization techniques which are loop unrolling and the use of SIMD operations, the initial template is maximized to meet the fact that several horizontal or vertical iterations of an operator are computed in parallel. If two horizontal iterations of the convolution operator are treated in parallel, the input and output geometry are changed to $(n+1) \times m$ and 2×1 respectively.

To download an amount of data consistent with the vertical and horizontal quantity required to execute a new iteration of the optimized node in either of these directions, we introduce the horizontal and vertical steps. For our $n \times m$ convolution example, the horizontal and vertical steps are $W_s=1$ and $h_s=1$ respectively whereas they would be defined as $W_s=2$, $h_s=1$ in the optimized case. Of course, nothing prevents the implementation of a node to take into account the non multiple cases ($(W_{ref}-W_t) \bmod W_s \neq 0$) but on the one hand, the implementation gets more complicated thus longer (emphasized by the fact that PP's VLIW algebraic assembly is difficult [1]) whereas on the other hand, the increased granularity favors instruction cache contention which is likely to decrease performance. This last remark depicts a phenomenon which much depends on the overall granularity of a chain and on the number of cache blocks required by the processing template itself. As an example, a binary NOT node shows a 30% drop of performance on a 512^2 image when one instruction cache block is reloaded per DMA request.

- **Overlapping templates**

Overlapping templates defined by $W_t > W_s$ and/or $H_t > H_s$ require a deeper analysis. Since data are brought using multiple DMA requests, we need to cope with template's pixels that are shared between 2 requests (Figure 1). We can either reload the data or introduce a more subtle mechanism where the node is responsible for launching the next bunch of input data once the re-using of the previous data completed. The first approach is preferred because it simplifies the node (smaller granularity) as well as the implementation of such a mechanism in a multi-node context. Furthermore, since overlapping templates are generally associated with complex nodes (e.g. convolution) for which processing is usually slower than transfer, longer transfers have little impact on the global processing duration. The pair (W_t,W_s) and\or (H_t,H_s) are then used to calculate the amount of data reloaded per direction, set equal to W_t-W_s\H_t-H_s.

Another issue that arises when dealing with convolution node is linked with the overwriting of the center data by the convolution result. This renders the next iteration calculus erroneous. Considering the concurrent transfer of the next DMA request, an intuitive approach suggests using a third memory bank to output the results while triple buffering. Offsetting the result as shown with Figure 1 is a much simpler alternative.

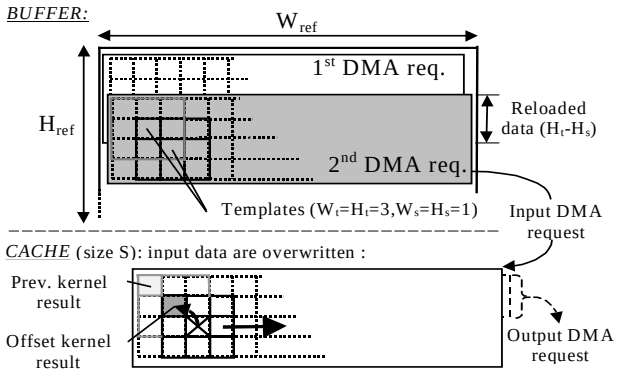


Figure 1: Data reloading and results offsetting

- **Sweeping internal and external data**

C80's DMA allows separate settings for the source and destination of a transfer. Thus we can efficiently de-correlate the way data are swept in the buffer compared to this in internal memory where data are organized linearly (to ease the linking of nodes). This feature is notably interesting for separable filters for which the same node is used whether the image is scanned vertically or horizontally. Moreover, we can think of color images for which buffer's data alternate the coding of different color components. To pinpoint the luminance component of a YUV buffer for example, we introduce 2 more parameters which are the data size D_s and the data pitch D_p. The multidimensional feature of the DMA allows practical data filtering and permits the same node to process interlaced or consecutive data. This discussion suggests the need for an external template geometry (upper-case $W_t \times H_t$, D_s, D_p, W_s, H_s) which stands as the external counterpart of a node's internal input\output template used when sweeping its associated input\output buffer. Internal input (^{I}) and output (^{O}) template parameters are designated in lower-case. We also define an internal data size and pitch (d_s,d_p) to allow reserving space between 2 data in internal memory. This simplify the implementation of nodes that expand the initial data size. The following table shows the diversity of templates configurations:

Node's name	**I**nput template: $w^I_t \times h^I_t, d^I_s, d^I_p, w^I_s, h^I_s$	**O**utput template: $w^O_t \times h^O_t, d^O_s, d^O_p, w^O_s, h^O_s$	Optimization
Histogram	1×1,1,1,1,1	None	
FGL 1st order IIR	2×1,1,1,1,1	1×1,1,1,1,1	
Binary NOT	4×1,1,1,4,1	4×1,1,1,4,1	SIMD
3x3 convolution	4×3,1,1,2,1	2×1,1,1,2,1	SIMD
Robert's gradient	5×2,1,1,4,1	4×1,1,1,4,1	SIMD+unrolling
Trunc.	4×1,2,2,8,1	4×1,1,1,4,1	

2.5 Building chains

Once nodes and buffers parameters are known, we can compose generic processing chains. This is based on a high level description of the chain. Then, our library features advanced mechanisms to parse the description while transparently generating all the DMA requests further used by the processing template. The next section details this important step.

