A Case Study of Design Space Exploration for Embedded Multimedia Applications on SoCs

Conference Paper *in* Proceedings of the International Workshop on Rapid System Prototyping · July 2006

DOI: 10.1109/RSP.2006.1 · Source: IEEE Xplore

CITATION
1

READS
17

6 authors, including:

Corinne Ancourt
MINES ParisTech
42 PUBLICATIONS **612** CITATIONS
SEE PROFILE

François Irigoin
MINES ParisTech
84 PUBLICATIONS **1,605** CITATIONS
SEE PROFILE

Michel Barreteau
Thales Research and Technology, Palaisea…
23 PUBLICATIONS **76** CITATIONS
SEE PROFILE

Nicolas Museux
Thales Research and Technology, Palaisea…
22 PUBLICATIONS **46** CITATIONS
SEE PROFILE

A case-study of design space exploration for embedded multimedia applications on SoCs*

Isabelle Hurbain, Corinne Ancourt, François Irigoin
École des Mines de Paris
F-77300 Fontainebleau, FRANCE

Michel Barreteau, Juliette Mattioli
THALES Research & Technology
F-91404 Orsay,FRANCE

Frédéric Pasquier
THOMSON R&D France
F-35576 Cesson-Sévigné, FRANCE

Abstract

Embedded real-time multimedia applications usually imply data parallel processing. SIMD processors embedded in SOCs are cost-effective to exploit the underlying parallelism. However, programming applications for SIMD targets requires data placement and operation scheduling which have been proven to be NP-complete problems and beyond present compiler abilities.

In this paper we show how our tool (based on concurrent constraint programming) can be used to explore the design space of a kernel in H.264 standard (video compression). Different cost functions are considered (e.g. execution time, memory occupancy, chip cost ...) to derive different source codes from the same functional specification. Future work includes model refinement as well as full code generation for rapid prototyping of such hardware and software intensive systems.

1. Introduction

Embedded real-time multimedia applications usually imply data parallel processing. Indeed image processing involves vectors or matrices of pixels for instance. Moreover they are often (statically) predictable and well-structured in such a way that the potential parallelism can be extracted at compile time. More than 80% of the execution time is spent in loop nests that consume and produce arrays. Efficiently performing these regular computations obviously is a key issue.

SIMD (Single Instruction Multiple Data Stream) architectures are well suited for this multimedia application domain because they naturally take advantage of stream-oriented parallelism. Even if architecturally speaking this technology can be seen as an old-fashioned one, it remains easily scalable and can be exploited in addition to other techniques. Such SIMD units are now embedded in SoCs and GPPs (e.g. AltiVec technology from Motorola [2] and MultiMedia eXtensions from Intel [1]).

Programming such domain-specific applications for these targets requires data placement and operation scheduling which have been proven to be NP-complete problems and beyond present compiler abilities. Constraint technology [5] enables to the efficient exploration of the combinatorial space of tiling and scheduling. This article does not present our APOTRES tool or its underlying constraint technology [3, 8] but focuses on a case study to show how easily design space is explored.

The next section describes the application and the architecture. Section 3 introduces constraint-based models such as tiling and scheduling, which define the solution space. Then several optimization criteria are considered such as execution time (5) and machine size (6) with technological or real time constraints (7,8). Results are compared in Section 9.

2. Application and Target Machine

2.1. Application Description

We use a fractional sample interpolation of the H.264 standard [10] as running example. In Figure 1 the grey squares represent existing pixels. Pixel **i** is the desired output. A vertical convolution produces pixels such as **cc**, **dd**, **h**, **m**, **ee**, **ff**. An horizontal one produces **j** from these pixels. Finally, **i** is derived from **h** and **j**.

[0]This work is funded by the RNTL DREAM-UP project. URL:http://www.telecom.gouv.fr/rntl/projet/Posters-PDF/RNTL-Poster-DREAM-UP-Thomson.pdf
Corresponding author: isabelle.hurbain@ensmp.fr

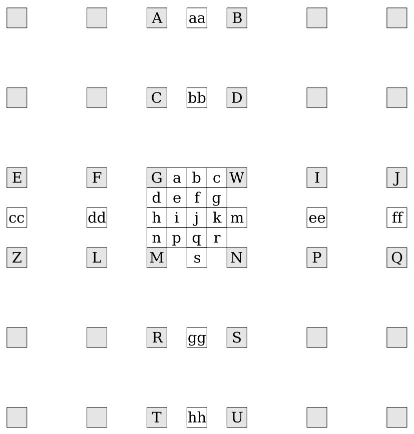

Figure 1. Fractional sample interpolation

For our application we compute 8×8 pixels **i** with the following equations:

$$\begin{aligned} H[i,j] &= conv(X[i-2:i+3,j]) \\ K[i,j] &= conv(H[i,j-2:j+3]) \\ Y[i,j] &= K[i,j] + H[i,j] \end{aligned}$$

where X is the 13×13 input matrix, H is a 8×13 intermediary matrix, K is a 8×8 intermediary matrix, Y is the 8×8 output matrix, and $conv$ is a 6-tap convolution filter.

2.2. Application Code

The code, derived from the equations defining H,K,Y, is composed of parallel loops and is in a single assignment form.

```
float X[0:12][0:12], Y[0:7][0:7], H[0:7][0:12],
      K[0:7][0:7], OUTPUT[0:7][0:7];

for(i0=0; i0<13; i0++)
 for(j0=0; j0<13; j0++)
  X[i0][j0] = input();

for(i1=0; i1<8; i1++)
 for(j1=0; j1<13; j1++)
  H[i][j] = conv(X[i1:i1+5][j1]);

for(i2=0; i2<8; i2++)
 for(j2=0; j2<8; j2++)
  K[i2][j2] = conv(H[i2][j2:j2+5]);

for(i3=0; i3<8; i3++)
 for(j3=0; j3<8; j3++)
  Y[i3][j3] = K[i3][j3] + H[i3][j3];

for(i4=0; i4<8; i4++)
 for(j4=0; j4<8; j4++)
  OUTPUT[i4][j4] = Y[i4][j4];
```

The loop nests are called T_0, T_1, T_2, T_3 and T_4.

2.3. The Target Architecture

The architecture has up to 16 processors, each with a local memory of 256, 512 or 1024 bytes. These parameters are entered as constraints in APOTRES.

A pipelined multiply-add is used and task durations are respectively 1, 9, 9, 4 and 1 cycles.

3. Placement Models

The mapping of applications onto the architecture uses different interacting models: tiling, scheduling, data flow dependences, memory capacity and data communication models. We introduce the main constraints of these models which are used concurrently by the solver to find the set of solutions.

3.1. Tiling

For each loop, the tiling partitions the iteration set and distributes it on three dimensions: (1) a cyclic temporal dimension, (2) a “processor” dimension, (3) a local dimension which exploits the local memory.

The tiling is formally defined in [9]. Let $\mathcal{I}$ be the loop nest iteration set (with n loops) contained in $\mathbb{Z}^n$ defined by $\mathcal{I} = \{0,\ldots,b_1-1\} \times \cdots \times \{0,\ldots,b_n-1\}$ where $1 \leq k \leq n, b_k \in \mathbb{N}^*$. Let P and L be $n \times n$ square diagonal integer matrices with non-null determinant. Then for each point i of $\mathcal{I}$, there exists one and only one triplet (c,p,l) of points of $\mathcal{I}$ such as:

$$i = LPc + Lp + l \tag{1}$$

with

$$\forall l, 0 \leq L^{-1}l < 1 \tag{2}$$

$$\forall p, 0 \leq P^{-1}p < 1 \tag{3}$$

$$\forall i \in \mathcal{I}, 0 \leq i < b$$

$$\det(L) \neq 0$$

$$\det(P) \neq 0 \text{ and } P \text{ diagonal}$$

The associated triplet (c,p,l) can be interpreted as following: at a logic time c, each processor p runs l iterations.