3. FLEXIBLE DMA PROGRAMMING

A flexible implementation of the methodology requires to integrate numerous parameters with the aim of splitting the buffer's data in a SPMD fashion and deriving the DMA requests for all buffers and processors. Here, performance depends on a near optimal solution.

3.1 The basic constrained system

If we focus on the reload approach that's been implemented, the total amount *Q* of data to be transferred is a function of (β,γ) which corresponds to the combined number of horizontal and vertical occurrences of a template that can fit in a cache of size *S-a×b* with $\begin{cases} a = d_s + (w_t - 1)\times d_p, b = h_t, c = w_s, d = h_s \\ A = D_s + (W_t - 1)\times D_p, B = H_t, C = W_s, D = H_s \end{cases}$. The pair (β,γ) is the key towards the calculation of the total number of DMA requests for a buffer and hence, their intrinsic settings. (β,γ) must verify $(\Psi)=\begin{cases} ab+\beta((\gamma+1)cd)+\gamma ad < S \\ A+\beta C \le (W = A+C\lfloor (W_{ref}-A)/C \rfloor) \\ B+\gamma D \le (H = B+D\lfloor (H_{ref}-B)/D \rfloor) \end{cases}$ and hence yield a total amount of transferred data Q equal to:

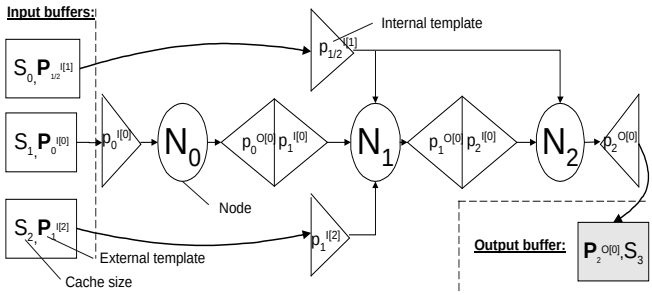
$$H\left\lceil \frac{W-A}{\beta C}-1\right\rceil (A-C)+\left(W+\left\lceil \frac{W-A}{\beta C}-1\right\rceil (A-C)\right)(B-D)\left\lceil \frac{H-B}{\gamma D}-1\right\rceil + W\times H\cdot$$

Q, is then used calculate the total number of DMA requests that we split among σ given processors. For simplicity, the former equations assume $D_p=D_s$, $A \le C$, $B \le D$. Experiments show that the less DMA requests the faster the processing which in terms of linear programming implies minimizing the objective function $S - ab + \beta cd + \gamma ad + \beta\gamma cd$. If we consider a chain gathering a single node connected to an input and output buffer, since our model is synchronous, we need to find (β,γ) that is compatible with both input and output templates and the related buffer parameters. We face a non-linear integer programming system which was proven to be NP-complete. We use a simple and fast heuristic which coarsely consists in getting β based on *W* and then deriving γ. The next section details how β and γ are calculated in a more general context.

3.2 Multi-node chains and synchronization constraints

Some nodes are directly associated with buffers and hence can be seen as *leaders* of *paths* gathering nodes that synchronize with them. On Figure 2, we enumerate 3 paths (N_0,N_1,N_2), (N_1,N_2) and (N_2) respectively associated with leading template $p_0^{I[0]}$, $p_1^{I[1]}$, $p_2^{I[1]} = p_1^{I[1]}$. In order to find (β,γ) per buffer, we first look for a *virtual template* which satisfies the minimum number of input data for a path. For each buffer linked with a leading node, we then deduce a couple (β,γ). Among all couples, we finally select the smallest β and γ.

Composing a chain can be seen as connecting an input template *k* with the output template of a previous node (*k-1*) keeping in mind that nodes may feature various data processing rates.



Template parameters: $\begin{cases} \mathbf{p}_k^{I[j]} = \left[w_{t_k}^{I[j]} \quad h_{t_k}^{I[j]} \quad d_{s_k}^{I[j]} \quad d_{p_k}^{I[j]} \quad w_{s_k}^{I[j]} \quad h_{s_k}^{I[j]} \right] \\ \mathbf{P}_k^{I[j]} = \left[W_{t_k}^{I[j]} \quad H_{t_k}^{I[j]} \quad D_{s_k}^{I[j]} \quad D_{p_k}^{I[j]} \quad W_{s_k}^{I[j]} \quad H_{s_k}^{I[j]} \right] \end{cases}$

Figure 2: A complex chain

This suggests satisfying independent synchronizing constraints for the vertical and horizontal amount of data required between 2 nodes. Hence, we need to solve 2 diopantine equations of the form $a_k^I + \beta^I \times c_k^I = a_{k-1}^O + \beta^O \times c_{k-1}^O$ solved thanks to Bezout's equality. Solutions appear as simultaneous congruences $\beta_k^I \equiv \phi_k'(\Phi_k')$, $\beta_{k-1}^O \equiv \phi_{k-1}(\Phi_{k-1})$ and are propagated such that they affect both the input and output template of a node. This keeps constant the possible difference of rate between the 2 node's templates. Then the next connection in the chain implies solving $a_k^I + (\beta_k^I \Phi_k + \phi_k)\times c_k^I = a_{k-1}^O + \beta_{k-1}^O \times c_{k-1}^O$ with (*k=k-1*).
This explains how virtual template parameters are estimated. For each path *c* linking *K* nodes with L being the leading node's index, we successively calculate with *k=[K-1..L]*, the pairs (β^I,β^O) and $(\gamma^I, \gamma^O) \in N^+$ from which we get Φ and Γ such that:

$$\begin{bmatrix} -c_{k-1}^O & c_k^I \Phi_{k-1} & a_k^I + c_k^I \phi_{k-1} - a_{k-1}^O \\ -d_{k-1}^O & d_k^I \Gamma_{k-1} & b_k^I + d_k^I \tau_{k-1} - b_{k-1}^O \end{bmatrix} \cdot \begin{bmatrix} \beta_{k-1}^O & \gamma_{k-1}^O \\ \beta_k^I & \gamma_k^I \\ 1 & 1 \end{bmatrix} = \vec{0}\cdot$$

Here, solutions for the height synchronization take the form of $\gamma_k^I \equiv \tau_k'(\Gamma_k'), \gamma_{k-1}^O \equiv \tau_{k-1}(\Gamma_{k-1})$ and the initial conditions are $\Phi_{K-1} = \Gamma_{K-1} = 1, \phi_{K-1} = \tau_{K-1} = 0$. Once β_L^O, γ_L^O are gained we then calculate (β,γ) starting with an estimation of the total number of internal templates that can fit in internal memory. For the *J* buffer(s) linked with internal template $p_L^{I[j]}$ *(j=[0..J-1])*, we evaluate $\xi = 1 + \lfloor (S/b_v - a_v)/c_v \rfloor$. Here, *S* is related to the buffer linked with current $p_L^{I[j]}$ and *v* indices describe the virtual template parameters gained with:

$$\begin{cases} w_t = w_{t_L}^{I[j]}, w_s = w_{s_L}^{I[j]}, d_s = d_{s_L}^{I[j]}, d_p = d_{p_L}^{I[j]}, h_t = h_{t_L}^{I[j]}, h_s = h_{s_L}^{I[j]} \\ a_v = d_s + (w_t + \phi_L \times w_s / d_p - 1)\times d_p, c_v = \Phi_L \times w_s \\ b_v = h_t + \tau_L \times h_s, d_v = \Gamma_L \times h_s \end{cases}.$$

Then, with A_v and C_v based on $\mathbf{P}_L^{I[j]}$ that translate from the same rules, we evaluate $\beta = (W_{ref} - A_v)/C_v$ and $\begin{cases} \theta = a_v + c_v(\xi - 1) \\ w = a_v + \beta \times c_v \end{cases}$ leading to $\gamma = 1 + \left\lfloor \frac{\theta \times b_v / w - b_v}{d_v} \right\rfloor$.
Here, equations are simplified and imply $w \ge \theta$. They must also satisfy (Ψ) set of inequalities (§3.1).

Symmetrically, the same overall approach is used to get (β,γ) couples from the output buffer point of view spreading the multiplicity constraints with k varying from 0 to L. Then, once we gain $(\beta_{min},\gamma_{min})$, the total number of DMA requests is calculated per buffer with $\beta = \phi_L^O + \beta_{\min}\Phi_L^O$ and $\gamma = \tau_L^O + \gamma_{\min}\Gamma_L^O$. The various DMA dimensions are then optimized such that the DMA request self-modify (pinpoints the next D_s data or the next image line with W_{pitch} and handles the internal double buffering that *"ping-pong"* on the *2 @CACHE* addresses). This all happens without intervention from the processing template which greatly favors instruction cache coherency.