P and L define a tiling of the iteration domain. The solver must find the numerical values of their elements.

3.2. Scheduling

Schedules are computed with respect to tilings. For each loop nest, an affine function represents the schedule. A schedule is legal if it respects the data flow dependence constraints. A schedule function is $d(c) = \alpha.c + \beta$ where c is the index of a computation block, α is a line vector of the same dimension as c, . is the standard scalar product and β is an integer. The solver selects values for all α and β.

3.3. Data Flow Dependence

If a block c' uses a value defined by a block c, block c must be executed first. This is called a data flow dependence. If there is a data flow dependence between two computation blocks c and c', then any legal schedule meets the constraint $d(c) < d(c')$.

3.4. Memory Capacity

A simple capacitive memory model is used. As each processor executes the same code, the required memory size is the same for each processor. Each task is allocated an input buffer, but all tasks share a common output buffer. As soon as they are computed, results are sent to all input buffers of tasks using them. The output buffer is sized to fit the output of any task and to support a flip-flop mechanism used to overlap computations and communications. The size of a task input buffer is the sum of the spaces required for each argument multiplied by the maximal number of live iterations. Liveness information is carried by the dependences and the schedule.

The memory constraints are linked to the partitioning, dependence and scheduling parameters. An example of code with memory allocation is given in Section 7.

3.5. Communications

Different communication models have been implemented in our tool. However, to simplify the presentation, we choose not to take data communications into account.

4. Optimization criteria

To show the characteristics of our tool, we present in the next four sections, various placements of the application depending on four different optimization criteria:

- execution time minimization (Section 5),
- cost minimization (Section 6),
- cost minimization under memory constraint (Section 7),
- cost minimization under execution time constraint(Section 8).

5. Execution Time Minimization

5.1. Tilings for Minimal Execution Time

To minimize the execution time, APOTRES selects the following tilings:

Task	# blocks	# processors	# local iterations
T_0	13	13	1
T_1	8	13	1
T_2	4	2×8	1
T_3	4	2×8	1
T_4	4	2×8	1

Table 1. Tilings for Execution Time Minimization

All 16 processors are used to exploit the parallelism available in tasks T_3 and T_4.

As explained in Section 3.1, tilings map loop nests on the time, processor and local memory dimensions.

As an example, let us consider the $(3, 6)$ iteration of the T_1 loop nest. To know on which processor it is mapped, Equation 1 is instantiated with the coefficients given by our tool:

$$\begin{pmatrix} 3 \\ 6 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 13 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} + \begin{pmatrix} l_1 \\ l_2 \end{pmatrix}$$

which gives us the following equations:

$$\left\{ \begin{array}{rcl} c_1 + p_1 + l_1 & = & 3 \\ 13c_2 + p_2 + l_2 & = & 6 \end{array} \right.$$

Condition 2 implies that $l1 = l2 = 0$, while Condition 3 gives us that $p1 = 0$ and $p2 < 13$. This also indicates us that 13 processors are used in the tiling of this application.

Consequently we have $c1 = 3$, $c2 = 0$ and $p2 = 6$. The iteration $(3, 6)$ of the T_1 loop nest is executed on the processor mapped to the $(0, 6)$ processor.

The tiled solution is expressed by the following parallelized code:

```
float X[0:12][0:12], Y[0:7][0:7], H[0:7][0:12],
      K[0:7][0:7], OUTPUT[0:7][0:7];

for(c=0; c<13; c++)
 forall(p=0; p<13; p++)
  X[c][p] = input();

for(c=0; c<8; c++)
 forall(p=0; p<13; p++)
  H[c][p]=conv(X[c:c+5][p]);

for(c1=0; c1<4; c1++)
 forall(p1=0;p1<2;p1++)
  forall(p2=0; p2<8; p2++)
   K[2*c1+p1][p2]=
     conv(H[2*c1+p1][p2:p2+5]);
```

```
for(c1=0; c1<4; c1++)
 forall(p1=0;p1<2;p1++)
  forall(p2=0; p2<8; p2++)
   Y[2*c1+p1][p2]=
     K[2*c1+p1][p2]+H[2*c1+p1][p2];

for(c1=0; c1<4; c1++)
 forall(p1=0;p1<2;p1++)
  forall(p2=0; p2<8; p2++)
   OUTPUT[2*c1+p1][p2]=Y[2*c1+p1][p2];
```

Note that l does not appear because $L = I$ (identity matrix) for each task.