4. PERFORMANCE RESULTS

This section presents the performances we gained with 3 sample processing chains. We used a sub-optimal C80 hardware configuration based on an external DRAM memory featuring 3 cycles per column for reads and 2 for writes and a 40 Mhz clock frequency (using SDRAM would reduce access time to 1 cycle and the frequency could be raised to 60 Mhz). Runtime modification of σ, W_{ref}, and S allowed detailed performance analysis.

Figure 3 shows the processing duration of a single binary NOT node while varying σ *(1..4)*, W_{ref} *(H=W)*, and *S*. Each surface is then associated with a number of PP assigned to processing in a SPMD fashion (each PP caches its own strip of input buffer). Here, the more PP, the lighter the surface. The estimated transfer duration is $2\times512^2\times2.5/40$ Mhz=3.2 ms. 4.7ms were measured with 1 PP which slightly increased if more PP were acting in parallel. Here, processing is faster than data transfers and since the DMA is a shared resource, adding more PP increases contention. Still the result we gained is as good as what is obtained in [5] (4.3 ms) with a comparable hardware configuration.

Experiments with an optimized 3×3 convolution confirmed that the reload approach has almost no impact. We measured μ=48 ms for a 512^2 image with 1 PP. The node's kernel requires 14 cycles for 2 pixels that we compare to 15 measured based on $2\times\mu\times freq\times\sigma/W\times H$. When $\sigma=4$, μ=**13** ms yields to an effective kernel of 16 cycles per PP. Transferring the input and output amount Q^I+Q^O itself requires 8 ms. Thus, theoretical performance are very close to what is measured despite extraneous transfers.

We now look at a more complex chain gathering 4 statistical nodes aimed to find the min-max, sum (average), sum of power (variance), and number of data below a threshold. When all PPs execute the entire chain in a SPMD mode, acceleration with 4 PPs is 237% (σ=***1\4***, *S=2048, W=H=512*⇒***27\8*** *ms*). Figure 4 shows that the cache size is a deciding factor towards performance prediction and that, in our example, small cache size can quadruple the processing time (σ=*1, S=****2048\256****, W=H=512*⇒***32\4*** *ms*). When we split the chain to 4 single-nodes each of which is assigned to execute on a dedicated PP (MPMD scheme), we measured a duration of 15ms (σ=*4, S=2048, W=H=512*). This illustrates that enhancing data locality, indeed, improves performance.

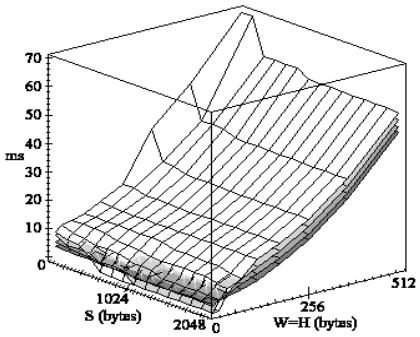


Figure 3: NOT node

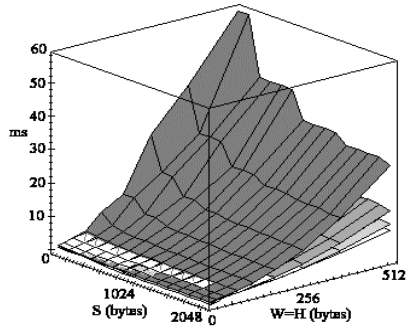


Figure 4: Statistical chain

5. CONCLUSIONS

In the context of multi-processors DSP architectures, we introduced an original software engineering methodology for the real-time implementation of generic low-level image processing chains. In section 2, we introduced the *generic* description of basic operators while stressing cache management issues. In section 3, we detailed an algorithm which permits the practical implementation of *flexible* processing chain using an advanced DMA. We then reviewed experimental *performances* gained while implementing our approach on the TMS320C80.

Within 6 months, we have written more than 50 image processing nodes. This high productivity results from genericity, flexibility and automatic DMA management. Moreover, code optimization [4] and accurate prediction lead to real-time execution of complex chains. This demonstrates that genericity and performance do not oppose and raises our methodology as an attractive alternative to RISC programming techniques that improve data locality and predictability [5][6].

[1] *TMS320C8X System-Level Synopsis*, Texas Instruments SPRU113b.

[2] Y. Kim, *Washington University C80's image processing library*, http://icsl.ee.washington.edu/projects/iclib

[3] *Matrox Genesis library*, http://www.matrox.com

[4] Jihong Kim, Graham Short, *Performance evaluation of register allocator for the advanced DSP of TMS320C80*. Proc of ICASSP'98

[5] M. Lam, E. Rothberg, M. E. Wolf. *The cache performance of blocked algorithms.* Proc. of ASPLOS IV, Apr. 1991.

[6] O. Temam, E. Granston, W. Jalby "*To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts*". Proc. of Supercomputing'93 (ACM), nov. 1993.