5.2. Schedule for Minimal Execution Time

The schedule solution, represented in Figure 2 in which tasks are colored (red, blue, black, green, purple), is:

$d_{T_0}(c) = 5.c$
$d_{T_1}(c) = 5.c + 61$
$d_{T_2}(c) = 10.c + 67$
$d_{T_3}(c) = 10.c + 68$
$d_{T_4}(c) = 10.c + 69$

All initializations are done first. The other tasks are pipelined.

Note that nothing happens at some time steps. For instance, at time 4, no iteration is scheduled. The logical duration in Figure 2 is 33 events, although the final date is 99.

The execution time, computed by the solver, is 141 cycles. Each of the 33 events lasts between one and nine cycles depending on the computation performed.

The schedule reorders the tiled code shown in Section 5.1. The main transformation is the fusion of the c loops:

```
float X[0:12][0:12], Y[0:7][0:7], H[0:7][0:12],
      K[0:7][0:7], OUTPUT[0:7][0:7];

for(c=0; c<13; c++)
  forall(p=0; p<13; p++)
    X[c][p] = input();

for(c1=0; c1<4; c1++){

 for(c2=0; c2<2; c++)
  forall(p=0; p<13; p++)
   H[2*c1+c2][p]=
     conv(X[2*c1+c2:2*c1+c2+5][p]);

 forall(p1=0;p1<2;p1++)
   forall(p2=0; p2<8; p2++)
    K[2*c1+p1][p2]=
      conv(H[2*c1+p1][p2:p2+5]);

 forall(p1=0;p1<2;p1++)
   forall(p2=0; p2<8; p2++)
    Y[2*c1+p1][p2]=
      K[2*c1+p1][p2]+H[2*c1+p1][p2];

 forall(p1=0;p1<2;p1++)
   forall(p2=0; p2<8; p2++)
     OUTPUT[2*c1+p1][p2]=Y[2*c1+p1][p2];
}
```

5.3. Memory Capacity

The memory capacity required per processor is 59 bytes. It is derived by the solver using the longest data dependence arc and the schedule period [9] which together define data liveness. The total memory capacity for 16 processors is 944 bytes.

6. Cost Minimization

The cost of SoCs is important for industrial exploitation. It depends on the cost of a single processor, on the cost of a memory unit, and on the number of processors and memory units that are required by the application.

6.1. Tilings

APOTRES selects a one-processor target machine, as could be expected.

Task	# blocks	# processors	# local iterations
T_0	13	1	13
T_1	8	1	13
T_2	8	1	8
T_3	8	1	8
T_4	8×8	1	1

Table 2. Tilings for Cost Minimization

6.2. Schedule for Cost Minimization

APOTRES provides a schedule *as soon as possible*, represented in Figure 3:

$d_{T_0}(c) = 5.c$
$d_{T_1}(c) = 5.c + 26$
$d_{T_2}(c) = 5.c + 27$
$d_{T_3}(c) = 5.c + 28$
$d_{T_4}(c) = 40.c_1 + 5.c_2 + 29$

The logical duration is 465, and the execution time is 2001 cycles, while the memory capacity is 574 bytes.

7. Cost Minimization Under Memory Constraint

Here we consider that the maximum memory for a processor is not 1024 bytes anymore, but 512 bytes. The previous solution cannot then be used.

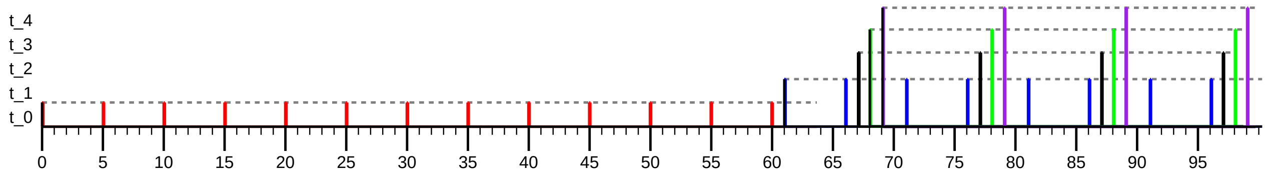

Figure 2. Schedule for execution time minimization

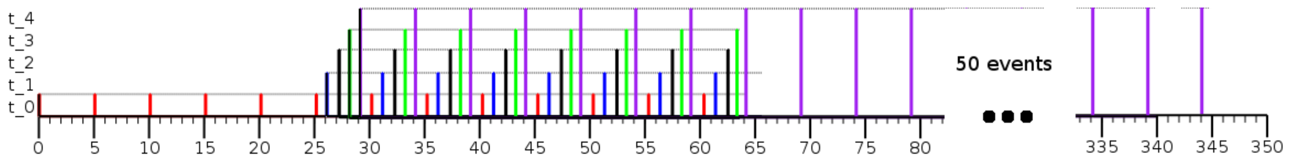

Figure 3. Schedule for Cost Minimization

7.1. Tilings

The best solution found by APOTRES uses 2 processors:

Task	# blocks	# processors	# local iterations
T_0	13	1	13
T_1	13	1	8
T_2	2×8	2	2
T_3	2×2	2	4×2
T_4	8×4	2	1

Table 3. Tilings Under Memory Constraint

7.2. Schedule

The schedule, represented in Figure 4, is specific. Tasks T_0 and T_1 are coupled. Although the constant coefficients (β) are non contiguous, it still is a schedule as soon as possible:

$d_{T_0}(c) = 5.c$
$d_{T_1}(c) = 5.c + 1$
$d_{T_2}(c) = 40.c_1 + 5.c_2 + 27$
$d_{T_3}(c) = 40.c_1 + 20.c_2 + 43$
$d_{T_4}(c) = 40.c_1 + 10.c_2 + 44$

The logical duration is 369, and the execution time is 1553 cycles.

In order to understand how memory allocation is performed, the scheduled code is given below:

```
float X[0:12][0:12], Y[0:7][0:7], H[0:7][0:12],
      K[0:7][0:7], OUTPUT[0:7][0:7];

for(c=0; c<13; c++)
  for(l=0; l<13; l++)
    X[l][c] = input();

for(c=0; c<13; c++)
  for(l=0; l<8; l++)
   H[l][c]=conv(X[l:l+5][c]);

for(c1=0; c1<2; c1++)
 for(c2=0; c2<8; c2++)
  forall(p=0;p<2;p++)
   for(l=0; l<2; l++)
    K[2*c1+2*p+l][c2]=
      conv(H[2*c1+2*p+l][c2:c2+5]);

for(c1=0; c1<2; c1++)
 for(c2=0; c2<2; c2++)
  forall(p=0;p<2;p++)
   for(l1=0; l1<4; l1++)
     for(l2=0; l2<2; l2++)
    Y[4*c1+l1][4*c2+2*p+l2]=
        K[4*c1+l1][4*c2+2*p+l2]+
        H[4*c1+l1][4*c2+2*p+l2];

for(c1=0; c1<8; c1++)
 for(c2=0; c2<4; c2++)
  forall(p=0;p<2;p++)
    OUTPUT[c1][2*c2+p]=Y[c1][2*c2+p];
```

7.3. Memory capacity for minimal execution time

This section presents the resulting code after memory allocation.

The output buffer is `B_W`. It is doubled to allow the flip-flop mechanism used to overlap communications and computations. The integer modulos are not simplified, to show the number of allocated buffers for each used array.

Every buffer is indexed by

1. the number of the virtual processor on which the computation is executed,
2. the index number of the tiled computation block that is

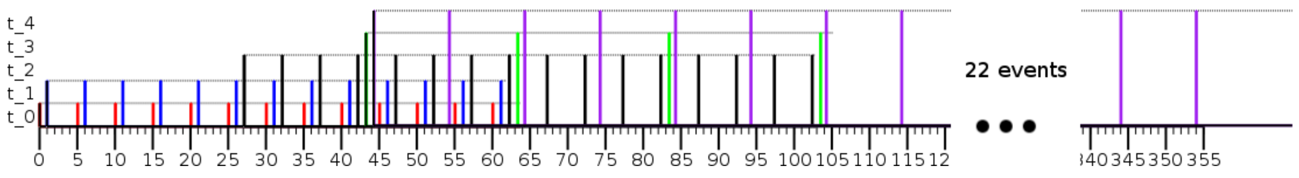

Figure 4. Schedule for cost minimization under memory constraint

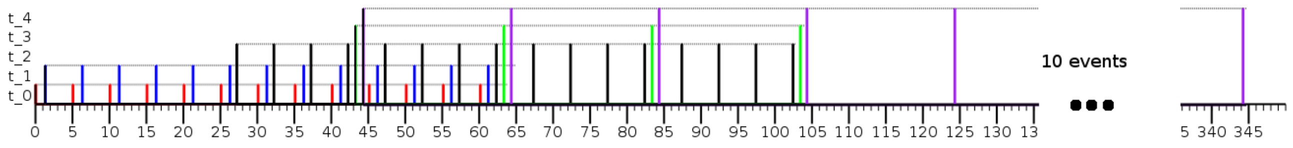

Figure 5. Schedule for cost minimization under execution time constraint

executed,

3. the executed local iteration indices, that also reference the elements of the local buffers.

With this information, the communication code can be automatically generated using the techniques presented in [4].

```
float B_W[0:0][0:1][0:12],B_X_1[0:0][0:0][0:12],
    B_H_2[0:0][0:13][0:11], B_H_3[0:0][0:3][0:7],
    B_K_3[0:0][0:0][0:7], B_Y_4[0:0][0:25][0:0]

for(c=0; c<13; c++)
  for(l=0; l<13; l++)
    B_W[p][c%2][l] = input();

for(c=0; c<13; c++)
  for(l=0; l<8; l++)
    B_W[p][c%2][l] = conv(B_X_1[p][c%1][l:l+5])

for(c1=0; c1<2; c1++)
 for(c2=0; c2<8; c2++)
   for(l=0; l<2; l++)
     B_W[p][(8*c1+c2)%2][l]=
       conv(B_H_2[p][(8*c1+c2)%14][6*l:6*l+5]);

for(c1=0; c1<2; c1++)
 for(c2=0; c2<2; c2++)
   for(l1=0; l1<4; l1++)
     for(l2=0; l2<2; l2++)
        B_W[p][(2*c1+c2)%2][4*l2+l1]=
          B_K_3[p][(2*c1+c2)%1][4*l2+l1]
          +B_H_3[p][(2*c1+c2)%4][4*l2+l1];

for(c1=0; c1<8; c1++)
 for(c2=0; c2<4; c2++)
    l=0
    B_W[p][(4*c1+c2)%2][l]=
      B_Y_4[p][(4*c1+c2)%26][l];
}
```

The memory capacity per processor is 273 bytes. The total memory capacity is 546 bytes.

8. Cost Minimization under Execution Time Constraint

Multimedia applications often must meet real-time constraints. Here we wish to set the execution time to a value strictly less than the 1553 cycles found in the previous solution.

8.1. Tilings

To reduce the execution time, APOTRES has to increase the number of processors from two to four.

Task	# blocks	# processors	# local iterations
T_0	13	1	13
T_1	13	2	4
T_2	2×8	4	1
T_3	2×2	4	4
T_4	8×2	4	1

Table 4. Tilings under Execution Time Constraint

8.2. Schedule

The schedule, represented in Figure 5, is nearly the same as the previous one:

$d_{T_0}(c) = 5.c$
$d_{T_1}(c) = 5.c + 1$
$d_{T_2}(c) = 40.c_1 + 5.c_2 + 27$
$d_{T_3}(c) = 40.c_1 + 20.c_2 + 43$
$d_{T_4}(c) = 40.c_1 + 20.c_2 + 44$

Cost Function	# procs	Duration	Local memory	Total memory	Efficiency
Execution time	16	141	59	944	0.89
Machine cost	1	2001	574	574	1.00
Machine cost under memory constraint	2	1553	273	546	0.64
Machine cost under execution time constraint	4	861	171	684	0.58

Table 5. Comparative table for the 4 solutions

The logical duration is 269, and the execution time is 861 cycles. The memory capacity per processor is 171 bytes. The total memory capacity is 684 bytes.

9. Discussion

Table 5 summarizes the different solutions depending of the cost function used and on additional constraints, hardware or software. Typically, to minimize the silicium area of a SoC, the two main components to take into account are the number of processors and the memory size attributed to each processor. Only considering these two criteria yields the solution with 1 processor and 574 bytes of local memory. Unfortunately, this solution is disastrous in terms of number of cycles for a real-time embedded application such as video decoding. So the number of cycles should also be considered. Furthermore, decreasing the number of cycles required to execute an application will allow to decrease the SoC frequency and thus to reduce the SoC surface by tightening the layers. The best solution seems to be four processors with 171 bytes memory.

The current search heuristics used by APOTRES fails sometimes. Two and four processors should be allocated to Task T_1 in Tables 3 and 4.

10. Conclusion

This article shows how a multimedia application can be rapidly prototyped onto a SIMD architecture: Our mapping tool is able to explore the tiling and scheduling spaces among the combinatorial space of solutions according to different criteria. It enables to find quickly (the tool runs in a few minutes) the best trade-off depending on the embedded real-time constraints. APOTRES is connected to PIPS [7], a tool that automatically analyzes and transforms codes written in a Fortran. Another potential input is ANSI-C code whose functional results can be checked (by THOMSON). Hence our prototyping chain is nearly seamless in the sense that a multimedia code can be parallelized from any standard specification (sometimes not parallel at all) translated “à la Fortran” or from a C sequential code.

To make APOTRES more useful, some improvements are needed in three different axes: memory capacity, data communication and code generation (control, allocation, communication). The current memory model is not flexible enough to support the variety of memory allocation schemes used by programmers. Communications have to be modelled with respect to the communication resources. APOTRES generates integer values which are interpreted as mapping directives. We are currently studying control generation using CLooG [6], which generates an efficient pseudo-code (in C for instance) from a description of iteration domains and schedules. APOTRES uses a heuristic to search the design space. It should be tuned according to the target architecture.

References

[1] http://www.intel.com/design/intarch/mmx/docs_mmx.htm.

[2] http://www.simdtech.org/altivec.

[3] C. Ancourt, D. Barthou, C. Guettier, F. Irigoin, B. Jeannet, J. Jourdan, and J. Mattioli. Automatic data mapping of signal processing applications. *IEEE International Conference on Application Specific Systems, Architectures and Processors*, pages 350–362, 1997.

[4] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static hpf code distribution. In *Fourth Workshop on Compilers for Parallel Computers, CPC’93*, Delft, Pays-Bas, 1993.

[5] K. R. Apt. *Principles of Constraint Programming*. Cambridge University press, 2003.

[6] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC’2 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubjana, october 2003.

[7] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the pips project. In *ACM International Conference on Supercomputing, ICS’91*, Cologne, Allemagne, June 1991.

[8] J. Mattioli, N. Museux, J. Jourdan, P. Savéant, and S. de Givry. A constraint optimization framework for mapping a digital signal processing application onto a parallel architecture. In *Principles and Practice of Constraint Programming*, 2000.

[9] N. Museux. Aide au placement d’applications de traitement du signal sur machines paralleles multi-spmd, 2001.

[10] T. Wiegand, G. Sullivan, and A. Luthra. Itu-t rec. h.264 — iso/iec 14496-10 avc - final draft. Technical report, Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, May 2003.